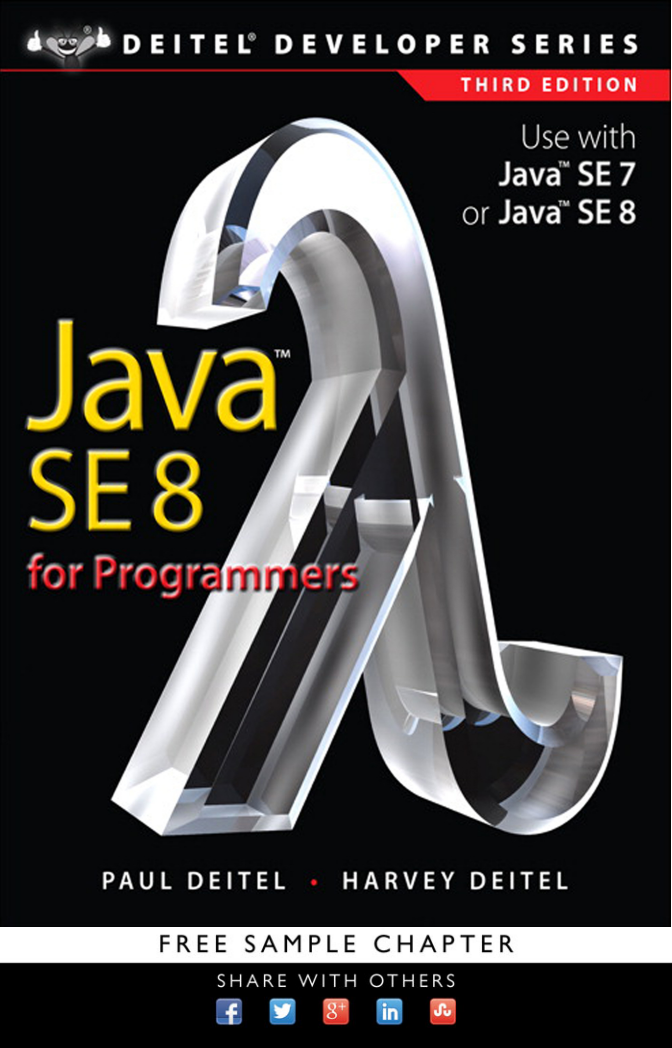
DEITEL® DEVELOPER SERIES
THIRD EDITION
Use with
Java™ SE 7
or Java™ SE 8
Java™
SE 8
for Programmers
PAUL DEITEL • HARVEY DEITEL
FREE SAMPLE CHAPTER
SHARE WITH OTHERS

JAVA™ SE 8 FOR PROGRAMMERS

THIRD EDITION

DEITEL® DEVELOPER SERIES

Library of Congress Cataloging-in-Publication Data

On file

ISBN-13: 978-0-13389138-6
ISBN-10: 0-13-389138-0

Text printed in the United States at Edwards Brothers Malloy in Ann Arbor, Michigan.
First printing, March 2014

JAVA™ SE 8 FOR PROGRAMMERS THIRD EDITION

DEITEL® DEVELOPER SERIES

Paul Deitel • Harvey Deitel
Deitel & Associates, Inc.

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Deitel® Series Page

Deitel® Developer Series

Android for Programmers: An App-Driven Approach, 2/E, Volume 1
C for Programmers with an Introduction to C11
C++11 for Programmers
C# 2012 for Programmers
Dive Into® iOS 6 for Programmers: An App-Driven Approach
Java™ for Programmers, 3/E
JavaScript for Programmers

How To Program Series

Android How to Program, 2/E
C++ How to Program, 9/E
C How to Program, 7/E
Java™ How to Program, 10/E
Java™ How to Program, Late Objects Version, 10/E
Internet & World Wide Web How to Program, 5/E
Visual C++® 2008 How to Program, 2/E
Visual Basic® 2012 How to Program, 6/E
Visual C#® 2012 How to Program, 5/E

Simply Series

Simply C++: An App-Driven Tutorial Approach
Simply Java™ Programming: An App-Driven Tutorial Approach

(continued from previous column)

Simply C#: An App-Driven Tutorial Approach
Simply Visual Basic® 2010: An App-Driven Approach, 4/E

CourseSmart Web Books

`www.deitel.com/books/CourseSmart/`

C++ How to Program, 8/E and 9/E
Simply C++: An App-Driven Tutorial Approach
Java™ How to Program, 9/E and 10/E
Simply Visual Basic® 2010: An App-Driven Approach, 4/E
Visual Basic® 2012 How to Program, 6/E
Visual Basic® 2010 How to Program, 5/E
Visual C#® 2012 How to Program, 5/E
Visual C#® 2010 How to Program, 4/E

LiveLessons Video Learning Products

`www.deitel.com/books/LiveLessons/`

Android App Development Fundamentals
C++ Fundamentals
Java™ Fundamentals
C# 2012 Fundamentals
C# 2010 Fundamentals
iOS® 6 App Development Fundamentals
JavaScript Fundamentals
Visual Basic® Fundamentals

To receive updates on Deitel publications, Resource Centers, training courses, partner offers and more, please join the Deitel communities on

- Facebook®—`facebook.com/DeitelFan`
- Twitter®—`@deitel`
- Google+™—`google.com/+DeitelFan`
- YouTube™—`youtube.com/DeitelTV`
- LinkedIn®—`linkedin.com/company/deitel-&-associates`

and register for the free *Deitel® Buzz Online* e-mail newsletter at:

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

To communicate with the authors, send e-mail to:

`deitel@deitel.com`

For information on *Dive-Into® Series* on-site seminars offered by Deitel & Associates, Inc. worldwide, write to us at `deitel@deitel.com` or visit:

`www.deitel.com/training/`

For continuing updates on Pearson/Deitel publications visit:

`www.deitel.com`
`www.pearsonhighered.com/deitel/`

Visit the Deitel Resource Centers that will help you master programming languages, software development, Android and iOS app development, and Internet- and web-related topics:

`www.deitel.com/ResourceCenters.html`

To Brian Goetz,
Oracle's Java Language Architect and
Specification Lead for Java SE 8's Project Lambda:

Your mentorship helped us make a better book.
Thank you for insisting that we get it right.

Paul and Harvey Deitel

Trademarks

DEITEL, the double-thumbs-up bug and DIVE-INTO are registered trademarks of Deitel & Associates, Inc.

Java is a registered trademark of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Google, Android, Google Play, Google Maps, Google Wallet, Nexus, YouTube, AdSense and AdMob are trademarks of Google, Inc.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided "as is" without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. Screen shots and icons reprinted with permission from the Microsoft Corporation. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

Throughout this book, trademarks are used. Rather than put a trademark symbol in every occurrence of a trademarked name, we state that we are using the names in an editorial fashion only and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Contents

This page intentionally left blank

Foreword

I've been enamored with Java even prior to its 1.0 release in 1995, and have subsequently been a Java developer, author, speaker, teacher and Oracle Java Technology Ambassador. In this journey, it has been my privilege to call Paul Deitel a colleague, and to often leverage and recommend his Java books. In their many editions, these books have proven to be great texts for college and professional courses that I and others have developed to teach the Java programming language.

One of the qualities that makes *Java SE 8 for Programmers, 3/e*, a great resource is its thorough and insightful coverage of Java concepts. Another useful quality is its treatment of concepts and practices essential to effective software development.

I'd like to point out some of the features of this new edition about which I'm most excited:

- An ambitious new chapter on Java lambda expressions and streams. This chapter starts out with a primer on functional programming, and introduces Java lambda expressions and how to use streams to perform functional programming tasks on collections.
- Although concurrency has been addressed since the first edition of the book, it is increasingly important because of multi-core architectures. There are timing examples—using the new Date/Time API classes introduced in Java SE 8—in the concurrency chapter that show the performance improvements with multi-core over single-core.
- JavaFX is Java's GUI/graphics/multimedia technology moving forward, so it is nice to see JavaFX introduced in the Deitel live-code pedagogic style.

Please join me in congratulating Paul and Harvey Deitel on their latest edition of a wonderful resource for software developers!

James L. Weaver
Java Technology Ambassador
Oracle Corporation

This page intentionally left blank

Preface

Welcome to Java and *Java SE 8 for Programmers, Third Edition*! This book presents leading-edge computing technologies for software developers.

We focus on software engineering best practices. At the heart of the book is the Deitel signature "live-code approach"—rather than using code snippets, we present concepts in the context of complete working programs that run on recent versions of Windows®, Linux® and OS X®. Each complete code example is accompanied by live sample executions. All the source code is available at

```
http://www.deitel.com/books/javafp3/
```

Keeping in Touch with the Authors

As you read the book, if you have questions, send an e-mail to us at

```
deitel@deitel.com
```

and we'll respond promptly. For updates on this book, visit

```
http://www.deitel.com/books/jfp3
```

subscribe to the *Deitel® Buzz Online* newsletter at

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

and join the Deitel social networking communities on

- Facebook® (`http://www.deitel.com/deitelfan`)
- Twitter® (`@deitel`)
- Google+™ (`http://google.com/+DeitelFan`)
- YouTube® (`http://youtube.com/DeitelTV`)
- LinkedIn® (`http://linkedin.com/company/deitel-&-associates`)

Modular Organization

Java SE 8 for Programmers, 3/e, is appropriate for programmers with a background in high-level language programming. It features a modular organization:

Introduction

- Chapter 1, Introduction to Java and Test-Driving a Java Application
- Chapter 2, Introduction to Java Applications; Input/Output and Operators
- Chapter 3, Introduction to Classes, Objects, Methods and Strings

Additional Programming Fundamentals

- Chapter 4, Control Statements: Part 1; Assignment, ++ and -- Operators
- Chapter 5, Control Statements: Part 2; Logical Operators
- Chapter 6, Methods: A Deeper Look
- Chapter 7, Arrays and ArrayLists
- Chapter 14, Strings, Characters and Regular Expressions
- Chapter 15, Files, Streams and Object Serialization

Object-Oriented Programming

- Chapter 8, Classes and Objects: A Deeper Look
- Chapter 9, Object-Oriented Programming: Inheritance
- Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces
- Chapter 11, Exception Handling: A Deeper Look

Swing and JavaFX Graphical User Interfaces; Java 2D Graphics

- Chapter 12, Swing GUI Components: Part 1
- Chapter 13, Graphics and Java 2D
- Chapter 19, Swing GUI Components: Part 2
- Chapter 22, JavaFX GUI

Generic Collections, Lambdas and Streams

- Chapter 16, Generic Collections
- Chapter 17, Java SE 8 Lambdas and Streams
- Chapter 18, Generic Classes and Methods

Concurrency/Database

- Chapter 20, Concurrency
- Chapter 21, Accessing Databases with JDBC

Object-Oriented Design

- Chapter 23, ATM Case Study, Part 1: Object-Oriented Design with the UML
- Chapter 24, ATM Case Study Part 2: Implementing an Object-Oriented Design

New and Updated Features

Here are the updates we've made for *Java SE 8 for Programmers, 3/e*:

- ***Easy to use with Java SE 7 or Java SE 8.*** This book was published coincident with the release of Java SE 8. To meet the needs of our diverse audiences, we designed the book for professionals interested in Java SE 7, Java SE 8 or a mixture

of both. The Java SE 8 features (Fig. 4.1) are covered in Chapter 17 and in easy-to-include-or-omit sections book wide.

Java SE 8 features
Lambda expressions
Type-inference improvements
`@FunctionalInterface` annotation
Parallel array sorting
Bulk data operations for Java Collections—`filter`, `map` and `reduce`
Library enhancements to support lambdas (e.g., `java.util.stream`, `java.util.function`)
Date & Time API (`java.time`)
Java concurrency API improvements
`static` and `default` methods in interfaces
Functional interfaces—interfaces that define only one `abstract` method and can include `static` and `default` methods
JavaFX enhancements

Fig. 4.1 | Java SE 8 features we discuss.

- ***Java SE 8 lambdas, streams, and interfaces with `default` and `static` methods.*** The most significant new features in Java SE 8 are lambdas and complementary technologies. In Chapter 17, you'll see that functional programming with lambdas and streams can help you write programs faster, more concisely, more simply, with fewer bugs and that are easier to parallelize (to get performance improvements on multi-core systems) than programs written with previous techniques (Fig. 4.2). You'll see that functional programming complements object-oriented programming.

Pre-Java-SE-8 topics	Corresponding Java SE 8 discussions and examples
Chapter 7, Arrays and ArrayLists	Sections 17.3–17.4 introduce basic lambda and streams capabilities that process one-dimensional arrays.
Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces	Section 10.10 introduces the new Java SE 8 interface features (`default` methods, `static` methods and the concept of functional interfaces) that support functional programming with lambdas and streams.
Chapters 12 and 19, Swing GUI Components: Parts 1 and 2	Section 17.9 shows how to use a lambda to implement a Swing event-listener functional interface.
Chapter 14, Strings, Characters and Regular Expressions	Section 17.5 shows how to use lambdas and streams to process collections of `String` objects.

Fig. 4.2 | Java SE 8 lambdas and streams discussions and examples. (Part 1 of 2.)

Pre-Java-SE-8 topics	Corresponding Java SE 8 discussions and examples
Chapter 15, Files, Streams and Object Serialization	Section 17.7 shows how to use lambdas and streams to process lines of text from a file.
Chapter 20, Concurrency	Shows that functional programs are easier to parallelize so that they can take advantage of multi-core architectures to enhance performance. Demonstrates parallel stream processing. Shows that `Arrays` method `parallelSort` improves performance on multi-core architectures when sorting large arrays.
Chapter 22, JavaFX GUI	Section 22.5.5 shows how to use a lambda to implement a JavaFX event-listener functional interface.

Fig. 4.2 | Java SE 8 lambdas and streams discussions and examples. (Part 2 of 2.)

- ***Java SE 7's `try`-with-resources statement and the `AutoClosable` interface.*** `AutoClosable` objects reduce the likelihood of resource leaks when you use them with the `try`-with-resources statement, which automatically closes the `AutoClosable` objects. In this edition, we use `try`-with-resources and `AutoClosable` objects as appropriate starting in Chapter 15, Files, Streams and Object Serialization.
- ***Java security.*** We audited our book against the CERT Oracle Secure Coding Standard for Java:

```
http://bit.ly/CERTOracleSecureJava
```

  See this Preface's Secure Java Programming section for more about CERT.
- ***Java NIO API.*** We updated the file-processing examples in Chapter 15 to use features from the Java NIO (new IO) API.
- ***Java Documentation.*** Throughout the book, we provide links to Java documentation where you can learn more about various topics that we present. For Java SE 7 documentation, the links begin with

```
http://docs.oracle.com/javase/7/
```

  and for Java SE 8 documentation, the links begin with

```
http://download.java.net/jdk8/
```

  These links could change when Oracle releases Java SE 8—*possibly* to links beginning with

```
http://docs.oracle.com/javase/8/
```

  For any links that change after publication, we'll post updates at

```
http://www.deitel.com/books/jfp3
```

Swing and JavaFX GUI; Java 2D Graphics

- ***Swing GUI and Java 2D graphics.*** Java's Swing GUI is discussed in Chapters 12 and 19. Swing is now in maintenance mode—Oracle has stopped development

and will provide only bug fixes going forward, however it will remain part of Java and is still widely used. Most of GUI-based legacy code in industry uses Swing GUI. Chapter 13 discusses Java 2D graphics.

- ***JavaFX GUI.*** Java's GUI, graphics and multimedia technology going forward is JavaFX. In Chapter 22, we use JavaFX 2.2 with Java SE 7. We use Scene Builder—a drag-and-drop tool for creating JavaFX GUIs quickly and conveniently. It's a standalone tool that you can use separately or with Java IDEs.

Concurrency

- ***Concurrency for optimal multi-core performance.*** In this edition, we were privileged to have as a reviewer Brian Goetz, co-author of *Java Concurrency in Practice* (Addison-Wesley). We updated Chapter 20, Concurrency, with Java SE 8 technology and idiom. We added a `parallelSort` vs. `sort` example that uses the Java SE 8 Date/Time API to time each operation and demonstrate `parallelSort`'s better performance on a multi-core system. We include a Java SE 8 parallel vs. sequential stream processing example, again using the Date/Time API to show performance improvements. Finally, we added a Java SE 8 `CompletableFuture` example that compares the relative performance of sequential and parallel execution of long-running calculations.
- ***`SwingWorker` class.*** We use class `SwingWorker` to create multithreaded user interfaces.
- ***Concurrency is challenging.*** There's a great variety of concurrency features. We point out the ones that most developers should use and mention those that should be left to the experts.

Getting Monetary Amounts Right

- ***Monetary amounts.*** In the early chapters, for convenience, we use type `double` to represent monetary amounts. Due to the potential for incorrect monetary calculations with type `double`, class `BigDecimal` (which is a bit more complex) should be used to represent monetary amounts. We demonstrate `BigDecimal` in Chapters 8 and 22.

Object Technology

- ***Object-oriented programming.*** We use an *early objects* approach, reviewing the basic concepts and terminology of object technology in Chapter 1. Readers develop their first customized classes and objects in Chapter 3.
- ***Early objects real-world case studies.*** The early classes and objects presentation features `Account`, `Student`, `AutoPolicy`, `Time`, `Employee`, `GradeBook` and `Card` shuffling-and-dealing case studies, gradually introducing deeper OO concepts.
- ***Inheritance, Interfaces, Polymorphism and Composition.*** We use a series of real-world case studies to illustrate each of these OO concepts and explain situations in which each is preferred in building industrial-strength applications. We discuss Java SE 8's improvements to the interface concept.

- ***Exception handling.*** We integrate basic exception handling early in the book then present a deeper treatment in Chapter 11. Exception handling is important for building "mission-critical" and "business-critical" applications. Programmers need to be concerned with, "What happens when the component I call on to do a job experiences difficulty? How will that component signal that it had a problem?" To use a Java component, you need to know not only how that component behaves when "things go well," but also what exceptions that component "throws" when "things go poorly."
- ***Class `Arrays` and `ArrayList`.*** Chapter 7 covers class `Arrays`—which contains methods for performing common array manipulations—and class `ArrayList`—which implements a dynamically resizable array-like data structure. This follows our philosophy of getting lots of practice using existing classes while learning how to define your own classes.
- ***Case Study: Developing an Object-Oriented Design and Java Implementation of an ATM.*** Chapters 23–24 include a case study on object-oriented design with the UML (Unified Modeling Language™)—the industry-standard graphical language for modeling object-oriented systems. We design and implement the software for a simple automated teller machine (ATM). We analyze a typical requirements document that specifies the system to be built. We determine the classes needed to implement that system, the attributes the classes need to have, the behaviors the classes need to exhibit and specify how the classes must interact with one another to meet the system requirements. From the design we produce a completely coded Java implementation. Participants in our professional Java courses often report having a "light-bulb moment"—the case study helps them "tie it all together" and really understand Java-based object-oriented programming.

Generic Collections

- ***Generic collections presentation.*** We begin with generic class `ArrayList` in Chapter 7. Chapters 16–18 provide a deeper treatment of generic collections—showing how to use the built-in collections of the Java API. We show how to implement generic methods and classes. Lambdas and streams (introduced in Chapter 17) are especially useful for working with generic collections.

Database

- ***JDBC.*** Chapter 21 covers JDBC and uses the Java DB database management system. The chapter introduces Structured Query Language (SQL) and features an OO case study on developing a database-driven address book that demonstrates prepared statements.

Secure Java Programming

It's difficult to build industrial-strength systems that stand up to attacks from viruses, worms, and other forms of "malware." Today, via the Internet, such attacks can be instantaneous and global in scope. Building security into software from the beginning of the development cycle can greatly reduce vulnerabilities. We incorporate various secure Java coding practices into our discussions and code examples.

The CERT® Coordination Center (`www.cert.org`) was created to analyze and respond promptly to attacks. CERT—the Computer Emergency Response Team—is a government-funded organization within the Carnegie Mellon University Software Engineering Institute™. CERT publishes and promotes secure coding standards for various popular programming languages to help software developers implement industrial-strength systems that avoid the programming practices which leave systems open to attack.

We'd like to thank Robert C. Seacord, Secure Coding Manager at CERT and an adjunct professor in the Carnegie Mellon University School of Computer Science. Mr. Seacord was a technical reviewer for our book, *C11 for Programmers*, where he scrutinized our C programs from a security standpoint, recommending that we adhere to the *CERT C Secure Coding Standard*. This experience influenced our coding practices in *C++11 for Programmers* and *Java SE 8 for Programmers, 3/e* as well.

Teaching Approach

Java SE 8 for Programmers, 3/e, contains hundreds of complete working examples. We stress program clarity and concentrate on building well-engineered software.

Syntax Coloring. For readability, we syntax color the code, similar to the way most integrated-development environments and code editors syntax color the code. Our syntax-coloring conventions are:

```
comments appear like this
keywords appear like this
constants and literal values appear like this
errors appear like this
all other code appears in black
```

Code Highlighting. We place yellow rectangles around each program's key code.

Using Fonts for Emphasis. We place the key terms and the index's page reference for each defining occurrence in **bold** text for easier reference. On-screen components are emphasized in the **bold Helvetica** font (e.g., the **File** menu) and Java program text in the `Lucida` font (e.g., `int x = 5;`).

Web Access. All of the source-code examples can be downloaded from:

```
www.deitel.com/books/javafp3
www.pearsonhighered.com/deitel
```

Objectives. The opening quotations are followed by a list of chapter objectives.

Illustrations/Figures. Abundant tables, line drawings, UML diagrams, programs and program outputs are included.

Programming Tips. We include programming tips to help you focus on important aspects of program development. These tips and practices represent the best we've gleaned from a combined seven decades of programming and teaching experience.

Good Programming Practice

The Good Programming Practices *call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.*

Common Programming Error

Pointing out these Common Programming Errors *reduces the likelihood that you'll make them.*

Error-Prevention Tip

These tips contain suggestions for exposing and removing bugs from your programs; many of the tips describe aspects of Java that prevent bugs from getting into programs.

Performance Tip 4.1

These tips highlight opportunities for making your programs run faster or minimizing the amount of memory that they occupy.

Portability Tip

The Portability Tips *help you write code that will run on a variety of platforms.*

Software Engineering Observation

The Software Engineering Observations *highlight architectural and design issues that affect the construction of software systems, especially large-scale systems.*

Look-and-Feel Observation

The Look-and-Feel Observations *highlight graphical-user-interface conventions. These observations help you design attractive, user-friendly graphical user interfaces that conform to industry norms.*

Index. We've included an extensive index. Defining occurrences of key terms are highlighted with a **bold** page number.

Software Used in *Java SE 8 for Programmers, 3/e*

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

We wrote most of the examples in *Java SE 8 for Programmers, 3/e*, using the free Java Standard Edition Development Kit (JDK) 7. For the Java SE 8 modules, we used the OpenJDK's early access version of JDK 8. In Chapter 22, we also used the Netbeans IDE. See the Before You Begin section that follows this Preface for more information.

Java Fundamentals: Parts I, II, III and IV LiveLessons, Second Edition, Video Training Product

Our *Java Fundamentals: Parts I, II, III and IV* LiveLessons,2/e (summer 2014), video training product shows you what you need to know to start building robust, powerful software with Java. It includes 30+ hours of expert training synchronized with *Java SE 8 for Programmers, Third Edition*. Visit

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

for information on purchasing Deitel LiveLessons video products online from Informit and Udemy. You may also access our LiveLessons videos if you have a subscription to Safari Books Online (`http://www.safaribooksonline.com`).

Acknowledgments

We'd like to thank Abbey Deitel and Barbara Deitel of Deitel & Associates, Inc. for long hours devoted to this project. Abbey co-authored Chapter 1 and this Preface, and she and Barbara painstakingly researched the new capabilities of Java SE 8.

We're fortunate to have worked on this project with the dedicated publishing professionals at Prentice Hall/Pearson. We appreciate the extraordinary efforts and 19-year mentorship of our friend and professional colleague Mark L. Taub, Editor-in-Chief of Pearson Technology Group. Carole Snyder recruited distinguished members of the Java community to review the manuscript and managed the review process. Chuti Prasertsith designed the cover. John Fuller managed the book's publication.

Reviewers

We wish to acknowledge the efforts of our recent editions reviewers—a distinguished group of Oracle Java team members, Oracle Java Champions, other industry professionals and academics. They scrutinized the text and the programs and provided countless suggestions for improving the presentation.

Third Edition reviewers: Lance Andersen (Oracle Corporation), Dr. Danny Coward (Oracle Corporation), Brian Goetz (Oracle Corporation), Evan Golub (University of Maryland), Dr. Huiwei Guan (Professor, Department of Computer & Information Science, North Shore Community College), Manfred Riem (Java Champion), Simon Ritter (Oracle Corporation), Robert C. Seacord (CERT, Software Engineering Institute, Carnegie Mellon University), Khallai Taylor (Assistant Professor, Triton College and Adjunct Professor, Lonestar College—Kingwood), Jorge Vargas (Yumbling and a Java Champion), Johan Vos (LodgON and Oracle Java Champion) and James L. Weaver (Oracle Corporation and author of *Pro JavaFX 2*).

Other recent editions reviewers: 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), 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), 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).

A Special Thank You to Brian Goetz

We were privileged to have Brian Goetz, Oracle's Java Language Architect and Specification Lead for Java SE 8's Project Lambda, and co-author of *Java Concurrency in Practice*, do a detailed full-book review. He thoroughly scrutinized every chapter, providing extremely helpful insights and constructive comments. Any remaining faults in the book are our own.

Well, there you have it! As you read the book, we'd appreciate your comments, criticisms, corrections and suggestions for improvement. Please address all correspondence to:

```
deitel@deitel.com
```

We'll respond promptly. We hope you enjoy working with *Java SE 8 for Programmers, 3/e*, as much as we enjoyed writing it!

Paul and Harvey 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 designations, 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 over 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. The Deitels' publications have earned international recognition, with translations published in Japanese, German, Russian, Spanish, French, Polish, Italian, Simplified Chinese, Traditional Chinese, Korean, Portuguese, Greek, Urdu and Turkish. Dr. Deitel has delivered hundreds of programming courses to corporate, academic, government and military clients.

About Deitel® & Associates, Inc.

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 Java™, Android app development, Objective-C and iOS app development, 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 39-year publishing partnership with Pearson/Prentice Hall, Deitel & Associates, Inc., publishes leading-edge programming textbooks and professional books in print and a wide range of e-book 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, visit

```
http://www.informit.com/store/sales.aspx
```

This page intentionally left blank

Before You Begin

This section contains information you should review before using this book. Any updates to the information presented here will be posted at:

```
http://www.deitel.com/books/javafp3
```

In addition, we provide Dive-Into® videos (which will be available in time for Fall 2014 classes) that demonstrate the instructions in this Before You Begin section.

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 emphasize on-screen components in a sans-serif bold **Helvetica** font (for example, **File** menu) and to emphasize Java code and commands in a sans-serif `Lucida` font (for example, `System.out.println()`).

Software Used in the Book

All the software you'll need for this book is available free for download from the web. With the exception of the examples that are specific to Java SE 8, all of the examples were tested with the Java SE 7 and Java SE 8 Java Standard Edition Development Kits (JDKs).

Java Standard Edition Development Kit 7 (JDK 7)
JDK 7 for Windows, OS X and Linux platforms is available from:

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

Java Standard Edition Development Kit (JDK) 8
At the time of this publication, the near-final version of JDK 8 for Windows, OS X and Linux platforms was available from:

```
https://jdk8.java.net/download.html
```

Once JDK 8 is released as final, it will be available from:

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

JDK Installation Instructions

After downloading the JDK installer, be sure to carefully follow the JDK installation instructions for your platform at:

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

Though these instructions are for JDK 7, they also apply to JDK 8—you'll need to update the JDK version number in any version-specific instructions.

Setting the PATH Environment Variable

The PATH environment variable on your computer designates which directories the computer searches when looking for applications, such as the applications that enable you to compile and run your Java applications (called javac and java, respectively). *Carefully follow the installation instructions for Java on your platform to ensure that you set the* PATH *environment variable correctly.* The steps for setting environment variables differ by operating system and sometimes by operating system version (e.g., Windows 7 vs. Windows 8). Instructions for various platforms are listed at:

```
http://www.java.com/en/download/help/path.xml
```

If you do not set the PATH variable correctly on Windows and some Linux installations, when you use the JDK's tools, you'll receive a message like:

```
'java' is not recognized as an internal or external command,
operable program or batch file.
```

In this case, go back to the installation instructions for setting the PATH and recheck your steps. If you've downloaded a newer version of the JDK, you may need to change the name of the JDK's installation directory in the PATH variable.

JDK Installation Directory and the `bin` *Subdirectory*

The JDK's installation directory varies by platform. The directories listed below are for Oracle's JDK 7 update 51:

- 32-bit JDK on Windows:
  `C:\Program Files (x86)\Java\jdk1.7.0_51`
- 64-bit JDK on Windows:
  `C:\Program Files\Java\jdk1.7.0_51`
- Mac OS X:
  `/Library/Java/JavaVirtualMachines/jdk1.7.0_51.jdk/Contents/Home`
- Ubuntu Linux:
  `/usr/lib/jvm/java-7-oracle`

Depending on your platform, the JDK installation folder's name might differ if you're using a different update of JDK 7 or using JDK 8. For Linux, the install location depends on the installer you use and possibly the version of Linux that you use. We used Ubuntu Linux. The PATH environment variable must point to the JDK installation directory's bin subdirectory.

When setting the PATH, be sure to use the proper JDK-installation-directory name for the specific version of the JDK you installed—as newer JDK releases become available, the JDK-installation-directory name changes to include an *update version number*. For example, at the time of this writing, the most recent JDK 7 release was update 51. For this version, the JDK-installation-directory name ends with "_51".

Setting the CLASSPATH Environment Variable

If you attempt to run a Java program and receive a message like

```
Exception in thread "main" java.lang.NoClassDefFoundError: YourClass
```

then your system has a CLASSPATH environment variable that must be modified. To fix the preceding error, follow the steps in setting the PATH environment variable to locate the CLASSPATH variable, then edit the variable's value to include the local directory—typically represented as a dot (.). On Windows add

```
.;
```

at the beginning of the CLASSPATH's value (with no spaces before or after these characters). On other platforms, replace the semicolon with the appropriate path separator characters—typically a colon (:).

Setting the JAVA_HOME Environment Variable

The Java DB database software that you'll use in Chapter 21 requires you to set the JAVA_HOME environment variable to your JDK's installation directory. The same steps you used to set the PATH may also be used to set other environment variables, such as JAVA_HOME.

Java Integrated Development Environments (IDEs)

There are many Java integrated development environments that you can use for Java programming. For this reason, we used only the JDK command-line tools for most of the book's examples. We provide Dive-Into® videos (which will be available in time for Fall 2014 classes) that show how to download, install and use three popular IDEs—NetBeans, Eclipse and IntelliJ IDEA. We use NetBeans in Chapter 22.

NetBeans Downloads
You can download the JDK/NetBeans bundle from:

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

The NetBeans version that's bundled with the JDK is for Java SE development. The online JavaServer Faces (JSF) chapters and web services chapter use the Java Enterprise Edition (Java EE) version of NetBeans, which you can download from:

```
https://netbeans.org/downloads/
```

This version supports both Java SE and Java EE development.

Eclipse Downloads
You can download the Eclipse IDE from:

```
https://www.eclipse.org/downloads/
```

For Java SE development choose the Eclipse IDE for Java Developers. For Java Enterprise Edition (Java EE) development (such as JSF and web services), choose the Eclipse IDE for Java EE Developers—this version supports both Java SE and Java EE development.

IntelliJ IDEA Community Edition Downloads
You can download the free IntelliJ IDEA Community Edition from:

```
http://www.jetbrains.com/idea/download/index.html
```

The free version supports only Java SE development.

Obtaining the Code Examples

The examples for *Java SE 8 for Programmers, 3/e* are available for download at

```
http://www.deitel.com/books/javafp3
```

under the heading **Download Code Examples and Other Premium Content**. The examples are also available from

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

When you download the ZIP archive file, write down the location where you choose to save it on your computer.

Extract the contents of `examples.zip` using a ZIP extraction tool such as 7-Zip (`www.7-zip.org`), WinZip (`www.winzip.com`) or the built-in capabilities of your operating system. Instructions throughout the book assume that the examples are located at:

- `C:\examples` on Windows
- your user account home folder's `examples` subfolder on Linux
- your `Documents` folders `examples` subfolder on Mac OS X

Java's Nimbus Look-and-Feel

Java comes bundled with a cross-platform look-and-feel known as Nimbus. For programs with Swing graphical user interfaces (e.g., Chapters 12 and 19), we configured our test computers to use Nimbus as the default look-and-feel.

To set Nimbus as the default for all Java applications, 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 folders visit `http://docs.oracle.com/javase/7/docs/webnotes/install/index.html`. [*Note:* In addition to the standalone JRE, there's a JRE nested in your JDK's installation folder. If you're using an IDE that depends on the JDK (e.g., NetBeans), you may also need to place the `swing.properties` file in the nested `jre` folder's `lib` folder.]

You're now ready to begin your Java studies with *Java SE 8 for Programmers, 3/e*. We hope you enjoy the book!

3

Introduction to Classes, Objects, Methods and Strings

Objectives

In this chapter you'll:

- Declare a class and use it to create an object.
- Implement a class's behaviors as methods.
- Implement a class's attributes as instance variables.
- Call an object's methods to make them perform their tasks.
- Understand how local variables of a method differ from instance variables.
- Understand what primitive types and reference types are.
- Use a constructor to initialize an object's data.

Outline

3.1 Introduction

[*Note:* This chapter depends on the terminology and concepts discussed in Section 1.2, Object Technology Concepts.]

In Chapter 2, you worked with existing classes, objects and methods. You used the predefined standard output object `System.out`, invoking its methods `print`, `println` and `printf` to display information on the screen. You used the existing `Scanner` class to create an object that reads into memory integer data typed by the user at the keyboard. Throughout the book, you'll use many more preexisting classes and objects.

In this chapter, you'll create your own classes and methods. Each new class you create becomes a new type that can be used to declare variables and create objects. You can declare new classes as needed; this is one reason why Java is known as an *extensible* language.

We present a case study on creating and using a simple, real-world bank account class—`Account`. Such a class should maintain as *instance variables* attributes such as its `name` and `balance`, and provide *methods* for tasks such as querying the balance (`getBalance`), making deposits that increase the balance (`deposit`) and making withdrawals that decrease the balance (`withdraw`). We'll build the `getBalance` and `deposit` methods into the class in the chapter's examples.

In Chapter 2 we used the data type `int` to represent integers. In this chapter, we introduce data type `double` to represent an account balance as a number that can contain a decimal *point*—such numbers are called floating-point numbers. [In Chapter 8, when we get a bit deeper into object technology, we'll begin representing monetary amounts precisely with class `BigDecimal` (package `java.math`) as you should do when writing industrial-strength monetary applications.]

3.2 Instance Variables, *set* Methods and *get* Methods

In this section, you'll create two classes—`Account` (Fig. 3.1) and `AccountTest` (Fig. 3.2). Class `AccountTest` is an *application class* in which the `main` method will create and use an `Account` object to demonstrate class `Account`'s capabilities.

3.2.1 Account Class with an Instance Variable, a *set* Method and a *get* Method

Different accounts typically have different names. For this reason, class `Account` (Fig. 3.1) contains a `name` *instance variable*. A class's instance variables maintain data for each object (that is, each instance) of the class. Later in the chapter we'll add an instance variable named `balance` so we can keep track of how much money is in the account. Class `Account` contains two methods—method `setName` stores a name in an `Account` object and method `getName` obtains a name from an `Account` object.

```
1  // Fig. 3.1: Account.java
2  // Account class that contains a name instance variable
3  // and methods to set and get its value.
4
5  public class Account
6  {
7     private String name; // instance variable
8
9     // method to set the name in the object
10    public void setName(String name)
11    {
12       this.name = name; // store the name
13    }
14
15    // method to retrieve the name from the object
16    public String getName()
17    {
18       return name; // return value of name to caller
19    }
20 } // end class Account
```

Fig. 3.1 | `Account` class that contains a `name` instance variable and methods to *set* and *get* its value.

Class Declaration

The *class declaration* begins in line 5. The keyword `public` (which Chapter 8 explains in detail) is an **access modifier**. For now, we'll simply declare every class `public`. Each `public` class declaration must be stored in a file having the *same* name as the class and ending with the `.java` filename extension; otherwise, a compilation error will occur. Thus, `public` classes `Account` and `AccountTest` (Fig. 3.2) *must* be declared in the *separate* files `Account.java` and `AccountTest.java`, respectively.

Every class declaration contains the keyword `class` followed immediately by the class's name—in this case, `Account`. Every class's body is enclosed in a pair of left and right braces as in lines 6 and 20 of Fig. 3.1.

Identifiers and Camel Case Naming

Class names, method names and variable names are all *identifiers* and by convention all use the same *camel case* naming scheme we discussed in Chapter 2. Also by convention, class

names begin with an initial *uppercase* letter, and method names and variable names begin with an initial *lowercase* letter.

Instance Variable `name`

Recall that an object has attributes, implemented as instance variables and carried with it throughout its lifetime. Instance variables exist before methods are called on an object, while the methods are executing and after the methods complete execution. Each object (instance) of the class has its *own* copy of the class's instance variables. A class normally contains one or more methods that manipulate the instance variables belonging to particular objects of the class.

Instance variables are declared *inside* a class declaration but *outside* the bodies of the class's methods. Line 7

```
private String name; // instance variable
```

declares instance variable `name` of type `String` *outside* the bodies of methods `setName` (lines 10–13) and `getName` (lines 16–19). `String` variables can hold character string values such as `"Jane Green"`. If there are many `Account` objects, each has its own `name`. Because `name` is an instance variable, it can be manipulated by each of the class's methods.

Good Programming Practice 3.1

We prefer to list a class's instance variables first in the class's body, so that you see the names and types of the variables before they're used in the class's methods. You can list the class's instance variables anywhere in the class outside its method declarations, but scattering the instance variables can lead to hard-to-read code.

Access Modifiers `public` *and* `private`

Most instance-variable declarations are preceded with the keyword `private` (as in line 7). Like `public`, `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. So, the variable `name` can be used only in each `Account` object's methods (`setName` and `getName` in this case). You'll soon see that this presents powerful software engineering opportunities.

`setName` *Method of Class* `Account`

Let's walk through the code of `setName`'s method declaration (lines 10–13):

```
public void setName(String name) ── This line is the method header
{
   this.name = name; // store the name
}
```

We refer to the first line of each method declaration (line 10 in this case) as the *method header*. The method's return type (which appears before the method name) specifies the type of data the method returns to its caller after performing its task. The return type `void` (line 10) indicates that `setName` will perform a task but will not return (i.e., give back) any information to its caller. In Chapter 2, you used methods that return information—for example, 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. As you'll soon see, `Account` method `getName` returns a value.

Method `setName` receives parameter `name` of type `String`. Parameters are declared in the parameter list, which is located inside the parentheses that follow the method name in the method header. When there are multiple parameters, each is separated from the next by a comma. Each parameter must specify a type (in this case, `String`) followed by a variable name (in this case, `name`).

Parameters Are Local Variables

In Chapter 2, we declared all of an app's variables in the `main` method. Variables declared in a particular method's body (such as `main`) are local variables which can be used only in that method. Each method can access only its own local variables, not those of other methods. When a method terminates, the values of its local variables are lost. A method's parameters also are local variables of the method.

`setName` Method Body

Every method body is delimited by a pair of braces (as in lines 11 and 13 of Fig. 3.1) containing one or more statements that perform the method's task(s). In this case, the method body contains a single statement (line 12) that assigns the value of the `name` parameter (a `String`) to the class's `name` instance variable, thus storing the account name in the object.

If a method contains a local variable with the same name as an instance variable (as in lines 10 and 7, respectively), that method's body will refer to the local variable rather than the instance variable. In this case, the local variable is said to *shadow* the instance variable in the method's body. The method's body can use the keyword **`this`** to refer to the shadowed instance variable explicitly, as shown on the left side of the assignment in line 12.

Good Programming Practice 3.2

We could have avoided the need for keyword `this` here by choosing a different name for the parameter in line 10, but using the `this` keyword as shown in line 12 is a widely accepted practice to minimize the proliferation of identifier names.

After line 12 executes, the method has completed its task, so it returns to its *caller*. As you'll soon see, the statement in line 21 of `main` (Fig. 3.2) calls method `setName`.

`getName` Method of Class `Account`

Method `getName` (lines 16–19 of Fig. 3.1)

```
public String getName()
{
   return name; // return value of name to caller
}
```

Keyword `return` passes the `String` `name` back to the method's caller

returns a particular `Account` object's `name` to the caller. The method has an empty parameter list, so it does not require additional information to perform its task. The method returns a `String`. When a method that specifies a return type other than `void` is called and completes its task, it must return a result to its caller. A statement that calls method `getName` on an `Account` object (such as the ones in lines 16 and 26 of Fig. 3.2) expects to receive the `Account`'s name—a `String`, as specified in the method declaration's return type.

The return statement in line 18 of Fig. 3.1 passes the `String` value of instance variable `name` back to the caller. For example, when the value is returned to the statement in lines 25–26 of Fig. 3.2, the statement uses that value to output the name.

3.2.2 AccountTest Class That Creates and Uses an Object of Class Account

Next, we'd like to use class `Account` in an app and *call* each of its methods. A class that contains a `main` method begins the execution of a Java app. Class `Account` cannot execute by itself because it does not contain a `main` method—if you type `java Account` in the command window, you'll get an error indicating "`Main method not found in class Account`." To fix this problem, you must either declare a separate class that contains a `main` method or place a `main` method in class `Account`.

Driver Class `AccountTest`

We use a separate class `AccountTest` (Fig. 3.2) containing method `main` to test class `Account`. Once `main` begins executing, it may call other methods in this and other classes; those may, in turn, call other methods, and so on. Class `AccountTest`'s `main` method creates one `Account` object and calls its `getName` and `setName` methods. Such a class is sometimes called a *driver class*—just as a `Person` object drives a `Car` object by telling it what to do (go faster, go slower, turn left, turn right, etc.), class `AccountTest` drives an `Account` object, telling it what to do by calling its methods.

```
1  // Fig. 3.2: AccountTest.java
2  // Creating and manipulating an Account object.
3  import java.util.Scanner;
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        // create a Scanner object to obtain input from the command window
10       Scanner input = new Scanner(System.in);
11
12       // create an Account object and assign it to myAccount
13       Account myAccount = new Account();
14
15       // display initial value of name (null)
16       System.out.printf("Initial name is: %s%n%n", myAccount.getName());
17
18       // prompt for and read name
19       System.out.println("Please enter the name:");
20       String theName = input.nextLine(); // read a line of text
21       myAccount.setName(theName); // put theName in myAccount
22       System.out.println(); // outputs a blank line
23
24       // display the name stored in object myAccount
25       System.out.printf("Name in object myAccount is:%n%s%n",
26          myAccount.getName());
27    }
28 } // end class AccountTest
```

Fig. 3.2 | Creating and manipulating an `Account` object. (Part 1 of 2.)

```
Initial name is: null

Please enter the name:
Jane Green

Name in object myAccount is:
Jane Green
```

Fig. 3.2 | Creating and manipulating an `Account` object. (Part 2 of 2.)

`Scanner` Object for Receiving Input from the User

Line 10 creates a `Scanner` object named `input` for inputting the name from the user. Line 19 prompts the user to enter a name. Line 20 uses the `Scanner` object's **`nextLine`** method to read the name from the user and assign it to the local variable `theName`. You type the name and press *Enter* to submit it to the program. Pressing *Enter* inserts a newline character after the characters you typed. Method `nextLine` reads characters (including white-space characters, such as the blank in `"Jane Green"`) until it encounters the newline, then returns a `String` containing the characters up to, but *not* including, the newline, which is discarded.

Class `Scanner` provides various other input methods, as you'll see throughout the book. A method similar to `nextLine`—named **`next`**—reads the next word. When you press *Enter* after typing some text, 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 subsequent statements that call the `Scanner`'s methods later in the program.

Instantiating an Object—Keyword `new` and Constructors

Line 13 creates an `Account` object and assigns it to variable `myAccount` of type `Account`. Variable `myAccount` is initialized with the result of the **class instance creation expression** `new Account()`. Keyword **`new`** creates a new object of the specified class—in this case, `Account`. The parentheses to the right of `Account` are required. As you'll learn in Section 3.4, those parentheses in combination with a class name represent a call to a **constructor**, which is similar to a method but is called implicitly by the `new` operator to initialize an object's instance variables when the object is created. In Section 3.4, you'll see how to place an argument in the parentheses to specify an initial value for an `Account` object's `name` instance variable—you'll enhance class `Account` to enable this. For now, we simply leave the parentheses empty. Line 10 contains a class instance creation expression for a `Scanner` object—the expression initializes the `Scanner` with `System.in`, which tells the `Scanner` where to read the input from (i.e., the keyboard).

Calling Class `Account`'s `getName` Method

Line 16 displays the initial name, which is obtained by calling the object's `getName` method. Just as we can use object `System.out` to call its methods `print`, `printf` and `println`, we can use object `myAccount` to call its methods `getName` and `setName`. Line 16 calls `getName` using the `myAccount` object created in line 13, followed by a **dot separator** (`.`),

then the method name `getName` and an empty set of parentheses because no arguments are being passed. When `getName` is called:

1. The app transfers program execution from the call (line 16 in `main`) to method `getName`'s declaration (lines 16–19 of Fig. 3.1). Because `getName` was called via the `myAccount` object, `getName` "knows" which object's instance variable to manipulate.
2. Next, method `getName` performs its task—that is, it returns the `name` (line 18 of Fig. 3.1). When the `return` statement executes, program execution continues where `getName` was called (line 16 in Fig. 3.2).
3. `System.out.printf` displays the `String` returned by method `getName`, then the program continues executing at line 19 in `main`.

Error-Prevention Tip 3.1

Never use as a format-control a string that was input from the user. When method `System.out.printf` evaluates the format-control string in its first argument, the method performs tasks based on the conversion specifier(s) in that string. If the format-control string were obtained from the user, a malicious user could supply conversion specifiers that would be executed by `System.out.printf`, possibly causing a security breach.

`null`—the Default Initial Value for `String` Variables

The first line of the output shows the name "`null`." Unlike local variables, which are *not* automatically initialized, every instance variable has a **default initial value**—a value provided by Java when you do not specify the instance variable's initial value. Thus, instance variables 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 an instance variable of type `String` (like `name` in this example) is `null`, which we discuss further in Section 3.3 when we consider reference types.

Calling Class `Account`'s `setName` Method

Line 21 calls `myAccounts`'s `setName` method. A method call can supply arguments whose values are assigned to the corresponding method parameters. In this case, the value of `main`'s local variable `theName` in parentheses is the argument that's passed to `setName` so that the method can perform its task. When `setName` is called:

1. The app transfers program execution from line 21 in `main` to `setName` method's declaration (lines 10–13 of Fig. 3.1), and the argument value in the call's parentheses (`theName`) is assigned to the corresponding parameter (`name`) in the method header (line 10 of Fig. 3.1). Because `setName` was called via the `myAccount` object, `setName` "knows" which object's instance variable to manipulate.
2. Next, method `setName` performs its task—that is, it assigns the `name` parameter's value to instance variable `name` (line 12 of Fig. 3.1).
3. When program execution reaches `setName`'s closing right brace, it returns to where `setName` was called (line 21 of Fig. 3.2), then continues at line 22.

The number of arguments in a method call must match the number of parameters in the method declaration's parameter list. 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 see in Chapter 6, an argument's type and its corresponding parameter's type are

not required to be identical.) In our example, the method call passes one argument of type `String` (`theName`)—and the method declaration specifies one parameter of type `String` (`name`, declared in line 10 of Fig. 3.1). So in this example, the type of the argument in the method call exactly matches the type of the parameter in the method header.

Displaying the Name That Was Entered by the User

Line 22 of Fig. 3.2 outputs a blank line. When the second call to method `getName` (line 26) executes, the name entered by the user in line 20 is displayed. When the statement at lines 25–26 completes execution, the end of method `main` is reached, so the program terminates.

3.2.3 Compiling and Executing an App with Multiple Classes

You must compile the classes in Figs. 3.1 and 3.2 before you can execute the app. This is the first time you've created an app with multiple classes. Class `AccountTest` has a `main` method; class `Account` does not. To compile this app, first change to the directory that contains the app's source-code files. Next, type the command

```
javac Account.java AccountTest.java
```

to compile both classes at once. If the directory containing the app includes *only* this app's files, you can compile both classes with the command

```
javac *.java
```

The asterisk (`*`) in `*.java` indicates that all files in the current directory ending with the filename extension "`.java`" should be compiled. If both classes compile correctly—that is, no compilation errors are displayed—you can then run the app with the command

```
java AccountTest
```

3.2.4 Account UML Class Diagram with an Instance Variable and *set* and *get* Methods

We'll often use UML class diagrams to summarize a class's attributes and operations. In industry, UML diagrams help systems designers specify a system in a concise, graphical, programming-language-independent manner, before programmers implement the system in a specific programming language. Figure 3.3 presents a **UML class diagram** for class `Account` of Fig. 3.1.

Top Compartment

In the UML, each class is modeled in a class diagram as a rectangle with three compartments. In this diagram the top compartment contains the class name `Account` centered horizontally in boldface type.

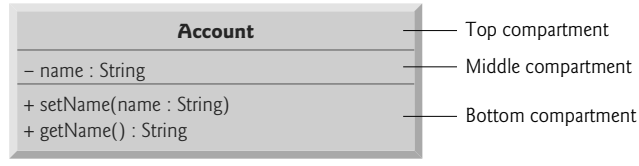

Fig. 3.3 | UML class diagram for class `Account` of Fig. 3.1.

Middle Compartment
The middle compartment contains the class's attribute `name`, which corresponds to the instance variable of the same name in Java. Instance variable `name` is `private` in Java, so the UML class diagram lists a minus sign (–) access modifier before the attribute name. Following the attribute name are a colon and the attribute type, in this case `String`.

Bottom Compartment
The bottom compartment contains the class's **operations**, `setName` and `getName`, which correspond to the methods of the same names in Java. The UML models operations by listing the operation name preceded by an access modifier, in this case + `getName`. This plus sign (+) indicates that `getName` is a public operation in the UML (because it's a `public` method in Java). Operation `getName` does not have any parameters, so the parentheses following the operation name in the class diagram are empty, just as they are in the method's declaration in line 16 of Fig. 3.1. Operation `setName`, also a public operation, has a `String` parameter called `name`.

Return Types
The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name. `Account` method `getName` (Fig. 3.1) has a `String` return type. Method `setName` does not return a value (because it returns `void` in Java), so the UML class diagram does not specify a return type after the parentheses of this operation.

Parameters
The UML models a parameter a bit differently from Java by listing the parameter name, followed by a colon and the parameter type in the parentheses after the operation name. The UML has its own data types similar to those of Java, but for simplicity, we'll use the Java data types. `Account` method `setName` (Fig. 3.1) has a `String` parameter named `name`, so Fig. 3.3 lists `name : String` between the parentheses following the method name.

3.2.5 Additional Notes on Class `AccountTest`

`static` Method `main`
In Chapter 2, each class we declared had one method named `main`. Recall that `main` is a special method that's always called automatically by the Java Virtual Machine (JVM) when you execute an app. You must call most other methods explicitly to tell them to perform their tasks.

Lines 7–27 of Fig. 3.2 declare method `main`. A key part of enabling the JVM to locate and call method `main` to begin the app'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*—in this case class `AccountTest`. We discuss `static` methods in detail in Chapter 6.

Notes on `import` Declarations
Notice the `import` declaration in Fig. 3.2 (line 3), which indicates to the compiler that the program uses class `Scanner`. As mentioned in Chapter 2, 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, like classes `Account` and `AccountTest`. 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 that package. Thus, an `import` declaration is not required when one class in a package uses another in the same package—such as when class `AccountTest` uses class `Account`.

The `import` declaration in line 3 is *not* required if we refer to class `Scanner` throughout this file 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 10 of Fig. 3.2 also could be written as

```
java.util.Scanner input = new java.util.Scanner(System.in);
```

Software Engineering Observation 3.1

The Java compiler does not require `import` *declarations in a Java source-code file if the fully qualified class name is specified every time a class name is used. Most Java programmers prefer the more concise programming style enabled by* `import` *declarations.*

3.2.6 Software Engineering with `private` Instance Variables and `public` *set* and *get* Methods

As you'll see, through the use of *set* and *get* methods, you can *validate* attempted modifications to `private` data and control how that data is presented to the caller—these are compelling software engineering benefits. We'll discuss this in more detail in Section 3.5.

If the instance variable were `public`, any **client** of the class—that is, any other class that calls the class's methods—could see the data and do whatever it wanted with it, including setting it to an invalid value.

You might think that even though a client of the class cannot directly access a `private` instance variable, the client can do whatever it wants with the variable through `public` *set* and *get* methods. You would think that you could peek at the `private` data any time with the `public` *get* method and that you could modify the `private` data at will through the `public` *set* method. But *set* methods can be programmed to validate their arguments and reject any attempts to *set* the data to bad values, such as a negative body temperature, a day in March out of the range 1 through 31, a product code not in the company's product catalog, etc. And a *get* method can present the data in a different form. For example, a `Grade` class might store a grade as an `int` between 0 and 100, but a `getGrade` method might return a letter grade as a `String`, such as `"A"` for grades between 90 and 100, `"B"` for grades between 80 and 89, etc. Tightly controlling the access to and presentation of `private` data can greatly reduce errors, while increasing the robustness and security of your programs.

Declaring instance variables with access modifier `private` is known as *data hiding* or *information hiding*. When a program creates (instantiates) an object of class `Account`, variable `name` is *encapsulated* (hidden) in the object and can be accessed only by methods of the object's class.

Software Engineering Observation 3.2

Precede each instance variable and method declaration with an access modifier. Generally, instance variables should be declared `private` *and methods* `public`*. Later in the book, we'll discuss why you might want to declare a method* `private`*.*

Conceptual View of an **`Account`** *Object with Encapsulated Data*

You can think of an `Account` object as shown in Fig. 3.4. The `private` instance variable `name` is *hidden* inside the object (represented by the inner circle containing `name`) and *protected* by an outer layer of `public` methods (represented by the outer circle containing `getName` and `setName`). Any client code that needs to interact with the `Account` object can do so *only* by calling the `public` methods of the protective outer layer.

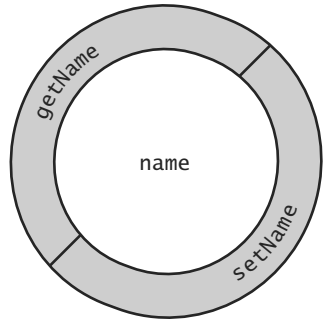

Fig. 3.4 | Conceptual view of an `Account` object with its encapsulated `private` instance variable `name` and protective layer of `public` methods.

3.3 Primitive Types vs. Reference Types

Java's types are divided into primitive types and **reference types**. In Chapter 2, you worked with variables of type `int`—one of the primitive types. The other primitive types are `boolean`, `byte`, `char`, `short`, `long`, `float` and `double`, each of which we discuss in this book—these are summarized in Appendix D. All nonprimitive types are reference types, so classes, which specify the types of objects, are reference types.

A primitive-type variable can hold exactly *one* value of its declared type at a time. For example, an `int` variable can store one integer at a time. When another value is assigned to that variable, the new value replaces the previous one—which is *lost*.

Recall that local variables are *not* initialized by default. Primitive-type instance variables *are* initialized by default—instance 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;
```

Programs use variables of reference types (normally called **references**) to store the *addresses* 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 10 of Fig. 3.2:

```
Scanner input = new Scanner(System.in);
```

creates an object of class `Scanner`, then assigns to the variable `input` a *reference* to that `Scanner` object. Line 13 of Fig. 3.2:

```
Account myAccount = new Account();
```

creates an object of class `Account`, then assigns to the variable `myAccount` a *reference* to that `Account` object. *Reference-type instance variables, if not explicitly initialized, are initialized by default to the value* `null`—which represents a "reference to nothing." That's why the first call to `getName` in line 16 of Fig. 3.2 returns `null`—the value of `name` has *not* yet been set, so the *default initial value* `null` is returned.

To call methods on an object, you need a reference to the object. In Fig. 3.2, the statements in method `main` use the variable `myAccount` to call methods `getName` (lines 16 and 26) and `setName` (line 21) to interact with the `Account` object. Primitive-type variables do *not* refer to objects, so such variables *cannot* be used to call methods.

3.4 Account Class: Initializing Objects with Constructors

As mentioned in Section 3.2, when an object of class `Account` (Fig. 3.1) is created, its `String` instance variable `name` is initialized to `null` by *default*. But what if you want to provide a name when you *create* an `Account` object?

Each class you declare can optionally provide a *constructor* with parameters that can be used to initialize an object of a class when the object is created. Java *requires* a constructor call for *every* object that's created, so this is the ideal point to initialize an object's instance variables. The next example enhances class `Account` (Fig. 3.5) with a constructor that can receive a name and use it to initialize instance variable `name` when an `Account` object is created (Fig. 3.6).

3.4.1 Declaring an Account Constructor for Custom Object Initialization

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 name for an `Account` object when the object is created, as in line 10 of Fig. 3.6:

```
Account account1 = new Account("Jane Green");
```

In this case, the `String` argument `"Jane Green"` is passed to the `Account` object's constructor and used to initialize the `name` instance variable. The preceding statement requires that the class provide a constructor that takes only a `String` parameter. Figure 3.5 contains a modified `Account` class with such a constructor.

```
1  // Fig. 3.5: Account.java
2  // Account class with a constructor that initializes the name.
3
```

Fig. 3.5 | `Account` class with a constructor that initializes the `name`. (Part 1 of 2.)

```
4  public class Account
5  {
6     private String name; // instance variable
7
8     // constructor initializes name with parameter name
9     public Account(String name) // constructor name is class name
10    {
11       this.name = name;
12    }
13
14    // method to set the name
15    public void setName(String name)
16    {
17       this.name = name;
18    }
19
20    // method to retrieve the name
21    public String getName()
22    {
23       return name;
24    }
25 } // end class Account
```

Fig. 3.5 | `Account` class with a constructor that initializes the `name`. (Part 2 of 2.)

`Account` Constructor Declaration

Lines 9–12 of Fig. 3.5 declare `Account`'s constructor. A constructor *must* have the *same name* as the class. A constructor's *parameter list* specifies that the constructor requires one or more pieces of data to perform its task. Line 9 indicates that the constructor has a `String` parameter called `name`. When you create a new `Account` object (as you'll see in Fig. 3.6), you'll pass a person's name to the constructor, which will receive that name in the parameter `name`. The constructor will then assign `name` to *instance variable* `name` in line 11.

Error-Prevention Tip 3.2

Even though it's possible to do so, do not call methods from constructors. We'll explain this in Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces.

Parameter `name` of Class `Account`'s Constructor and Method `setName`

Recall from Section 3.2.1 that method parameters are local variables. In Fig. 3.5, the constructor and method `setName` both have a parameter called `name`. Although these parameters have the same identifier (`name`), the parameter in line 9 is a local variable of the constructor that's *not* visible to method `setName`, and the one in line 15 is a local variable of `setName` that's *not* visible to the constructor.

3.4.2 Class `AccountTest`: Initializing `Account` Objects When They're Created

The `AccountTest` program (Fig. 3.6) initializes two `Account` objects using the constructor. Line 10 creates and initializes the `Account` object `account1`. Keyword `new` requests memory from the system to store the `Account` object, then implicitly calls the class's con-

structor to *initialize* the object. The call is indicated by the parentheses after the class name, which contain the *argument* `"Jane Green"` that's used to initialize the new object's name. The class instance creation expression in line 10 returns a *reference* to the new object, which is assigned to the variable `account1`. Line 11 repeats this process, passing the argument `"John Blue"` to initialize the name for `account2`. Lines 14–15 use each object's `getName` method to obtain the names and show that they were indeed initialized when the objects were *created*. The output shows *different* names, confirming that each `Account` maintains its *own copy* of instance variable `name`.

```
1  // Fig. 3.6: AccountTest.java
2  // Using the Account constructor to initialize the name instance
3  // variable at the time each Account object is created.
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        // create two Account objects
10       Account account1 = new Account("Jane Green");
11       Account account2 = new Account("John Blue");
12
13       // display initial value of name for each Account
14       System.out.printf("account1 name is: %s%n", account1.getName());
15       System.out.printf("account2 name is: %s%n", account2.getName());
16    }
17 } // end class AccountTest
```

```
account1 name is: Jane Green
account2 name is: John Blue
```

Fig. 3.6 | Using the `Account` constructor to initialize the `name` instance variable at the time each `Account` object is created.

Constructors Cannot Return Values

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`—later in the book we'll explain when to use `private` constructors.

Default Constructor

Recall that line 13 of Fig. 3.2

```
Account myAccount = new Account();
```

used `new` to create an `Account` object. The *empty* parentheses after "`new Account`" indicate a call to the class's **default constructor**—in any class that does *not* explicitly declare a constructor, the compiler provides a default constructor (which always has no parameters). When a class has only the default constructor, the class's instance variables are initialized to their *default values*. In Section 8.5, you'll learn that classes can have multiple constructors.

There's No Default Constructor in a Class That Declares a Constructor
If you declare a constructor for a class, the compiler will *not* create a *default constructor* for that class. In that case, you will not be able to create an `Account` object with the class instance creation expression `new Account()` as we did in Fig. 3.2—unless the custom constructor you declare takes *no* parameters.

Software Engineering Observation 3.3
Unless default initialization of your class's instance variables is acceptable, provide a custom constructor to ensure that your instance variables are properly initialized with meaningful values when each new object of your class is created.

Adding the Constructor to Class **`Account`***'s UML Class Diagram*
The UML class diagram of Fig. 3.7 models class `Account` of Fig. 3.5, which has a constructor with a `String name` parameter. Like operations, the UML models constructors in the *third* compartment of a class diagram. To distinguish a constructor from the class's operations, the UML requires that the word "constructor" be enclosed in **guillemets** (« and ») and placed before the constructor's name. It's customary to list constructors *before* other operations in the third compartment.

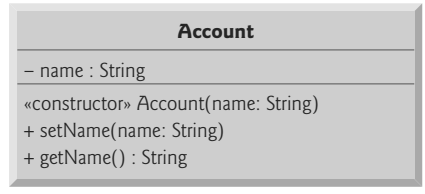

Fig. 3.7 | UML class diagram for `Account` class of Fig. 3.5.

3.5 Account Class with a Balance; Floating-Point Numbers

We now declare an `Account` class that maintains the *balance* of a bank account in addition to the name. Most account balances are not integers. So, class `Account` represents the account balance as a **floating-point number**—a number with a *decimal point*, such as 43.95, 0.0, –129.8873. [In Chapter 8, we'll begin representing monetary amounts precisely with class `BigDecimal` as you should do when writing industrial-strength monetary applications.]

Java provides two primitive types for storing floating-point numbers in memory—`float` and `double`. Variables of type **`float`** represent **single-precision floating-point numbers** and can hold 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 can hold up to *15 significant digits*—about *double* the precision of `float` variables.

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 D, Primitive Types, for the precise ranges of values for `float`s and `double`s.

3.5.1 Account Class with a balance Instance Variable of Type double

Our next app contains a version of class `Account` (Fig. 3.8) that maintains as instance variables the `name` *and* the `balance` of a bank account. A typical bank services *many* accounts, each with its *own* balance, so line 8 declares an instance variable `balance` of type `double`. Every instance (i.e., object) of class `Account` contains its *own* copies of *both* the `name` and the `balance`.

```
1  // Fig. 3.8: Account.java
2  // Account class with a double instance variable balance and a constructor
3  // and deposit method that perform validation.
4
5  public class Account
6  {
7     private String name; // instance variable
8     private double balance; // instance variable
9
10    // Account constructor that receives two parameters
11    public Account(String name, double balance)
12    {
13       this.name = name; // assign name to instance variable name
14
15       // validate that the balance is greater than 0.0; if it's not,
16       // instance variable balance keeps its default initial value of 0.0
17       if (balance > 0.0) // if the balance is valid
18          this.balance = balance; // assign it to instance variable balance
19    }
20
21    // method that deposits (adds) only a valid amount to the balance
22    public void deposit(double depositAmount)
23    {
24       if (depositAmount > 0.0) // if the depositAmount is valid
25          balance = balance + depositAmount; // add it to the balance
26    }
27
28    // method returns the account balance
29    public double getBalance()
30    {
31       return balance;
32    }
33
34    // method that sets the name
35    public void setName(String name)
36    {
37       this.name = name;
38    }
39
40    // method that returns the name
41    public String getName()
42    {
```

Fig. 3.8 | `Account` class with a `double` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 1 of 2.)

```
43          return name; // give value of name back to caller
44       } // end method getName
45    } // end class Account
```

Fig. 3.8 | `Account` class with a `double` instance variable `balance` and a constructor and `deposit` method that perform validation. (Part 2 of 2.)

`Account` Class Two-Parameter Constructor

The class has a *constructor* and four *methods*. It's common for someone opening an account to deposit money immediately, so the constructor (lines 11–19) now receives a second parameter—`initialBalance` of type `double` that represents the *starting balance*. Lines 17–18 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*.

`Account` Class `deposit` Method

Method `deposit` (lines 22–26) does *not* return any data when it completes its task, so its return type is `void`. The method receives one parameter named `depositAmount`—a `double` value that's *added* to the `balance` *only* if the parameter value is *valid* (i.e., greater than zero). Line 25 first adds the current `balance` and `depositAmount`, forming a *temporary* sum which is *then* assigned to `balance`, *replacing* its prior value (recall that addition has a *higher* precedence than assignment). It's important to understand that the calculation on the right side of the assignment operator in line 25 does *not* modify the balance—that's why the assignment is necessary.

`Account` Class `getBalance` Method

Method `getBalance` (lines 29–32) allows *clients* of the class (i.e., other classes whose methods call the methods of this class) to obtain the value of a particular `Account` object's `balance`. The method specifies return type `double` and an *empty* parameter list.

`Account`'s Methods Can All Use `balance`

Once again, the statements in lines 18, 25 and 31 use the 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.

3.5.2 `AccountTest` Class to Use Class `Account`

Class `AccountTest` (Fig. 3.9) creates two `Account` objects (lines 9–10) and initializes them with a *valid* balance of `50.00` and an *invalid* balance of `-7.53`, respectively—for the purpose of our examples, we assume that balances must be greater than or equal to zero. The calls to method `System.out.printf` in lines 13–16 output the account names and balances, which are obtained by calling each `Account`'s `getName` and `getBalance` methods.

```
1  // Fig. 3.9: AccountTest.java
2  // Inputting and outputting floating-point numbers with Account objects.
3  import java.util.Scanner;
```

Fig. 3.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 1 of 3.)

```
4
5  public class AccountTest
6  {
7     public static void main(String[] args)
8     {
9        Account account1 = new Account("Jane Green", 50.00);
10       Account account2 = new Account("John Blue", -7.53);
11
12       // display initial balance of each object
13       System.out.printf("%s balance: $%.2f%n",
14          account1.getName(), account1.getBalance());
15       System.out.printf("%s balance: $%.2f%n%n",
16          account2.getName(), account2.getBalance());
17
18       // create a Scanner to obtain input from the command window
19       Scanner input = new Scanner(System.in);
20
21       System.out.print("Enter deposit amount for account1: "); // prompt
22       double depositAmount = input.nextDouble(); // obtain user input
23       System.out.printf("%nadding %.2f to account1 balance%n%n",
24          depositAmount);
25       account1.deposit(depositAmount); // add to account1's balance
26
27       // display balances
28       System.out.printf("%s balance: $%.2f%n",
29          account1.getName(), account1.getBalance());
30       System.out.printf("%s balance: $%.2f%n%n",
31          account2.getName(), account2.getBalance());
32
33       System.out.print("Enter deposit amount for account2: "); // prompt
34       depositAmount = input.nextDouble(); // obtain user input
35       System.out.printf("%nadding %.2f to account2 balance%n%n",
36          depositAmount);
37       account2.deposit(depositAmount); // add to account2 balance
38
39       // display balances
40       System.out.printf("%s balance: $%.2f%n",
41          account1.getName(), account1.getBalance());
42       System.out.printf("%s balance: $%.2f%n%n",
43          account2.getName(), account2.getBalance());
44    } // end main
45 } // end class AccountTest
```

```
Jane Green balance: $50.00
John Blue balance: $0.00

Enter deposit amount for account1: 25.53

adding 25.53 to account1 balance

Jane Green balance: $75.53
John Blue balance: $0.00
```

Fig. 3.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 2 of 3.)

```
Enter deposit amount for account2: 123.45

adding 123.45 to account2 balance

Jane Green balance: $75.53
John Blue balance: $123.45
```

Fig. 3.9 | Inputting and outputting floating-point numbers with `Account` objects. (Part 3 of 3.)

Displaying the **`Account`** *Objects' Initial Balances*
When method `getBalance` is called for `account1` from line 14, the value of `account1`'s `balance` is returned from line 31 of Fig. 3.8 and displayed by the `System.out.printf` statement (Fig. 3.9, lines 13–14). Similarly, when method `getBalance` is called for `account2` from line 16, the value of the `account2`'s `balance` is returned from line 31 of Fig. 3.8 and displayed by the `System.out.printf` statement (Fig. 3.9, lines 15–16). The `balance` of `account2` is initially `0.00`, because the constructor rejected the attempt to start `account2` with a *negative* balance, so the balance retains its default initial value.

Formatting Floating-Point Numbers for Display
Each of the `balance`s is output by `printf` with the format specifier `%.2f`. The **`%f` format specifier** 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 and 27.33379 would be rounded to 27.33.

Reading a Floating-Point Value from the User and Making a Deposit
Line 21 (Fig. 3.9) prompts the user to enter a deposit amount for `account1`. Line 22 declares local variable `depositAmount` to store each deposit amount entered by the user. Unlike instance variables (such as `name` and `balance` in class `Account`), local variables (like `depositAmount` in `main`) are not initialized by default, so they normally must be initialized explicitly. As you'll learn momentarily, variable `depositAmount`'s initial value will be determined by the user's input.

Common Programming Error 3.1

The Java compiler will issue a compilation error if you attempt to use the value of an uninitialized local variable. This helps you avoid dangerous execution-time logic errors. It's always better to get the errors out of your programs at compilation time rather than execution time.

Line 22 obtains the input from the user by calling `Scanner` object `input`'s **`nextDouble`** method, which returns a `double` value entered by the user. Lines 23–24 display the `depositAmount`. Line 25 calls object `account1`'s `deposit` method with the `depositAmount` as the method's argument. When the method is called, the argument's value is assigned to the parameter `depositAmount` of method `deposit` (line 22 of Fig. 3.8); then method `deposit` adds that value to the `balance`. Lines 28–31 (Fig. 3.9) output the `name`s and `balance`s of both `Account`s again to show that *only* `account1`'s `balance` has changed.

Line 33 prompts the user to enter a deposit amount for `account2`. Line 34 obtains the input from the user by calling `Scanner` object `input`'s `nextDouble` method. Lines 35–36 display the `depositAmount`. Line 37 calls object `account2`'s `deposit` method with `depositAmount` as the method's argument; then method `deposit` adds that value to the `balance`. Finally, lines 40–43 output the `name`s and `balance`s of both `Account`s again to show that only `account2`'s `balance` has changed.

UML Class Diagram for Class `Account`

The UML class diagram in Fig. 3.10 concisely models class `Account` of Fig. 3.8. The diagram models in its second compartment the `private` attributes `name` of type `String` and `balance` of type `double`.

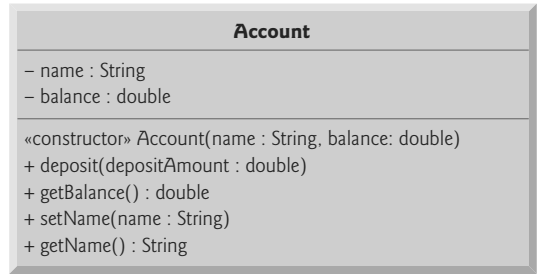

Fig. 3.10 | UML class diagram for `Account` class of Fig. 3.8.

Class `Account`'s constructor is modeled in the third compartment with parameters `name` of type `String` and `initialBalance` of type `double`. The class's four `public` methods also are modeled in the *third* compartment—operation `deposit` with a `depositAmount` parameter of type `double`, operation `getBalance` with a return type of `double`, operation `setName` with a `name` parameter of type `String` and operation `getName` with a return type of `String`.

3.6 Wrap-Up

In this chapter, you learned how to create your own Java classes and methods, create objects of those classes and call methods of those objects to perform useful actions. You declared instance variables of a class to maintain data for each object of the class, and you declared your own methods to operate on that data. You called methods and passed information to them as arguments whose values are assigned to the method's parameters. 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. You saw how to create UML class diagrams that model the methods, attributes and constructors of classes. Finally, you learned about floating-point numbers (numbers with decimal points)—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 Chapter 8, we'll begin representing monetary amounts precisely with class `BigDecimal`.] In the next chapter we discuss control statements, which specify the order in which a program's actions are performed.

This page intentionally left blank

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

M

N

P

Q

R

S

T

U

V

W

X

Y

Z

This page intentionally left blank