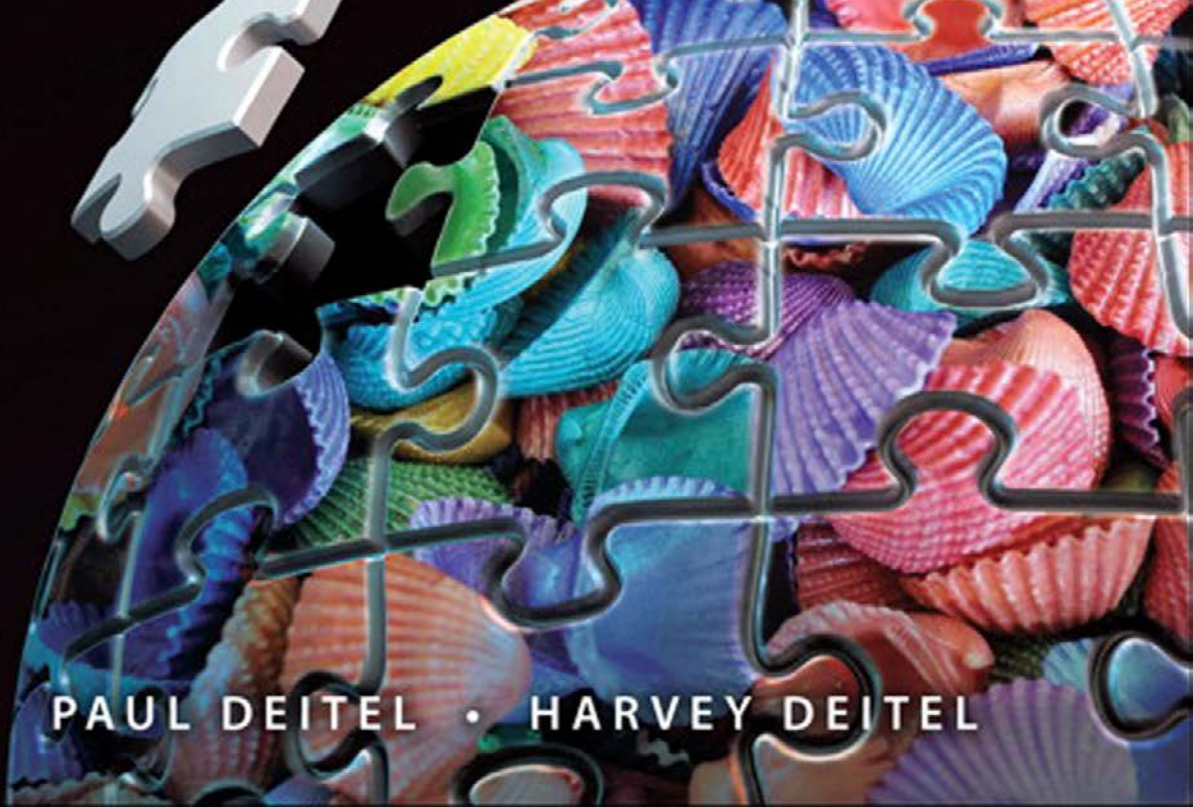
DEITEL® DEVELOPER SERIES

Java® 9 for Programmers

Interactive Java *with* **JShell**

Java Platform Module System (Project Jigsaw)



PAUL DEITEL • HARVEY DEITEL

DEITEL® DEVELOPER SERIES

The DEITEL® DEVELOPER SERIES is designed for professional programmers. The series presents focused treatments on emerging and mature technologies, including Java®, C++, C, C# and .NET, JavaScript®, Internet and web development, Android™ app development, iOS® app development, Swift™ and more. Each book in the series contains the same live-code teaching methodology used in the Deitels' HOW TO PROGRAM SERIES college textbooks—in this book, most concepts are presented in the context of completely coded, live apps.

ABOUT THE COVER

The cover art we selected reflects some key themes of ***Java® 9 for Programmers***: The art includes a colorful collection of sea shells—one of Java 9's key features is **JShell**, which is **Java 9's REPL (read-eval-print-loop) for interactive Java**. JShell is one of the most important pedagogic and productivity enhancements to Java since it was announced in 1995. We discuss JShell in detail in Chapter 23 and show how you can use it for discovery and experimentation, rapid prototyping of code segments and to learn Java faster. The shell art is overlayed onto a sphere of jigsaw puzzle pieces representing the earth. The most important new software-engineering capability in Java 9 is the **Java Platform Module System** (Chapter 27)—which was developed by Project Jigsaw.

Cover graphic: Jigsaw Puzzles, © StacieStauffSmith Photos/ShutterStock
Sea Shells, © Mopic/ShutterStock

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, Internet and web software technology, and Android and iOS app development. The company's clients over the years have included many of the world's largest corporations, 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. Through its 42-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., creates leading-edge programming professional books, college textbooks, LiveLessons video products, Learning Paths in the Safari service (`http://www.safaribooksonline.com`), e-books and REVEL™ interactive multimedia courses with integrated labs and assessment (`revel.pearson.com`). To learn more about Deitel & Associates, Inc., its text and video publications and its worldwide instructor-led, on-site training curriculum, visit `www.deitel.com` or send an email to `deitel@deitel.com`. Join the Deitel social networking communities on LinkedIn® (`bit.ly/DeitelLinkedIn`), Facebook® (`www.facebook.com/DeitelFan`), Twitter® (`twitter.com/deitel`), and YouTube™ (`www.youtube.com/DeitelTV`), and subscribe to the ***Deitel® Buzz Online*** newsletter (`www.deitel.com/newsletter/subscribe.html`).

JAVA® 9
FOR PROGRAMMERS
FOURTH EDITION
DEITEL® DEVELOPER SERIES

Deitel® Series

Deitel® Developer Series

Android™ 6 for Programmers: An App-Driven Approach, 3/E
C for Programmers with an Introduction to C11
C++11 for Programmers
C# 6 for Programmers
Java® 9 for Programmers
JavaScript for Programmers
Swift™ for Programmers

How to Program Series

Android™ How to Program, 3/E
C++ How to Program, 10/E
C How to Program, 8/E
Java® How to Program, Early Objects Version, 11/E
Java® How to Program, Late Objects Version, 11/E
Internet & World Wide Web How to Program, 5/E
Visual Basic® 2012 How to Program, 6/E
Visual C#® How to Program, 6/E

Simply Series

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

VitalSource Web Books

`http://bit.ly/DeitelOnVitalSource`

Android™ How to Program, 2/E and 3/E
C++ How to Program, 9/E and 10/E
Java® How to Program, 10/E and 11/E
Simply C++: An App-Driven Tutorial Approach
Simply Visual Basic® 2010: An App-Driven Approach, 4/E
Visual Basic® 2012 How to Program, 6/E
Visual C#® How to Program, 6/E
Visual C#® 2012 How to Program, 5/E

LiveLessons Video Learning Products

`http://deitel.com/books/LiveLessons/`

Android™ 6 App Development Fundamentals, 3/E
C++ Fundamentals
Java SE 8® Fundamentals, 2/E
Java SE 9® Fundamentals, 3/E
C# 6 Fundamentals
C# 2012 Fundamentals
JavaScript Fundamentals
Swift™ Fundamentals

REVEL™ Interactive Multimedia

REVEL™ for Deitel Java™

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

- Facebook®—`http://facebook.com/DeitelFan`
- Twitter®—`http://twitter.com/deitel`
- LinkedIn®—`http://linkedin.com/company/deitel-&-associates`
- YouTube™—`http://youtube.com/DeitelTV`

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

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

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

`deitel@deitel.com`

For information on programming-languages corporate training seminars offered by Deitel & Associates, Inc. worldwide, write to `deitel@deitel.com` or visit:

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

For continuing updates on Pearson/Deitel publications visit:

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

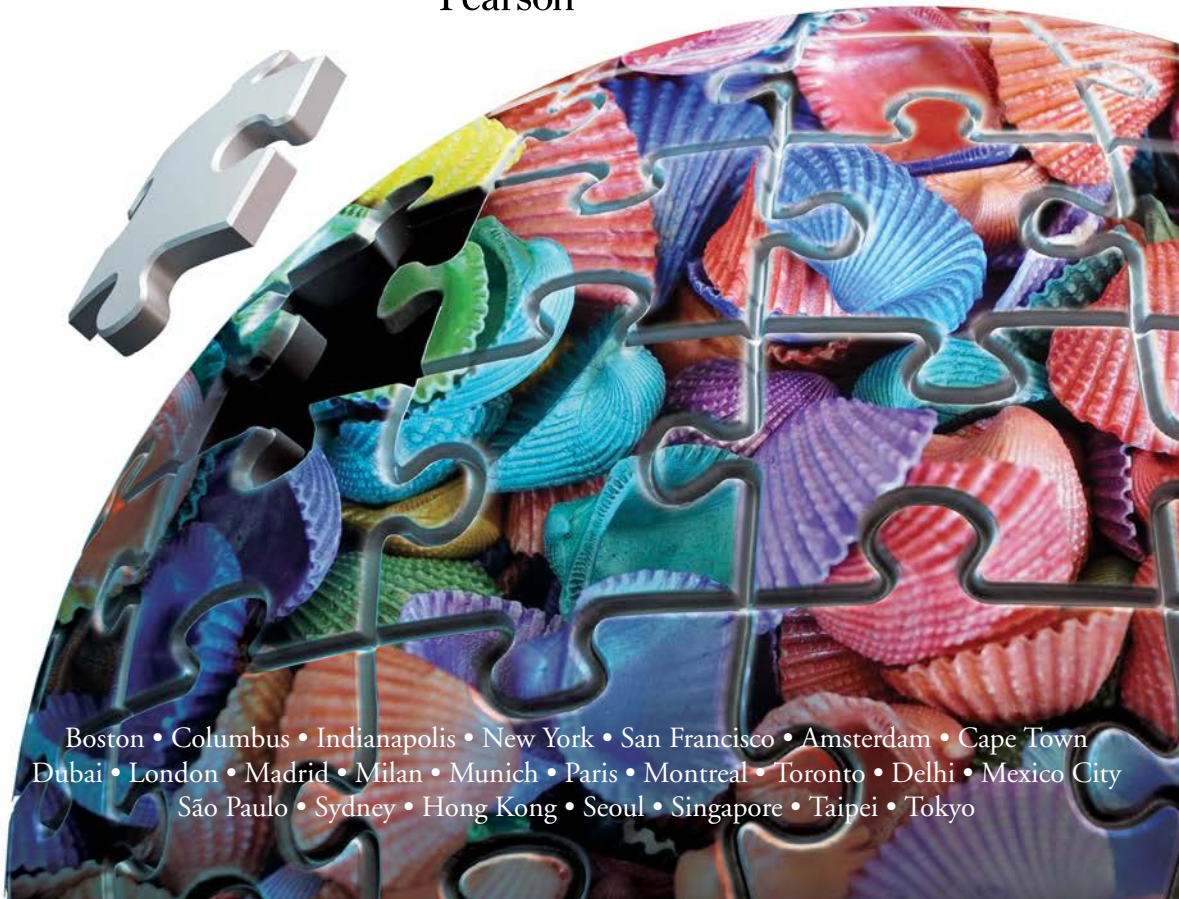
Visit the Deitel Resource Centers, which will help you master programming languages, software development, Android™ and iOS® app development, and Internet- and web-related topics:

`http://www.deitel.com/ResourceCenters.html`

Java® 9 for Programmers
Fourth Edition
Deitel® Developer Series

Paul Deitel • Harvey Deitel
Deitel & Associates, Inc.

Pearson



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

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

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

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the web: informit.com

Library of Congress Control Number: 2017937968

ISBN-13: 978-0-13-477756-6
ISBN-10: 0-13-477756-5

2 20

In memory of Dr. Henry Heimlich:
Barbara Deitel used your Heimlich maneuver to save Abbey Deitel's life. Our family is forever grateful to you.

Harvey, Barbara, Paul and Abbey 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 and Android 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

Throughout my career I've met and interviewed many expert Java developers who've learned from Paul and Harvey, through one or more of their professional books, college textbooks, videos and corporate training. Many Java User Groups have joined together around the Deitels' publications, which are used internationally in professional training programs and university courses. You are joining an elite group.

How do I become an expert Java developer?

This is one of the most common questions I receive at events with Java professionals and talks for university students. Programmers want to become expert developers—and this is a great time to be one. The market is wide open, full of opportunities and fascinating projects.

So, how do you do it? First, let's be clear: Software development is hard, but it opens the door to great opportunities. Accept that it's hard, embrace the complexity, enjoy the ride. There are no limits to how much you can expand your skills. The success or failure of initiatives everywhere depends on expert developers' knowledge and skills.

The push for you to get expert-level skills is what makes *Java® 9 for Programmers* so compelling. It's written by authors who are educators and developers, with nine decades of computing experience between them and with input over the years from some of the world's leading professional Java experts—Java Champions, open-source Java developers, even creators of Java itself. Their collective knowledge and experience will guide you. Even seasoned Java professionals will learn and grow their expertise with the wisdom in these pages.

Java was released in 1995—Paul and Harvey had the first edition of their college textbook *Java How to Program* ready for Fall 1996 classes. Since that groundbreaking book, they've produced ten more editions and many editions of their *Java® for Programmers* professional books, keeping current with the latest developments and idioms in the Java software-engineering community. You hold in your hands the map that will enable you to rapidly develop your Java skills.

The Deitels have broken down the humongous Java world into well-defined, specific goals. With both Java 8 and Java 9 in the same book, you'll have up-to-date skills on the latest Java technologies. I'm impressed with the care that the Deitels always take to accommodate readers at all levels. They ease you into difficult concepts and deal with the challenges that professionals encounter in industry projects. And if you have a question, don't be shy—the Deitels publish their e-mail address—`deitel@deitel.com`—in every book they write to encourage interaction.

One of my favorite chapters is Lambdas and Streams. It covers the topic in detail and the examples shine—many real-world challenges that will help you sharpen your skills.

I love the chapter about JShell—the new Java 9 tool that enables interactive Java. JShell allows you to explore, discover and experiment with new concepts, language features and APIs, make mistakes—accidentally and intentionally—and correct them, and rapidly prototype new code. It will prove to be an important tool for leveraging your learning and productivity. Paul and Harvey give a full treatment of JShell that developers at all levels will be able to put to use immediately.

Perhaps most important: Check out the chapter on the Java Platform Module System—the single most important new software-engineering capability introduced in Java since its inception. Developers building new big systems—and especially developers migrating existing systems to Java 9—will appreciate the Deitel's detailed walkthrough of modularity.

There's lots of additional information about Java 9, the important new Java release. You can jump right in and learn the latest Java features. If you're still working with Java 8, you can ease into Java 9 at your own pace.

Also check out the amazing coverage of JavaFX—Java's latest GUI, graphics and multimedia capabilities. JavaFX is the recommended toolkit for new projects. And be sure to dig in on Paul and Harvey's treatment of concurrency. They explain the basic concepts so clearly that the intermediate and advanced examples and discussions will be easy to master. You will be ready to maximize your applications' performance in an increasingly multicore world.

Enjoy the book—I wish you great success!

Bruno Sousa
bruno@javaman.com.br
Java Champion
Java Specialist at ToolsCloud
President of SouJava (the Brazilian Java Society)
SouJava representative at the Java Community Process

Preface

Welcome to the Java® programming language and *Java® 9 for Programmers*! This book presents leading-edge computing technologies for software developers. It also will help you prepare for most topics covered by the following Oracle Java Standard Edition 8 (Java SE 8) Certifications (and the Java SE 9 editions, when they appear):

- Oracle Certified Associate, Java SE 8 Programmer
- Oracle Certified Professional, Java SE 8 Programmer

If you haven't already done so, please read the back cover and the additional reviewer comments inside the back cover—these concisely capture the essence of the book. In this Preface we provide more details.

We focus on software engineering best practices. At the heart of the book is the Deitel signature **live-code approach**—we present most concepts in the context of hundreds of complete working programs that have been tested on **Windows®**, **macOS®** and **Linux®**. The code examples are accompanied by live sample executions. All the source code for the book's code examples is available at:

```
http://www.deitel.com/books/Java9FP
```

New and Updated Features

In the following sections, we discuss the key features and updates we've made for *Java® 9 for Programmers*, including:

- Coverage of Java SE 9: The **Java Platform Module System**, **interactive Java with JShell**, and many other Java 9 topics
- Object-oriented programming emphasizing current idioms
- **JavaFX GUI, graphics (2D and 3D), animation and video** coverage
- **Generics** and **generic collections**
- Deep **lambdas** and **streams** coverage
- **Concurrency** and **multi-core performance**
- Database: **JDBC** and **JPA**
- **Object-oriented design case study** with full Java code implementation

Flexibility Using Java SE 8 or the New Java SE 9

8
9

To meet the needs of our diverse audiences, we designed the book for professionals interested in Java SE 8 or Java SE 9, which from this point forward we'll refer to simply as Java 8 and Java 9, respectively. Features first introduced in Java 8 or Java 9 are accompanied by an 8 or 9 icon, respectively, in the margin, like those to the left of this paragraph. The new

Java 9 capabilities are covered in clearly marked chapters and sections. Figures 1 and 2 list some key Java 8 and Java 9 features that we cover, respectively.

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

Fig. 1 | Some key features we cover that were introduced in Java 8.

Java 9 features	
New Java Platform Module System chapter	_ is no longer allowed as an identifier
New JShell chapter	Mentions of:
`private` interface methods	`CompletableFuture` enhancements
Effectively `final` variables can be used in `try`-with-resources statements	Stack Walking API
	JEP 254, Compact Strings
Collection factory methods	Overview of Java 9 security enhancements
HTML5 Javadoc enhancements	Object serialization security enhancements
Regular expressions: New `Matcher` methods	Enhanced deprecation
`Stream` interface enhancements	Features removed from Java in Java 9
JavaFX 9 skin APIs and other enhancements	Features proposed for future removal

Fig. 2 | Some key new features we cover that were introduced in Java 9.

Java 9 for Programmers Modular Organization

Java 9 for Programmers features a modular organization. Many Java 9 features are integrated throughout the book. The pure Java 9 chapters (23, 27 and 28) are shown in **bold**:

Part 1: Introduction
Chapter 1, Introduction 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
Chapter 23, Introduction to JShell: Java 9's REPL for Interactive Java

Part 2: Core Programming Topics
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, Input/Output Streams, NIO and XML Serialization

Part 3: 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

Part 4: JavaFX Graphical User Interfaces, Graphics, Animation and Video
Chapter 12, JavaFX Graphical User Interfaces: Part 1
Chapter 13, JavaFX GUI: Part 2
Chapter 20, JavaFX Graphics, Animation and Video

Part 5: Generics, Generic Collections, Lambdas and Streams
Chapter 16, Generic Collections
Chapter 17, Lambdas and Streams
Chapter 18, Recursion
Chapter 19, Generic Classes and Methods: A Deeper Look

Part 6: Concurrency and Multi-Core Performance
Chapter 21, Concurrency and Multi-Core Performance

Part 7: Database-Driven Desktop Development
Chapter 22, Accessing Databases with JDBC
Chapter 24, Java Persistence API (JPA)

Part 8: Modularity and Additional Java 9 Topics
Chapter 27, Java Platform Module System
Chapter 28, Additional Java 9 Topics

Part 9: Object-Oriented Design
Chapter 25, ATM Case Study, Part 1: Object-Oriented Design with the UML
Chapter 26, ATM Case Study Part 2: Implementing an Object-Oriented Design

Introduction and Programming Fundamentals (Parts 1 and 2)

Chapters 1 through 7 provide an example-driven treatment of traditional fundamental programming topics. This book features an early objects approach—see the section "Object-Oriented Programming" later in this Preface. Note in the preceding outline that Part 1 includes the Chapter 23 on Java 9's new JShell.

Flexible Coverage of Java 9: The Module System, JShell and Other Java 9 Topics

Java 9 was still under development when this book was published. In addition to the topics discussed in this section, many Java 9 topics are integrated throughout the book. Check the following website for updates to this content:

```
http://www.deitel.com/books/Java9FP
```

9 *The Java Platform Module System*
Chapter 27 includes a substantial treatment of the Java Platform Module System (JPMS)—Java 9's most important new software-engineering technology. Modularity—the result of Project Jigsaw—helps professional developers be more productive as they build, maintain and evolve large systems. Figure 3 outlines the chapter's topics.

Java Platform Module System chapter outline

Module Declarations
`requires`, `requires transitive`, `exports` and `exports...to`, `uses`, `provides...with`, `open`, `opens` and `opens...to`

Modularized `Welcome` App
`Welcome` App's Structure, Class `Welcome`, `module-info.java`, Module-Dependency Graph, Compiling a Module, Running an App from a Module's Exploded Folders, Packaging a Module into a Modular JAR File, Running the `Welcome` App from a Modular JAR File, Aside: Classpath vs. Module Path

Creating and Using a Custom Module
Exporting a Package for Use in Other Modules, Using a Class from a Package in Another Module, Compiling and Running the Example, Packaging the App into Modular JAR Files, Strong Encapsulation and Accessibility

Module-Dependency Graphs: A Deeper Look
`java.sql`, `java.se`, Browsing the JDK Module Graph, Error: Module Graph with a Cycle

Migrating Code to Java 9
Unnamed Module, Automatic Modules, `jdeps`: Java Dependency Analysis

Resources in Modules; Using an Automatic Module
Automatic Modules, Requiring Multiple Modules, Opening a Module for Reflection, Module-Dependency Graph, Compiling the Module, Running a Modularized App

Creating Custom Runtimes with `jlink`
Listing the JRE's Modules, Custom Runtime Containing Only `java.base`, Creating a Custom Runtime for the `Welcome` App, Executing the `Welcome` App Using a Custom Runtime, Using the Module Resolver on a Custom Runtime

Services and `ServiceLoader`
Service-Provider Interface, Loading and Consuming Service Providers, `uses` Module Directive and Service Consumers, Running the App with No Service Providers, Implementing a Service Provider, `provides...with` Module Directive and Declaring a Service Provider, Running the App with One Service Provider, Implementing a Second Service Provider, Running the App with Two Service Providers

Fig. 3 | Java Platform Module System (Chapter 27) outline.

9 *JShell: Java 9's REPL (Read-Eval-Print-Loop) for Interactive Java*
JShell provides a friendly environment that enables you to quickly explore, discover and experiment with Java's language features and its extensive libraries. JShell replaces the tedious cycle of editing, compiling and executing with its **read-evaluate-print-loop**. Rather than complete programs, you write JShell commands and Java code snippets. When you enter a snippet, JShell *immediately*

- **reads** it,
- **evaluates** it and
- **prints** messages that help you see the effects of your code, then it
- **loops** to perform this process again for the next snippet.

As you work through Chapter 23's scores of examples, you'll see how JShell and its **instant feedback** keep your attention, enhance your performance and speed the learning and software-development processes.

You'll find JShell easy and fun to use. It will help you learn Java features faster and more deeply and will help you quickly verify that these features work as they're supposed to. JShell encourages you to dig in. You'll appreciate how JShell helps you rapidly prototype key code segments and discover and experiment with new APIs.

We chose a modular approach with the JShell content packaged in Chapter 23. The chapter:

1. is **easy to include or omit**.
2. is organized as a series of 16 sections, many of which are designed to be covered after a specific earlier chapter of the book (Fig. 4).
3. offers rich coverage of JShell's capabilities. It's **example-intensive**—you should do each of the examples. Get JShell into your fingertips. You'll appreciate how quickly and conveniently you can do things.
4. includes **dozens of Self-Review Exercises, each with an answer**. These exercises can be done after you read Chapter 2 and Section 23.3. As you do each of them, flip the page and check your answer. You'll master the basics of JShell quickly. Then as you do the remaining examples you'll master the vast majority of JShell's capabilities.

JShell discussions	Can be covered after
Section 23.3 introduces **JShell**, including starting a session, executing statements, declaring variables, evaluating expressions, JShell's **type-inference** capabilities and more. Section 23.4 discusses command-line input with **`Scanner`** in JShell.	Chapter 2, Introduction to Java Applications; Input/Output and Operators
Section 23.5 discusses how to declare and use **classes** in JShell, including how to load a Java source-code file containing an existing class declaration. Section 23.6 shows how to use JShell's **auto-completion** capabilities to discover a class's capabilities and JShell commands.	Chapter 3, Introduction to Classes, Objects, Methods and Strings
Section 23.7 presents additional JShell auto-completion capabilities for **experimentation and discovery**, including viewing method parameters, documentation and method overloads. Section 23.8 shows how to declare and use methods in JShell, including **forward referencing** a method that does not yet exist in the JShell session.	Chapter 6, Methods: A Deeper Look

Fig. 4 | Chapter 23 JShell discussions that may be covered after specific earlier chapters. (Part 1 of 2.)

JShell discussions	Can be covered after
Section 23.9 shows how **exceptions** are handled in JShell.	Chapter 7, Arrays and `ArrayLists`
Section 23.10 shows how to add existing **packages** to the classpath and import them for use in JShell.	Chapter 8, Classes and Objects: A Deeper Look
The remaining JShell sections are reference material that can be covered after Section 23.10. Topics include using an external editor, a summary of JShell commands, getting help in JShell, additional features of `/edit` command, `/reload` command, `/drop` command, feedback modes, other JShell features configurable with `/set`, keyboard shortcuts for snippet editing, how JShell reinterprets Java for interactive use and IDE JShell support.	

Fig. 4 | Chapter 23 JShell discussions that may be covered after specific earlier chapters. (Part 2 of 2.)

9 *Chapter 28: Additional Java 9 Topics*
Chapter 28 reviews the list of Java 9 features discussed throughout the book, then presents live-code examples and discussions of several additional features from Fig. 2. The chapter also lists some Java 9 features that are beyond the book's scope, points out features that are being removed in Java 9 and lists features that are proposed for future removal.

Object-Oriented Programming (Part 3)

Object-oriented programming. We introduce object-oriented concepts and terminology in Chapter 1. You'll develop customized classes and objects in Chapter 3.

Early objects real-world case studies. The early classes and objects presentation in Chapters 3–7 features `Account`, `AutoPolicy`, `Time`, `Employee`, `GradeBook` and `Card` shuffling-and-dealing case studies, gradually introducing deeper OO concepts.

Inheritance, Interfaces, Polymorphism and Composition. The deeper treatment of object-oriented programming in Chapters 8–10 features additional real-world case studies, including class `Time`, an `Employee` class hierarchy, and a `Payable` interface implemented in disparate `Employee` and `Invoice` classes. We explain the use of current idioms, such as "**programming to an interface not an implementation**" and "**preferring composition to inheritance**" in building industrial-strength applications.

Exception handling. We integrate basic exception handling beginning in Chapter 7 then present a deeper treatment in Chapter 11. Exception handling is important for building **mission-critical** and **business-critical** applications. 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" and how your code should handle those exceptions.

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 collection.

JavaFX GUI, Graphics (2D and 3D), Animation and Video Coverage (Part 4)

We've significantly updated our JavaFX presentation, replacing our Swing GUI and graphics coverage. In Chapters 12–13, we use JavaFX and **Scene Builder**—a drag-and-drop tool for creating JavaFX GUIs quickly and conveniently—to build several apps demonstrating various JavaFX layouts, controls and event-handling capabilities. In Swing, drag-and-drop tools and their generated code were *IDE dependent*. JavaFX Scene Builder is a standalone tool that you can use separately or with any of the Java IDEs to do **portable drag-and-drop GUI design**. In Chapter 20, we present JavaFX 2D and 3D graphics, animation and video capabilities. Despite the fact that the JavaFX chapters are spread out in the book, **Chapter 20 can be covered immediately after Chapter 13**.

Integrating Swing GUI Components in JavaFX GUIs

With JavaFX, you still can use your favorite Swing capabilities. For example, in Chapter 22, we demonstrate how to display database data in a Swing `JTable` component that's embedded in a JavaFX GUI via a JavaFX 8 `SwingNode`. As you explore Java further, you'll see that you also can incorporate JavaFX capabilities into your Swing GUIs.

Generics and Generic Collections (Part 5)

We begin with generic class `ArrayList` in Chapter 7. Our later discussions (Chapters 16–19) provide a deeper treatment of **generic collections**—showing how to use the built-in collections of the Java API. We discuss **recursion**, which is important for many reasons including implementing tree-like, data-structure classes.

We then show how to implement **generic methods and classes**. Rather than building custom generic data structures, most programmers should use the pre-built generic collections. **Lambdas and streams** (introduced in Chapter 17) are especially useful for working with generic collections.

Flexible Lambdas and Streams Coverage (Chapter 17)

The most significant new features in Java 8 were lambdas and streams. This book has several audiences, including

- those who'd like a significant treatment of lambdas and streams
- those who want a basic introduction with a few simple examples
- those who do not want to use lambdas and streams yet.

For this reason, we've placed most of the lambdas and streams treatment in Chapter 17, which is architected as a series of *easy-to-include-or-omit* sections that are keyed to the book's earlier sections and chapters. We do integrate lambdas and streams into a few examples after Chapter 17, because their capabilities are so compelling.

In Chapter 17, you'll see that lambdas and streams can help you write programs faster, more concisely, more simply, with fewer bugs and that are easier to **parallelize** (to realize performance improvements on **multi-core systems**) than programs written with previous techniques. You'll see that "functional programming" with lambdas and streams complements object-oriented programming.

Many of Chapter 17's sections are written so they can be covered earlier in the book (Fig. 5). After reading Chapter 17, you'll be able to cleverly reimplement many examples throughout the book.

Lambdas and streams discussions	Can be covered after
Sections 17.2–17.5 introduce basic lambda and streams capabilities that you can use to **replace counting loops**, and discuss the mechanics of how streams are processed.	Chapter 5, Control Statements: Part 2; Logical Operators
Section 17.6 introduces **method references** and additional streams capabilities.	Chapter 6, Methods: A Deeper Look
Section 17.7 introduces streams capabilities that process **one-dimensional arrays**.	Chapter 7, Arrays and `ArrayLists`
Sections 17.8–17.10 demonstrate additional streams capabilities and present various **functional interfaces used in streams processing**.	Section 10.10 which introduces Java 8 interface features for the functional interfaces that support lambdas and streams.
Section 17.11 shows how to use lambdas and streams to **process collections of `String` objects**.	Chapter 14, Strings, Characters and Regular Expressions
Section 17.12 shows how to use lambdas and streams to **process a `List<Employee>`**.	Chapter 16, Generic Collections
Section 17.13 shows how to use lambdas and streams to **process lines of text from a file**.	Chapter 15, Files, Input/Output Streams, NIO and XML Serialization
Section 17.14 introduces streams of random values	All earlier Chapter 17 sections.
Section 17.15 introduces infinite streams	All earlier Chapter 17 sections.
Section 17.16 shows how to use lambdas to implement **JavaFX event-listener interfaces**.	Chapter 12, JavaFX Graphical User Interfaces: Part 1
Chapter 21, Concurrency and Multi-Core Performance, shows that programs using lambdas and streams are often easier to **parallelize** so they can take advantage of **multi-core architectures** to enhance performance. The chapter demonstrates **parallel stream processing** and shows that **`Arrays` method `parallelSort`** improves performance on **multi-core architectures** when sorting large arrays.	

Fig. 5 | Java 8 lambdas and streams discussions and examples.

Concurrency and Multi-Core Performance (Part 6)

We were privileged to have as a reviewer of the previous edition of this book (*Java SE 8 for Programmers*) Brian Goetz, co-author of *Java Concurrency in Practice* (Addison-Wesley). We updated Chapter 21, Concurrency and Multi-Core Performance, with Java 8 technology and idiom. We added a **`parallelSort` vs. `sort`** example that uses the **Java 8 Date/Time API** to time each operation and demonstrate `parallelSort`'s better performance on a **multi-core** system. We included a **Java 8 parallel vs. sequential stream processing** example, again using the Date/Time API to show performance improvements. We added a

Java 8 **`CompletableFuture`** example that demonstrates sequential and parallel execution of long-running calculations and we discuss **`CompletableFuture` enhancements** in Chapter 28, Additional Java 9 Topics. 9

JavaFX concurrency. We now use **JavaFX concurrency** features, including class `Task` to execute long-running tasks in separate threads and display their results in the JavaFX application thread, and the `Platform` class's `runLater` method to schedule a `Runnable` for execution in the JavaFX application thread.

Database: JDBC and JPA (Part 7)

JDBC. Chapter 22 covers the widely used **JDBC** and uses the **Java DB database management system**. The chapter introduces **Structured Query Language** (**SQL**) and features a case study on developing a JavaFX database-driven address book that demonstrates **prepared statements**. In JDK 9, Oracle no longer bundles Java DB, which is simply an Oracle-branded version of Apache Derby. JDK 9 users can download and use Apache Derby instead (`https://db.apache.org/derby/`). 9

Java Persistence API. Chapter 24 covers the newer **Java Persistence API** (**JPA**)—a standard for **object relational mapping** (**ORM**) that uses JDBC "under the hood." ORM tools can look at a database's **schema** and generate a set of classes that enabled you to interact with a database without having to use JDBC and SQL directly. This speeds database-application development, reduces errors and produces more portable code.

Object-Oriented Design Case Study (Part 9)

Developing an Object-Oriented Design and Java Implementation of an ATM. Chapters 25–26 include a case study on object-oriented design using the UML (Unified Modeling Language™)—a 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 **complete Java implementation**. Readers often report having a "light-bulb moment"—the case study helps them "tie it all together" and understand object orientation more deeply.

Teaching Approach

Java 9 for Programmers contains hundreds of complete working code examples. We stress program clarity and concentrate on building well-engineered software.

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

```
comments appear in green
keywords appear in dark blue
errors appear in red
constants and literal values appear in light blue
all other code appears in black
```

Code Highlighting. We place transparent yellow rectangles around key code segments.

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

Objectives. The chapter objectives provide a high-level overview of the chapter's contents.

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

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

Software Development Wisdom

We include hundreds of tips to help you focus on important aspects of software development. These represent the best we've gleaned from a combined nine decades of programming and teaching experience in industry and academia.

Good Programming Practice 1.1

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 1.1

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

Error-Prevention Tip 1.1

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

Performance Tip 1.1

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

Portability Tip 1.1

The Portability Tips *help you write code that will run on a variety of platforms. Such portability is a key aspect of Java's mandate.*

Software Engineering Observation 1.1

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

Look-and-Feel Observation 1.1

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

JEPs, JSRs and the JCP

Throughout the book we encourage you to research various aspects of Java online. Some acronyms you're likely to see are JEP, JSR and JCP.

JEPs (JDK Enhancement Proposals) are used by Oracle to gather proposals from the Java community for changes to the Java language, APIs and tools, and to help create the roadmaps for future Java Standard Edition (Java SE), Java Enterprise Edition (Java EE) and Java Micro Edition (Java ME) platform versions and the JSRs (Java Specification Requests) that define them. The complete list of JEPs can be found at

```
http://openjdk.java.net/jeps/0
```

JSRs (Java Specification Requests) are the formal descriptions of Java platform features' technical specifications. Each new feature that gets added to Java (Standard Edition, Enterprise Edition or Micro Edition) has a JSR that goes through a review and approval process before the feature is added to Java. Sometimes JSRs are grouped together into an "umbrella" JSR. For example JSR 337 is the umbrella for Java 8 features, and JSR 379 is the umbrella for Java 9 features. The complete list of JSRs can be found at

```
https://www.jcp.org/en/jsr/all
```

The **JCP (Java Community Process)** is responsible for developing JSRs. JCP expert groups create the JSRs, which are publicly available for review and feedback. You can learn more about the JCP at:

```
https://www.jcp.org
```

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 audited our book against the CERT® Oracle Secure Coding Standard for Java

```
http://bit.ly/CERTSecureJava
```

and adhered to various secure coding practices as appropriate for a book at this level.

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 by employing programming practices that prevent system attacks from succeeding.

We'd like to thank Robert C. Seacord. A few years back, when Mr. Seacord was the Secure Coding Manager at CERT and an adjunct professor in the Carnegie Mellon University School of Computer Science, he was a technical reviewer for our book, *C How to Program, 7/e*, where he scrutinized our C programs from a security standpoint, recommending that we adhere to the *CERT C Secure Coding Standard*. This experience also influenced our coding practices in our C++, C# and Java books.

Java® 9 Fundamentals LiveLessons, Third Edition

Our *Java® 9 Fundamentals LiveLessons, 3/e* (summer 2017) video training product shows you what you need to know to start building robust, powerful software with Java. It includes 40+ hours of expert training synchronized with *Java® How to Program, Early Objects, 11/e—Java® 9 for Programmers* is a subset of that book. The videos are available to Safari subscribers at:

```
http://safaribooksonline.com
```

and may be purchased from

```
http://informit.com
```

Professionals like viewing the LiveLessons for reinforcement of core concepts and for additional insights.

Software Used in *Java® 9 for Programmers*

All the software you'll need is available for download. See the **Before You Begin** section that follows this Preface for links to each download. We wrote most of the examples in *Java 9® for Programmers* using the free Java Standard Edition Development Kit (JDK) 8. For the Java 9 content, we used the OpenJDK's early access version of JDK 9. All of the Java 9 programs run on the early access versions of JDK 9. All of the remaining programs run on both JDK 8 and early access versions of JDK 9, and were tested on Windows, macOS and Linux. Several of the later chapters also use the NetBeans IDE

8

9

```
http://netbeans.org
```

There are many other free Java IDEs, the most popular of which are Eclipse (from the Eclipse Foundation)

```
http://eclipse.org
```

and IntelliJ Community edition (from JetBrains)

```
https://www.jetbrains.com/idea/
```

Java Documentation Links

Throughout the book, we provide links to Java documentation where you can learn more about various topics that we present. For Java 8 documentation, the links begin with

8

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

and for Java 9 documentation, the links currently begin with

9

```
http://download.java.net/java/jdk9/
```

The Java 9 documentation links will change when Oracle releases Java 9—*possibly* to links beginning with

```
http://docs.oracle.com/javase/9/
```

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

```
http://www.deitel.com/books/Java9FP
```

Java® How to Program, 11/e

If you're an industry professional who also teaches college courses, you may want to consider our two college textbook versions of *Java 9 for Programmers*:

- *Java® How to Program, Early Objects, 11/e*, and
- *Java® How to Program, Late Objects, 11/e*

There are several approaches to teaching introductory course sequences in Java programming. The two most popular are the **early objects approach** and the **late objects approach**. The key difference between them is the order in which we present Chapters 1–7. The books have identical content in Chapter 8 and higher. These textbooks are supported by various instructor supplements. College instructors can request an examination copy of either of these books and obtain access to the supplements from their Pearson representative:

```
http://www.pearsonhighered.com/educator/replocator/
```

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 **book updates**, visit

```
http://www.deitel.com/books/Java9FP
```

subscribe to the ***Deitel® Buzz Online* newsletter** at

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

and join the **Deitel social networking communities** on

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

Acknowledgments

We'd like to thank Barbara Deitel for long hours devoted to technical research on this project. We're fortunate to have worked with the dedicated team of publishing professionals at Pearson. We appreciate the efforts and 22-year mentorship of our friend and professional colleague Mark L. Taub, Editor-in-Chief of the Pearson Technology Group. Mark and his team publish our professional books, LiveLessons video products and Learning Paths in the Safari service (`http://www.safaribooksonline.com`). Kristy Alaura recruited the book's reviewers and managed the review process. Sandra Schroeder and Julie Nahil managed the book's production. We selected the cover art and Chuti Prasertsith designed the cover.

Reviewers

We wish to acknowledge the efforts of our reviewers—a distinguished group of Oracle Java team members, Oracle Java Champions and other industry professionals. They scrutinized the text and the programs and provided countless suggestions for improving the presentation. Any remaining faults in the book are our own.

We appreciate the guidance of JavaFX experts Jim Weaver and Johan Vos (co-authors of *Pro JavaFX 8*), Jonathan Giles and Simon Ritter on the JavaFX chapters.

Reviewers: Lance Anderson (Java Platform Module System and Additional Java 9 Topics chapters; Principle Member of Technical Staff, Oracle), Alex Buckley (Java Platform Module System chapter; Specification Lead, Java Language & VM, Oracle), Robert Field (JShell chapter only; JShell Architect, Oracle), Trisha Gee (JetBrains, Java Champion), Jonathan Giles (Consulting Member of Technical Staff, Oracle), Brian Goetz (JShell and Java Platform Module System chapters; Oracle's Java Language Architect), Maurice Naftalin (JShell chapter only; Java Champion), José Antonio González Seco (Consultant), Bruno Souza (President of SouJava—the Brazilian Java Society, Java Specialist at ToolsCloud, Java Champion and SouJava representative at the Java Community Process), Dr. Venkat Subramaniam (President, Agile Developer, Inc. and Instructional Professor, University of Houston) and Johan Vos (CTO, Cloud Products at Gluon, Java Champion).

A Special Thank You to Robert Field

Robert Field, Oracle's JShell Architect reviewed the new JShell chapter, responding to our numerous emails in which we asked JShell questions, reported bugs we encountered as JShell evolved and suggested improvements. It was a privilege having our content scrutinized by the person responsible for JShell.

A Special Thank You to Brian Goetz

Brian Goetz, Oracle's Java Language Architect and Specification Lead for Java 8's Project Lambda, and co-author of *Java Concurrency in Practice*, did a full-book review of our book *Java® How to Program, Early Objects, 10/e*. He provided us with an extraordinary collection of insights and constructive comments. For *Java® 9 for Programmers*, he did a detailed review of our new JShell and Java Platform Module System chapters and answered our Java questions throughout the project.

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

```
deitel@deitel.com
```

We'll respond promptly. We wish you great success!

Paul and Harvey Deitel

About the Authors

Paul J. Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT and has over 35 years of experience in computing. 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 (now Oracle), Dell, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world's best-selling programming-language professional book/textbook/video authors.

Dr. Harvey M. Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has over 55 years of experience in computing. Dr. Deitel earned B.S. and M.S. degrees in Electrical Engineering from MIT and a Ph.D. in Mathematics from Boston University—he studied computing in each of these programs before they spun off Computer Science programs. He has extensive industry and 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 more than 100 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 academic, corporate, 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, Swift, iOS, C++, C, Visual C#®, object technology, Internet and web programming and a growing list of additional programming and software development courses.

Through its 42-year publishing partnership with Pearson/Prentice Hall, Deitel & Associates, Inc., publishes leading-edge programming professional books and textbooks in print and e-book formats available from booksellers worldwide, and **LiveLessons** video courses (available at `informit.com` an `safaribooksonline.com`) and Learning Paths (available at `safaribooksonline.com`). Deitel & Associates, Inc. and the authors can be reached at:

```
deitel@deitel.com
```

To learn more about Deitel's corporate training curriculum, visit:

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

To request a proposal for worldwide on-site, instructor-led training, write to

```
deitel@deitel.com
```

Individuals wishing to purchase Deitel books and *LiveLessons* video training can do so through `amazon.com`, `informit.com` and other online retailers. 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/Java9FP
```

In addition, we provide getting-started videos 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()`).

Java SE Development Kit (JDK)

The software you'll need for this book is available free for download from the web. Most of the examples were tested with the Java SE Development Kit 8 (also known as JDK 8). The most recent JDK version is available from:

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

The current version of the JDK at the time of this writing is JDK 8 update 121.

Java SE 9

The Java SE 9-specific features that we discuss in optional sections and chapters require JDK 9. At the time of this writing, JDK 9 was available as a Developer Preview. If you're using this book before the final JDK 9 is released, see the section "Installing and Configuring JDK 9 Developer Preview" later in this Before You Begin. We also discuss in that section how you can manage multiple JDK versions on Windows, macOS and Linux.

JDK Installation Instructions

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

```
https://docs.oracle.com/javase/8/docs/technotes/guides/install/
   install_overview.html
```

You'll need to update the JDK version number in any version-specific instructions. For example, the instructions refer to `jdk1.8.0`, but the current version at the time of this writing is `jdk1.8.0_121`. If you're a Linux user, your distribution's software package manager

might provide an easier way to install the JDK. For example, you can learn how to install the JDK on Ubuntu here:

```
http://askubuntu.com/questions/464755/how-to-install-openjdk-8-on-
   14-04-lts
```

Setting the PATH Environment Variable

The `PATH` environment variable designates which directories your computer searches for applications, such as those for compiling and running 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. Instructions for various platforms are listed at:

```
https://docs.oracle.com/javase/8/docs/technotes/guides/install/
   install_overview.html
```

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 8 update 121:

- JDK on Windows:
  `C:\Program Files\Java\jdk1.8.0_121`
- macOS (formerly called OS X):
  `/Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home`
- Ubuntu Linux:
  `/usr/lib/jvm/java-8-oracle`

Depending on your platform, the JDK installation folder's name might differ. For Linux, the install location depends on the installer you use and possibly the Linux version as well. 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 with a new *update version number*. For example, at the time of this writing, the most recent JDK 8 release was update 121. For this version, the JDK-installation-directory name typically ends with `_121`.

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 macOS and Linux, add

```
.:
```

Setting the JAVA_HOME Environment Variable

Several chapters require 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. Because the steps for using them differ, we used only the JDK command-line tools for most of the book's examples. We provide getting-started videos that show how to download, install and use three popular IDEs—NetBeans, Eclipse and IntelliJ IDEA.

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. For the standalone NetBeans installer, visit:

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

For Java Enterprise Edition (Java EE) development, choose the Java EE version, which supports both Java SE and Java EE development.

Eclipse Downloads

You can download the Eclipse IDE from:

```
https://eclipse.org/downloads/eclipse-packages/
```

For Java SE development choose the Eclipse IDE for Java Developers. For Java Enterprise Edition (Java EE) development, choose the Eclipse IDE for Java EE Developers, which supports both Java SE and Java EE development.

IntelliJ IDEA Community Edition Downloads

You can download the free IntelliJ IDEA Community from:

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

The free version supports only Java SE development, but there is a paid version that supports other Java technologies.

Scene Builder

Our JavaFX GUI, graphics and multimedia examples (starting in Chapter 12) use the free Scene Builder tool, which enables you to create graphical user interfaces (GUIs) with drag-and-drop techniques. You can download Scene Builder from:

```
http://gluonhq.com/labs/scene-builder/
```

Obtaining the Code Examples

The *Java 9 for Programmers* examples are available for download at

```
http://www.deitel.com/books/Java9FP/
```

Click the **Download Code Examples** link to download a ZIP archive file containing the examples—typically, the file will be saved in you user account's `Downloads` folder.

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's `Documents/examples` subfolder on macOS or Linux

Installing and Configuring JDK 9 Developer Preview

Throughout the book, we introduce various new Java 9 features. The Java 9 features require JDK 9, which at the time of this writing was still early access software available from

```
https://jdk9.java.net/download/
```

This page provides installers for Windows and macOS (formerly Mac OS X). On these platforms, download the appropriate installer, double click it and follow the on-screen instructions. For Linux, the download page provides only a `tar.gz` archive file. You can download that file, then extract its contents to a folder on your system. *If you have both JDK 8 and JDK 9 installed*, we provide instructions below showing how to specify which JDK to use on Windows, macOS or Linux.

JDK Version Numbers

Prior to Java 9, JDK versions were numbered `1.X.0_`*updateNumber* where `X` was the major Java version. For example,

- Java 8's current JDK version number is `jdk1.8.0_121` and
- Java 7's final JDK version number was `jdk1.7.0_80`.

As of Java 9, the numbering scheme has changed. JDK 9 initially will be known as `jdk-9`. Eventually, there will be minor version updates that add new features, and security updates that fix security holes in the Java platform. These updates will be reflected in the JDK version numbers. For example, in `9.1.3`:

- `9`—is the major Java version number
- `1`—is the minor version update number and
- `3`—is the security update number.

So 9.2.5 would indicate the version of Java 9 for which there have been two minor version updates and five total security updates across all Java 9 major and minor versions. For the new version-numbering scheme's details, see JEP (Java Enhancement Proposal) 223 at

```
http://openjdk.java.net/jeps/223
```

Managing Multiple JDKs on Windows

On Windows, you use the `PATH` environment variable to tell the operating system where to find a JDK's tools. The instructions at

```
https://docs.oracle.com/javase/8/docs/technotes/guides/install/
   windows_jdk_install.html#BABGDJFH
```

specify how to update the `PATH`. Replace the JDK version number in the instructions with the JDK version number you wish to use—currently `jdk-9`. You should check your JDK 9's installation folder name for an updated version number. This setting will automatically be applied to each new **Command Prompt** you open.

If you prefer not to modify your system's `PATH`—perhaps because you're also using JDK 8—you can open a **Command Prompt** window then set the `PATH` only for that window. To do so, use the command

```
set PATH=location;%PATH%
```

where *location* is the full path to JDK 9's `bin` folder and `;%PATH%` appends the **Command Prompt** window's original `PATH` contents to the new `PATH`. Typically, the command would be

```
set PATH="C:\Program Files\Java\jdk-9\bin";%PATH%
```

Each time you open a new **Command Prompt** window to use JDK 9, you'll have to reissue this command.

Managing Multiple JDKs on macOS

On a Mac, you can determine which JDKs you have installed by opening a **Terminal** window and entering the command

```
/usr/libexec/java_home -V
```

which shows the version numbers, names and locations of your JDKs—note that `-V` is a capital `V`, not lowercase. On our system the following is displayed:

```
Matching Java Virtual Machines (2):
    9, x86_64:  "Java SE 9-ea  "/Library/Java/JavaVirtualMachines/
       jdk-9.jdk/Contents/Home
    1.8.0_121, x86_64:  "Java SE 8  "/Library/Java/
       JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home
```

the version numbers are `9` and `1.8.0_121`. In "Java SE 9-ea" above, "ea" means "early access."

To set the default JDK version, enter

```
/usr/libexec/java_home -v # --exec javac -version
```

where # is the version number of the specific JDK that should be the default. At the time of this writing, for JDK 8, # should be `1.8.0_121` and, for JDK 9, # should be `9`.

Next, enter the command:

```
export JAVA_HOME=`/usr/libexec/java_home -v #`
```

where # is the version number of the current default JDK. This sets the **Terminal** window's `JAVA_HOME` environment variable to that JDK's location. This environment variable will be used when launching JShell.

Managing Multiple JDKs on Linux

The way you manage multiple JDK versions on Linux depends on how you install your JDKs. If you use your Linux distribution's tools for installing software (we used `apt-get` on Ubuntu Linux), then on many Linux distributions you can use the following command to list the installed JDKs:

```
sudo update-alternatives --config java
```

If more than one is installed, the preceding command shows you a numbered list of JDKs—you then enter the number for the JDK you wish to use as the default. For a tutorial showing how to use `apt-get` to install JDKs on Ubuntu Linux, see

```
https://www.digitalocean.com/community/tutorials/how-to-install-
java-with-apt-get-on-ubuntu-16-04
```

If you installed JDK 9 by downloading the `tar.gz` file and extracting it to your system, you'll need to specify in a shell window the path to the JDK's `bin` folder. To do so, enter the following command in your shell window:

```
export PATH="location:$PATH"
```

where *location* is the path to JDK 9's `bin` folder. This updates the `PATH` environment variable with the location of JDK 9's commands, like `javac` and `java`, so that you can execute the JDK's commands in the shell window.

You're now ready to begin reading *Java 9 for Programmers*. We hope you enjoy the book!

1

Introduction and Test-Driving a Java Application

Objectives

In this chapter you'll:

- Understand the importance of Java.
- Review object-technology concepts.
- Understand a typical Java program-development environment.
- Test-drive a Java application.
- Review some key recent software technologies.
- See where to get your Java questions answered.

Outline

1.1 Introduction

Welcome to Java—one of the world's most widely used computer programming languages and, according to the TIOBE Index (`https://www.tiobe.com/tiobe-index/`), the world's most popular. For many organizations, Java is the preferred language for meeting their enterprise programming needs. It's also widely used for implementing Internet-based applications and software for devices that communicate over a network.

There are billions of personal computers in use and an even larger number of mobile devices with computers at their core. According to Oracle's 2016 JavaOne conference keynote presentation,, there are now 10 million Java developers worldwide and Java runs on 15 billion devices (Fig. 1.1), including two billion vehicles and 350 million medical devices. In addition, the explosive growth of mobile phones, tablets and other devices is creating significant opportunities for programming mobile apps.

Devices		
Access control systems	Airplane systems	ATMs
Automobiles	Blu-ray Disc™ players	Building controls
Cable boxes	Copiers	Credit cards
CT scanners	Desktop computers	e-Readers
Game consoles	GPS navigation systems	Home appliances
Home security systems	Internet-of-Things gateways	Light switches
Logic controllers	Lottery systems	Medical devices
Mobile phones	MRIs	Network switches
Optical sensors	Parking meters	Personal computers
Point-of-sale terminals	Printers	Robots
Routers	Servers	Smart cards
Smart meters	Smartpens	Smartphones
Tablets	Televisions	Thermostats
Transportation passes	TV set-top boxes	Vehicle diagnostic systems

Fig. 1.1 | Some devices that use Java.

Java Standard Edition

Java has evolved so rapidly that the eleventh edition of our sister book *Java How to Program*—based on **Java Standard Edition 8 (Java SE 8)** and the new **Java Standard Edition 9 (Java SE 9)**—was published just 21 years after the first edition. Java Standard Edition contains the capabilities needed to develop desktop and server applications. The book can be used conveniently with *either* Java SE 8 *or* Java SE 9 (released just after this book was published). For those who want to stay with Java 8 for a while, the Java SE 9 features are discussed in modular, easy-to-include-or-omit sections throughout this book.

Prior to Java SE 8, Java supported three programming paradigms:

- *procedural programming*,
- *object-oriented programming* and
- *generic programming*.

Java SE 8 added the beginnings of *functional programming with lambdas and streams*. In Chapter 17, we'll show how to use lambdas and streams to write programs faster, more concisely, with fewer bugs and that are easier to *parallelize* (i.e., perform multiple calculations simultaneously) to take advantage of today's *multi-core* hardware architectures to enhance application performance.

Java Enterprise Edition

Java is used in such a broad spectrum of applications that it has two other editions. The **Java Enterprise Edition (Java EE)** is geared toward developing large-scale, distributed networking applications and web-based applications. In the past, most computer applications ran on "standalone" computers (that is, not networked together). Today's applications can be written with the aim of communicating among the world's computers via the Internet and the web.

Java Micro Edition

The **Java Micro Edition (Java ME)**—a subset of Java SE—is geared toward developing applications for resource-constrained embedded devices, such as smartwatches, television set-top boxes, smart meters (for monitoring electric energy usage) and more. Many of the devices in Fig. 1.1 use Java ME.

1.2 Object Technology Concepts

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

1.2.1 Automobile as an Object

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

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

1.2.2 Methods and Classes

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

1.2.3 Instantiation

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

1.2.4 Reuse

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

1.2.5 Messages and Method Calls

When you drive a car, pressing its gas pedal sends a *message* to the car to perform a task—that is, to go faster. Similarly, you *send messages to an object*. Each message is implemented

as a **method call** that tells a method of the object to perform its task. For example, a program might call a bank-account object's *deposit* method to increase the account's balance.

1.2.6 Attributes and Instance Variables

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

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

1.2.7 Encapsulation and Information Hiding

Classes (and their objects) **encapsulate**, i.e., encase, their attributes and methods. A class's (and its object's) attributes and methods are intimately related. Objects may communicate with one another, but they're normally not allowed to know how other objects are implemented—implementation details can be *hidden* within the objects themselves. This **information hiding**, as we'll see, is crucial to good software engineering.

1.2.8 Inheritance

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

1.2.9 Interfaces

Java also supports **interfaces**—collections of related methods that typically enable you to tell objects *what* to do, but not *how* to do it (we'll see exceptions to this in Java SE 8 and Java SE 9 when we discuss interfaces in Chapter 10). In the car analogy, a "basic-driving-capabilities" interface consisting of a steering wheel, an accelerator pedal and a brake pedal would enable a driver to tell the car *what* to do. Once you know how to use this interface for turning, accelerating and braking, you can drive many types of cars, even though manufacturers may *implement* these systems *differently*.

8
9

A class **implements** zero or more interfaces, each of which can have one or more methods, just as a car implements separate interfaces for basic driving functions, controlling the radio, controlling the heating and air conditioning systems, and the like. Just as car manufacturers implement capabilities *differently*, classes may implement an interface's methods *differently*. For example a software system may include a "backup" interface that offers the methods *save* and *restore*. Classes may implement those methods differently,

depending on the types of things being backed up, such as programs, text, audios, videos, etc., and the types of devices where these items will be stored.

1.2.10 Object-Oriented Analysis and Design (OOAD)

Soon you'll be writing programs in Java. How will you create the **code** (i.e., the program instructions) for your programs? Perhaps, like many programmers, you'll simply turn on your computer and start typing. This approach may work for small programs (like the ones we present in the early chapters of the book), but what if you were asked to create a software system to control thousands of automated teller machines for a major bank? Or suppose you were asked to work on a team of 1,000 software developers building the next generation of the U.S. air traffic control system? For projects so large and complex, you should not simply sit down and start writing programs.

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

1.2.11 The UML (Unified Modeling Language)

Although many different OOAD processes exist, a single graphical language for communicating the results of *any* OOAD process has come into wide use. The Unified Modeling Language (UML) is now the most widely used graphical scheme for modeling object-oriented systems. We present our first UML diagrams in Chapters 3 and 4, then use them in our deeper treatment of object-oriented programming through Chapter 11. In our ATM Software Engineering Case Study in Chapters 25–26 we present a simple subset of the UML's features as we guide you through an object-oriented design experience.

1.3 Java

The microprocessor revolution's most important contribution to date is that it enabled the development of personal computers. Microprocessors also have had a profound impact in intelligent consumer-electronic devices, including the recent explosion in the "Internet of Things." Recognizing this early on, Sun Microsystems in 1991 funded an internal corporate research project led by James Gosling, which resulted in a C++-based object-oriented programming language that Sun called Java. Using Java, you can write programs that will run on a great variety of computer systems and computer-controlled devices. This is sometimes called "write once, run anywhere."

Java drew the attention of the business community because of the phenomenal interest in the Internet. It's now used to develop large-scale enterprise applications, to enhance the functionality of web servers (the computers that provide the content we see

in our web browsers), to provide applications for consumer devices (cell phones, smartphones, television set-top boxes and more), to develop robotics software and for many other purposes. It's also the key language for developing Android smartphone and tablet apps. Sun Microsystems was acquired by Oracle in 2010.

Java has become the most widely used general-purpose programming language with more than 10 million developers. In this book, you'll study the two most recent versions of Java—Java Standard Edition 8 (Java SE 8) and Java Standard Edition 9 (Java SE 9).

Java Class Libraries

You can create each class and method you need to form your programs. However, most Java programmers take advantage of the rich collections of existing classes and methods in the **Java class libraries**, also known as the **Java APIs (Application Programming Interfaces)**.

Performance Tip 1.1

Using Java API classes and methods instead of writing your own versions can improve program performance, because they're carefully written to perform efficiently. This also shortens program development time.

Android

Android is the fastest growing mobile and smartphone operating system. There are now approximately 6 million Android app developers worldwide[1] and Java is Android's primary development language (though apps can also be developed in C#, C++ and C). One benefit of developing Android apps is the openness of the platform. The operating system is open source and free.

The Android operating system was developed by Android, Inc., which was acquired by Google in 2005. In 2007, the Open Handset Alliance™

```
http://www.openhandsetalliance.com/oha_members.html
```

was formed to develop, maintain and evolve Android, driving innovation in mobile technology and improving the user experience while reducing costs. According to Statista.com, as of Q3 2016, Android had 87.8% of the global smartphone market share, compared to 11.5% for Apple., The Android operating system is used in numerous smartphones, e-reader devices, tablets, in-store touch-screen kiosks, cars, robots, multimedia players and more.

We present an introduction to Android app development in our book, *Android 6 for Programmers: An App-Driven Approach, Third Edition*. After you learn Java, you'll find it straightforward to begin developing and running Android apps. You can place your apps on Google Play (`play.google.com`), and if they're successful, you may even be able to launch a business.

1. `http://www.businessofapps.com/12-million-mobile-developers-worldwide-nearly-half-develop-android-first/`.

1.4 A Typical Java Development Environment

We now explain the steps to create and execute a Java application. Normally there are five phases—edit, compile, load, verify and execute. We discuss them in the context of the Java SE 8 Development Kit (JDK). See the *Before You Begin* section *for information on downloading and installing the JDK on Windows, Linux and macOS.*

Phase 1: Creating a Program

Phase 1 consists of editing a file with an *editor program*, normally known simply as an *editor* (Fig. 1.2). Using the editor, you type a Java program (typically referred to as **source code**), make any necessary corrections and save it on a secondary storage device, such as your hard drive. Java source code files are given a name ending with the **`.java` extension**, indicating that the file contains Java source code.

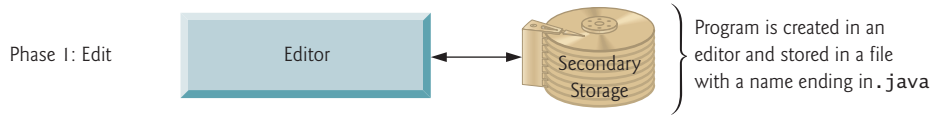


Fig. 1.2 | Typical Java development environment—editing phase.

Two editors widely used on Linux systems are `vi` and `emacs`. Windows provides **Notepad**. macOS provides **TextEdit**. Many freeware and shareware editors are also available online, including Notepad++ (`http://notepad-plus-plus.org`), EditPlus (`http://www.editplus.com`), TextPad (`http://www.textpad.com`), jEdit (`http://www.jedit.org`) and more.

Integrated development environments (IDEs) provide tools that support the software development process, such as editors, debuggers for locating **logic errors** that cause programs to execute incorrectly and more. The most popular Java IDEs are:

- Eclipse (`http://www.eclipse.org`)
- IntelliJ IDEA (`http://www.jetbrains.com`)
- NetBeans (`http://www.netbeans.org`)

On the book's website at

```
http://www.deitel.com/books/java9fp
```

we provide videos that show you how to execute this book's Java applications and how to develop new Java applications with Eclipse, NetBeans and IntelliJ IDEA.

Phase 2: Compiling a Java Program into Bytecodes

In Phase 2, you use the command **`javac`** (the **Java compiler**) to **compile** a program (Fig. 1.3). For example, to compile a program called `Welcome.java`, you'd type

```
javac Welcome.java
```

in your system's command window (i.e., the **Command Prompt** in Windows, the **Terminal** application in macOS) or a Linux shell (also called **Terminal** in some Linux versions). If the program compiles, the compiler produces a **`.class`** file called `Welcome.class`. IDEs typi-

cally provide a menu item, such as **Build** or **Make**, that invokes the `javac` command for you. If the compiler detects errors, you'll need to go back to Phase 1 and correct them. In Chapter 2, we'll say more about the kinds of errors the compiler can detect.

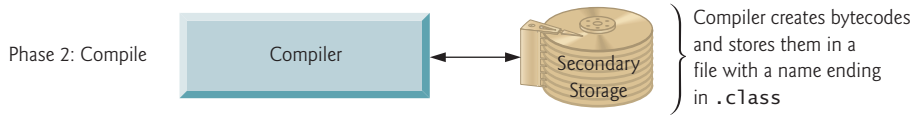


Fig. 1.3 | Typical Java development environment—compilation phase.

Common Programming Error 1.1

When using `javac`, if you receive a message such as "`bad command or filename`," "`javac: command not found`" or "`'javac' is not recognized as an internal or external command, operable program or batch file`," then your Java software installation was not completed properly. This indicates that the system's `PATH` environment variable was not set properly. Carefully review the installation instructions in the Before You Begin section of this book. On some systems, after correcting the `PATH`, you may need to reboot your computer or open a new command window for these settings to take effect.

The Java compiler translates Java source code into **bytecodes** that represent the tasks to execute in the execution phase (Phase 5). The **Java Virtual Machine** (**JVM**)—a part of the JDK and the foundation of the Java platform—executes bytecodes. A **virtual machine** (**VM**) is software that simulates a computer but hides the underlying operating system and hardware from the programs that interact with it. If the same VM is implemented on many computer platforms, applications written for that type of VM can be used on all those platforms. The JVM is one of the most widely used virtual machines. Microsoft's .NET uses a similar virtual-machine architecture.

Unlike machine-language instructions, which are *platform dependent*, bytecodes are *platform independent* and thus **portable**—without recompiling the source code, the same bytecodes can execute on any platform containing a JVM that understands the version of Java in which the bytecodes were compiled. The JVM is invoked by the **`java`** command. For example, to execute a Java application called `Welcome`, you'd type the command

```
java Welcome
```

in a command window to invoke the JVM, which would then initiate the steps necessary to execute the application. This begins Phase 3. IDEs typically provide a menu item, such as **Run**, that invokes the `java` command for you.

Phase 3: Loading a Program into Memory

In Phase 3, the JVM places the program in memory to execute it—this is known as **loading** (Fig. 1.4).The JVM's **class loader** takes the `.class` files containing the program's bytecodes and transfers them to primary memory. It also loads any of the `.class` files provided by Java that your program uses. The `.class` files can be loaded from a disk on your system or over a network (e.g., your local college or company network, or the Internet).

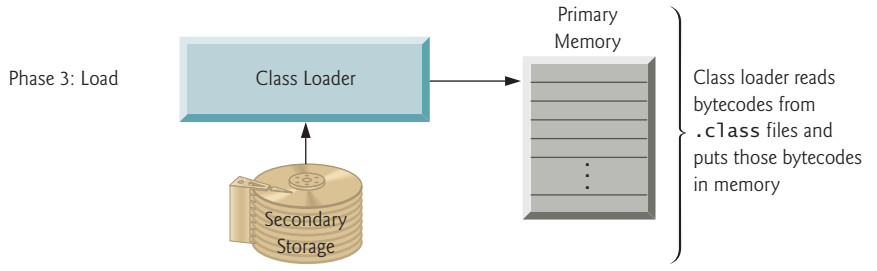


Fig. 1.4 | Typical Java development environment—loading phase.

Phase 4: Bytecode Verification

In Phase 4, as the classes are loaded, the **bytecode verifier** examines their bytecodes to ensure that they're valid and do not violate Java's security restrictions (Fig. 1.5). Java enforces strong security to make sure that Java programs arriving over the network do not damage your files or your system (as computer viruses and worms might).

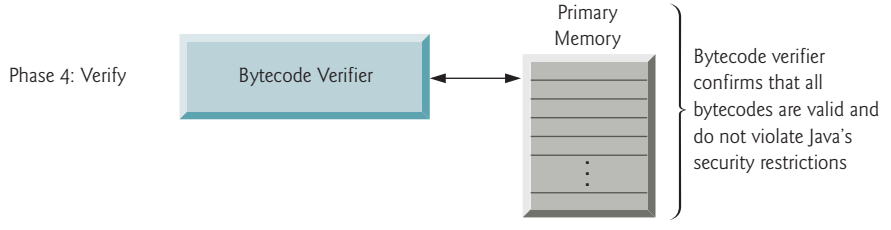


Fig. 1.5 | Typical Java development environment—verification phase.

Phase 5: Execution

In Phase 5, the JVM **executes** the bytecodes to perform the program's specified actions (Fig. 1.6). In early Java versions, the JVM was simply a Java-bytecode *interpreter*. Most programs would execute slowly, because the JVM would interpret and execute one bytecode at a time. Some modern computer architectures can execute several instructions in parallel. Today's JVMs typically execute bytecodes using a combination of interpretation and **just-in-time (JIT) compilation**. In this process, the JVM analyzes the bytecodes as they're interpreted, searching for *hot spots*—bytecodes that execute frequently. For these parts, a **just-in-time (JIT) compiler**, such as Oracle's **Java HotSpot™ compiler**, translates the bytecodes into the computer's machine language. When the JVM encounters these compiled parts again, the faster machine-language code executes. Thus programs actually go through *two* compilation phases—one in which Java code is translated into bytecodes (for portability across JVMs on different computer platforms) and a second in which, during execution, the bytecodes are translated into *machine language* for the computer on which the program executes.

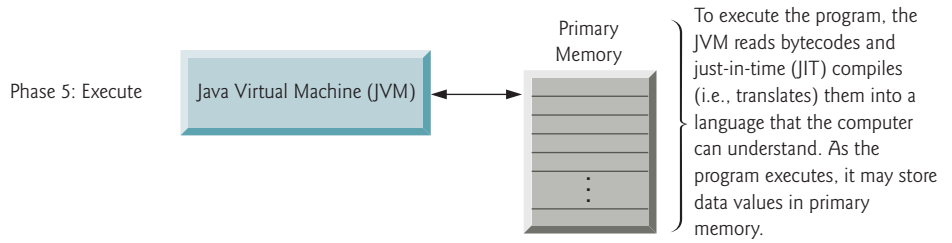


Fig. 1.6 | Typical Java development environment—execution phase.

Problems That May Occur at Execution Time

Programs might not work on the first try. Each of the preceding phases can fail because of various errors that we'll discuss throughout this book. For example, an executing program might try to divide by zero (an illegal operation for whole-number arithmetic in Java). This would cause the Java program to display an error message. If this occurred, you'd return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine whether the corrections fixed the problem(s). [*Note:* Most programs in Java input or output data. When we say that a program displays a message, we normally mean that it displays that message on your computer's screen.]

1.5 Test-Driving a Java Application

In this section, you'll run and interact with an existing Java **Painter** app, which you'll build in a later chapter. The elements and functionality you'll see are typical of what you'll learn to program in this book. Using the **Painter**'s graphical user interface (GUI), you choose a drawing color and pen size, then drag the mouse to draw circles in the specified color and size. You also can undo each drawing operation or clear the entire drawing. [*Note:* We emphasize screen features like window titles and menus (e.g., the **File** menu) in a **sans-serif font** and emphasize nonscreen elements, such as file names and program code (e.g., `ProgramName.java`), in a `fixed-width sans-serif font`.]

The steps in this section show you how to execute the **Painter** app from a **Command Prompt** (Windows), shell (Linux) or **Terminal** (macOS) window on your system. Throughout the book, we'll refer to these windows simply as *command windows*. We assume that the book's examples are located in `C:\examples` on Windows or in your user account's `Documents/examples` folder on Linux or macOS.

Checking Your Setup

Read the Before You Begin section that follows the Preface to set up Java on your computer and ensure that you've downloaded the book's examples to your hard drive.

Changing to the Completed Application's Directory

Open a command window and use the `cd` command to change to the folder for the **Painter** application:

- On Windows type the following command, then press *Enter*:

```
cd C:\examples\ch01\Painter
```

- On Linux/macOS, type the following command, then press *Enter*.

```
cd ~/Documents/examples/ch01/Painter
```

Compiling the Application

In the command window, type the following command then press *Enter* to compile all the files for the **Painter** example:

```
javac *.java
```

The `*` indicates that all files with names that end in `.java` should be compiled.

Running the Painter *Application*

Recall from Section 1.4 that the `java` command, followed by the name of an app's `.class` file (in this case, `Painter`), executes the application. Type the command `java Painter` then press *Enter* to execute the app. Figure 1.7 shows the **Painter** app running on Windows, Linux and macOS, respectively. The app's capabilities are identical across operating systems, so the remaining steps in this section show only Windows screen captures. Java commands are *case sensitive*—that is, uppercase letters are different from lowercase letters. It's important to type `Painter` with a capital `P`. Otherwise, the application will *not* execute. Also, if you receive the error message, "`Exception in thread "main" java.lang.NoClassDefFoundError: Painter`," your system has a `CLASSPATH` problem. Please refer to the Before You Begin section for instructions to help you fix this problem.

a) **Painter** app running on Windows

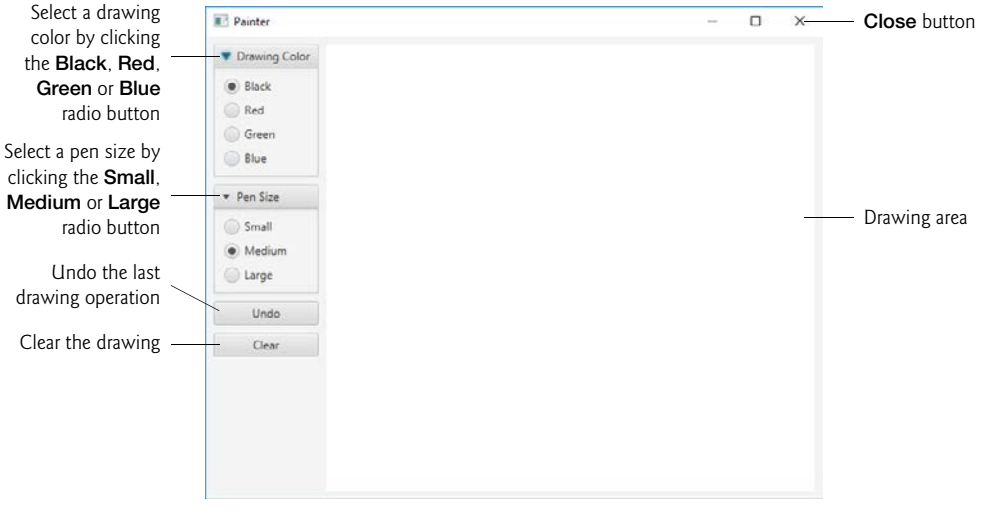
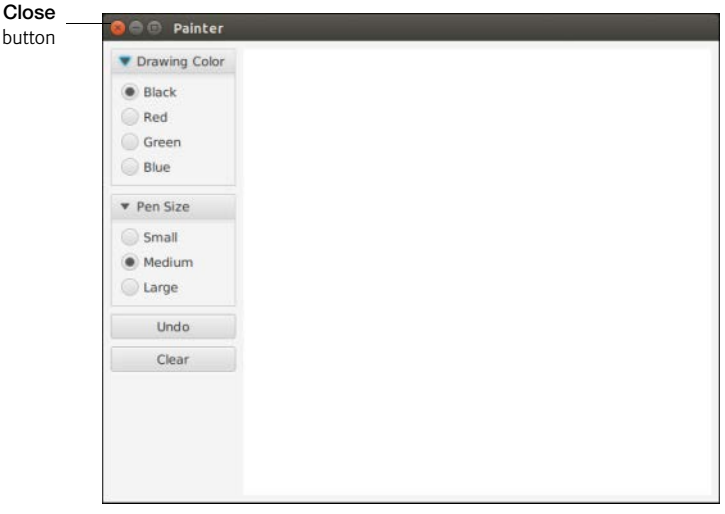


Fig. 1.7 | **Painter** app executing in Windows, Linux and macOS. (Part 1 of 2.)

b) **Painter** app running on Linux.



c) **Painter** app running on macOS.

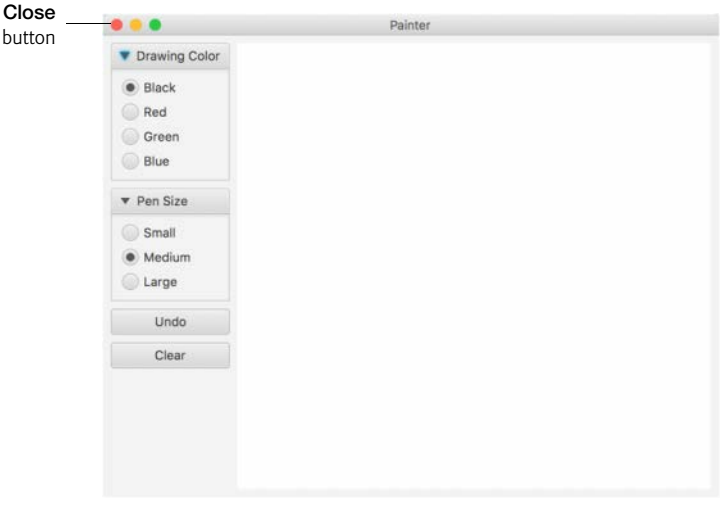


Fig. 1.7 | **Painter** app executing in Windows, Linux and macOS. (Part 2 of 2.)

Drawing the Flower Petals

In this section's remaining steps, you'll draw a red flower with a green stem, green grass and blue rain. We'll begin with the flower petals in a red, medium-sized pen. Change the drawing color to red by clicking the **Red** radio button. Next, drag your mouse on the drawing area to draw flower petals (Fig. 1.8). If you don't like a portion of what you've drawn, you can click the **Undo** button repeatedly to remove the most recent circles that were drawn, or you can begin again by clicking the **Clear** button.

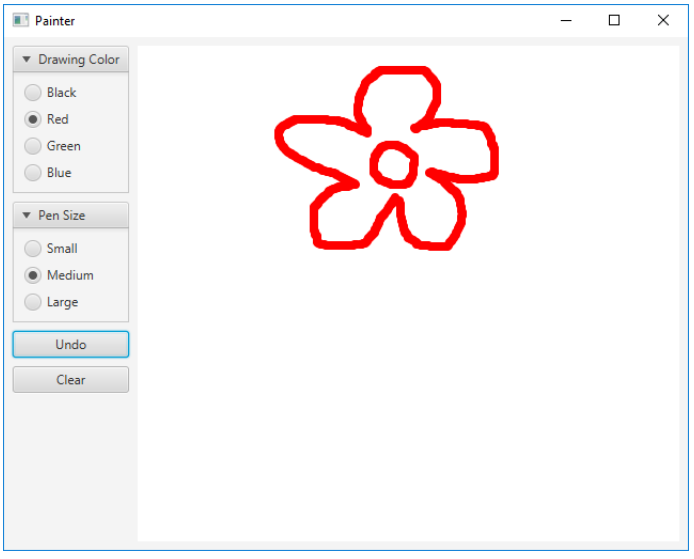


Fig. 1.8 | Drawing the flower petals.

Drawing the Stem, Leaves and Grass

Change the drawing color to green and the pen size to large by clicking the **Green** and **Large** radio buttons. Then, draw the stem and the leaves as shown in Fig. 1.9. Next, change the pen size to medium by clicking the **Medium** radio button, then draw the grass as shown in Fig. 1.9.

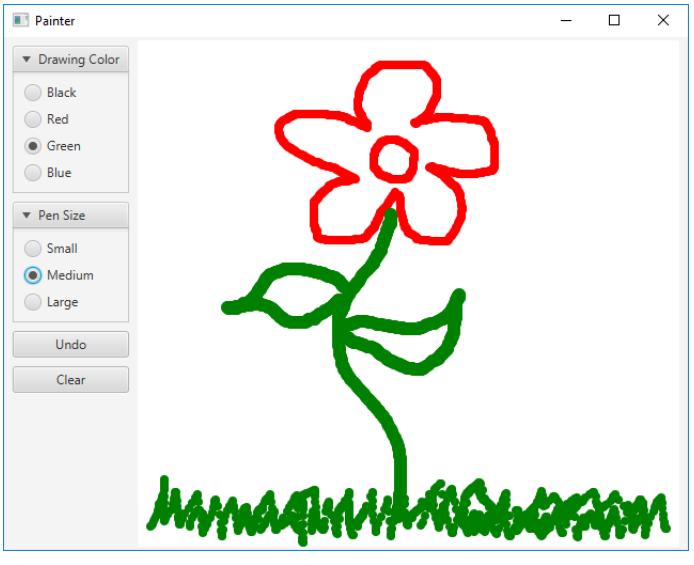


Fig. 1.9 | Drawing the stem and grass.

Drawing the Rain

Change the drawing color to blue and the pen size to small by clicking the **Blue** and **Small** radio buttons. Then, draw some rain as shown in Fig. 1.10.

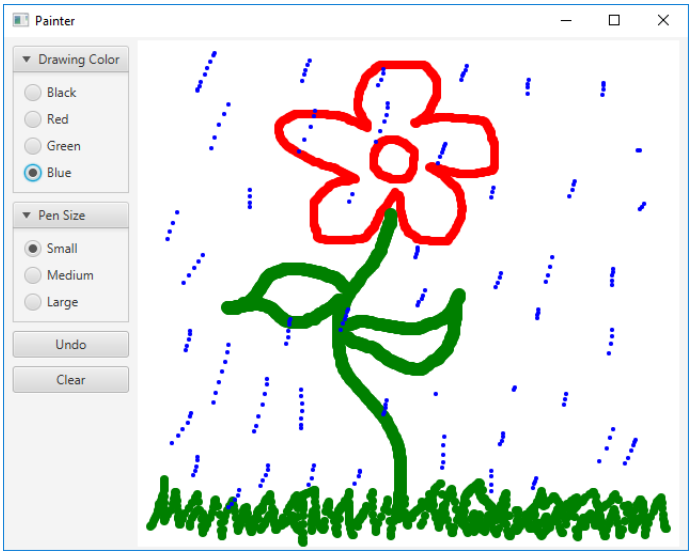


Fig. 1.10 | Drawing the rain.

Exiting the Painter App

At this point, you can close the **Painter** app. To do so, simply click the app's close box (shown for Windows, Linux and macOS in Fig. 1.7).

1.6 Software Technologies

Figure 1.11 lists a number of popular software technologies.

Technology	Description
Agile software development	**Agile software development** is a set of methodologies that try to get software implemented faster and using fewer resources. Check out the Agile Alliance (`www.agilealliance.org`) and the Agile Manifesto (`www.agilemanifesto.org`).
Refactoring	**Refactoring** involves reworking programs to make them clearer and easier to maintain while preserving their correctness and functionality. It's widely employed with agile development methodologies. Many IDEs contain built-in *refactoring tools* to do major portions of the reworking automatically.

Fig. 1.11 | Software technologies. (Part 1 of 3.)

Technology	Description
Design patterns	**Design patterns** are proven architectures for constructing flexible and maintainable object-oriented software. The field of design patterns tries to enumerate those recurring patterns, encouraging software designers to *reuse* them to develop better-quality software using less time, money and effort.
LAMP	**LAMP** is an acronym for the open-source technologies that many developers use to build web applications inexpensively—it stands for *Linux*, *Apache*, *MySQL* and *PHP* (or *Perl* or *Python*—two other popular scripting languages). MySQL is an open-source database-management system. PHP is a popular open-source server-side "scripting" language for developing web applications. Apache is the most popular web server software. The equivalent for Windows development is WAMP—*Windows*, *Apache*, *MySQL* and *PHP*.
Software as a Service (SaaS)	Software has generally been viewed as a product; most software still is offered this way. If you want to run an application, you buy a software package from a software vendor—often a CD, DVD or web download. You then install that software on your computer and run it as needed. As new versions appear, you upgrade your software, often at considerable cost in time and money. This process can become cumbersome for organizations that must maintain tens of thousands of systems on a diverse array of computer equipment. With **Software as a Service (SaaS)**, the software runs on servers elsewhere on the Internet. When that server is updated, all clients worldwide see the new capabilities—no local installation is needed. You access the service through a browser. Browsers are quite portable, so you can run the same applications on a wide variety of computers from anywhere in the world. Salesforce.com, Google, Microsoft and many other companies offer SaaS.
Platform as a Service (PaaS)	**Platform as a Service (PaaS)** provides a computing platform for developing and running applications as a service over the web, rather than installing the tools on your computer. Some PaaS providers are Google App Engine, Amazon EC2 and Windows Azure™.
Cloud computing	SaaS and PaaS are examples of cloud computing. You can use software and data stored in the "cloud"—i.e., accessed on remote computers (or servers) via the Internet and available on demand—rather than having it stored locally on your desktop, notebook computer or mobile device. This allows you to increase or decrease computing resources to meet your needs at any given time, which is more cost effective than purchasing hardware to provide enough storage and processing power to meet occasional peak demands. Cloud computing also saves money by shifting to the service provider the burden of managing these apps (such as installing and upgrading the software, security, backups and disaster recovery).
Software Development Kit (SDK)	**Software Development Kits (SDKs)** include the tools and documentation developers use to program applications.

Fig. 1.11 | Software technologies. (Part 2 of 3.)

Technology	Description
Big Data	The amount of data being produced worldwide is enormous and growing quickly. According to IBM, approximately 2.5 quintillion bytes (2.5 *exabytes*) of data are created daily,, and according to Salesforce.com, as of October 2015 90% of the world's data was created in just the prior 12 months!, According to an IDC study, the global data supply will reach 40 *zettabytes* (equal to 40 trillion gigabytes) annually by 2020., Figure 1.4 shows some common byte measurements. **Big data** applications deal with massive amounts of data and this field is growing quickly, creating lots of opportunity for software developers. Millions of IT jobs globally already are supporting big data applications.

Fig. 1.11 | Software technologies. (Part 3 of 3.)

Software is complex. Large, real-world software applications can take many months or even years to design and implement. When large software products are under development, they typically are made available to the user communities as a series of releases, each more complete and polished than the last (Fig. 1.12).

Version	Description
Alpha	*Alpha* software is the earliest release of a software product that's still under active development. Alpha versions are often buggy, incomplete and unstable and are released to a relatively small number of developers for testing new features, getting early feedback, etc. Alpha software also is commonly called *early access* software.
Beta	*Beta* versions are released to a larger number of developers later in the development process after most major bugs have been fixed and new features are nearly complete. Beta software is more stable, but still subject to change.
Release candidates	*Release candidates* are generally *feature complete*, (mostly) bug free and ready for use by the community, which provides a diverse testing environment—the software is used on different systems, with varying constraints and for a variety of purposes.
Final release	Any bugs that appear in the release candidate are corrected, and eventually the final product is released to the general public. Software companies often distribute incremental updates over the Internet.
Continuous beta	Software that's developed using this approach (for example, Google search or Gmail) generally does not have version numbers. It's hosted in the *cloud* (not installed on your computer) and is constantly evolving so that users always have the latest version.

Fig. 1.12 | Software product-release terminology.

1.7 Getting Your Questions Answered

There are many online forums in which you can get your Java questions answered and interact with other Java programmers. Some popular Java and general programming forums include:

- `StackOverflow.com`
- `Coderanch.com`
- The Oracle Java Forum—`https://community.oracle.com/community/java`
- `</dream.in.code>`—`http://www.dreamincode.net/forums/forum/32-java/`

2

Introduction to Java Applications; Input/Output and Operators

Objectives

In this chapter you'll:

- Write simple Java applications.
- Use input and output statements.
- Use Java's primitive types.
- Use arithmetic operators.
- Understand the precedence of arithmetic operators.
- Write decision-making statements.
- Use relational and equality operators.

Outline

2.1 Introduction

This chapter introduces Java programming. We begin with examples of programs that display (output) messages on the screen. We then present a program that obtains (inputs) two numbers from a user, calculates their sum and displays the result. You'll perform arithmetic calculations and save their results for later use. The last example demonstrates how to make decisions. The application compares two numbers, then displays messages that show the comparison results. You'll use the JDK command-line tools to compile and run this chapter's programs. If you prefer to use an integrated development environment (IDE), we've also posted getting-started videos at

```
http://www.deitel.com/books/java9fp
```

for the three most popular Java IDEs—Eclipse, NetBeans and IntelliJ IDEA.

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

A Java **application** is a computer program that executes when you use the **java command** to launch the Java Virtual Machine (JVM). Sections 2.2.1–2.2.2 discuss how to compile and run a Java application. First we consider a simple application that displays a line of text. Figure 2.1 shows the program followed by a box that displays its output.

```
1  // Fig. 2.1: Welcome1.java
2  // Text-printing program.
3
4  public class Welcome1 {
5     // main method begins execution of Java application
6     public static void main(String[] args) {
7        System.out.println("Welcome to Java Programming!");
8     } // end method main
9  } // end class Welcome1
```

```
Welcome to Java Programming!
```

Fig. 2.1 | Text-printing program.

The figure includes line numbers—they're *not* part of a Java program. Line 7 does the program's work—displaying the phrase `"Welcome to Java Programming!"` on the screen.

Commenting Your Programs

By convention, we begin every program with a comment indicating the figure number and the program's filename. The comment in line 1 begins with `//`, indicating that it's an **end-of-line comment**—it terminates at the end of the line on which the `//` appears. Line 2, by our convention, is a comment that describes the purpose of the program.

Java also has **traditional comments**, which can be spread over several lines as in

```
/* This is a traditional comment. It
   can be split over multiple lines */
```

These begin with the delimiter `/*` and end with `*/`. The compiler ignores all text between the delimiters. Java incorporated traditional comments and end-of-line comments from the C and C++ programming languages, respectively.

Java provides comments of a third type—**Javadoc comments**. These are delimited by `/**` and `*/`. The compiler ignores all text between the delimiters. Javadoc comments enable you to embed program documentation directly in your programs. Such comments are the preferred Java documenting format in industry. The **`javadoc` utility program** (part of the JDK) reads Javadoc comments and uses them to prepare program documentation in HTML5 web-page format. We use `//` comments throughout our code, rather than traditional or Javadoc comments, to save space.

Using Blank Lines

Blank lines (like line 3), space characters and tabs can make programs easier to read. Together, they're known as **white space**. The compiler ignores white space.

Declaring a Class

Line 4 begins a **class declaration** for class `Welcome1`. Every Java program consists of at least one class that you define. The **`class` keyword** introduces a class declaration and is immediately followed by the **class name** (`Welcome1`). **Keywords** are reserved for use by Java and are spelled with all lowercase letters. The complete list of keywords is shown in Appendix C.

In Chapters 2–7, every class we define begins with the **`public`** keyword. For now, we simply require it. You'll learn more about `public` and non-`public` classes in Chapter 8.

Filename for a `public` Class

A `public` class *must* be placed in a file that has a filename of the form *ClassName*`.java`, so class `Welcome1` is stored in the file `Welcome1.java`.

Common Programming Error 2.1

A compilation error occurs if a `public` class's filename is not exactly the same name as the class (in terms of both spelling and capitalization) followed by the `.java` extension.

Class Names and Identifiers

By convention, class names begin with a capital letter and capitalize the first letter of each word they include (e.g., `SampleClassName`). A class name is an **identifier**—a series of char-

acters consisting of letters, digits, underscores (_) and dollar signs ($) that does *not* begin with a digit and does *not* contain spaces. Some valid identifiers are `Welcome1`, `$value`, `_value`, `m_inputField1` and `button7`. The name `7button` is *not* a valid identifier because it begins with a digit, and the name `input field` is *not* a valid identifier because it contains a space. Normally, an identifier that does not begin with a capital letter is not a class name. Java is **case sensitive**—uppercase and lowercase letters are distinct—so `value` and `Value` are different (but both valid) identifiers.

Good Programming Practice 2.1

*By convention, every word in a class-name identifier begins with an uppercase letter. For example, the class-name identifier `DollarAmount` starts its first word, `Dollar`, with an uppercase `D` and its second word, `Amount`, with an uppercase `A`. This naming convention is known as **camel case**, because the uppercase letters stand out like a camel's humps.*

9 *Underscore (_) in Java 9*

As of Java 9, you can no longer use an underscore (_) by itself as an identifier.

Class Body

A **left brace** (at the end of line 4), `{`, begins the **body** of every class declaration. A corresponding **right brace** (at line 9), `}`, must end each class declaration. Lines 5–8 are indented.

Good Programming Practice 2.2

By convention, indent the entire body of each class declaration one "level" between the braces that delimit the class's body. This format emphasizes the class declaration's structure and makes it easier to read. We use three spaces to form a level of indent—many programmers prefer two or four spaces. Whatever you choose, use it consistently.

Declaring a Method

Line 5 is a comment indicating the purpose of lines 6–8 of the program. Line 6 is the starting point of every Java application. The **parentheses** after the identifier `main` indicate that it's a **method**. Java class declarations normally contain one or more methods. For a Java application, one of the methods *must* be called `main` and must be defined as in line 6; otherwise, the program will not execute. We'll explain the purpose of keyword `static` in Section 3.2.5. Keyword **`void`** indicates that this method will *not* return any information. The `String[] args` in parentheses is a required part of `main`'s declaration—we discuss this in Chapter 7.

The left brace at the end of line 6 begins the **body of the method declaration**. A corresponding right brace ends it (line 8). Line 7 is indented between the braces.

Good Programming Practice 2.3

Indent the entire body of each method declaration one "level" between the braces that define the method's body. This emphasizes the method's structure and makes it easier to read.

Performing Output with `System.out.println`

Line 7 displays the characters between the double quotation marks. The quotation marks themselves are *not* displayed. Together, the quotation marks and the characters between them are a **string**—also known as a **character string** or a **string literal**. White-space char-

acters in strings are *not* ignored by the compiler. Strings *cannot* span multiple lines of code—later we'll show how to conveniently deal with long strings.

The `System.out` object—which is predefined for you—is known as the **standard output object**. It allows a program to display information in the **command window** from which the program executes. In Microsoft Windows, the command window is the **Command Prompt**. In UNIX/Linux/macOS, the command window is called a **terminal** or a **shell**. Many programmers call it simply the **command line**.

Method `System.out.println` displays (or prints) a *line* of text in the command window. The string in the parentheses in line 7 is the method's **argument**. When `System.out.println` completes its task, it positions the output cursor (the location where the next character will be displayed) at the beginning of the next line in the command window. This is similar to what happens when you press the *Enter* key while typing in a text editor—the cursor appears at the beginning of the next line in the document.

The entire line 7, including `System.out.println`, the argument `"Welcome to Java Programming!"` in the parentheses and the **semicolon** (`;`), is called a **statement**. A method typically contains statements that perform its task. Most statements end with a semicolon.

2.2.1 Compiling the Application

We're now ready to compile and execute the program. We assume you're using the Java Development Kit's command-line tools, not an IDE. The following instructions assume that the book's examples are located in `c:\examples` on Windows or in your user account's `Documents/examples` folder on Linux/macOS.

Open a command window and change to the directory where the program is stored. Many operating systems use the command `cd` to change directories (or folders). On Windows, for example,

```
cd c:\examples\ch02\fig02_01
```

changes to the `fig02_01` directory. On UNIX/Linux/macOS, the command

```
cd ~/Documents/examples/ch02/fig02_01
```

changes to the `fig02_01` directory. To compile the program, type

```
javac Welcome1.java
```

If the program does not contain compilation errors, this command creates the file called `Welcome1.class` (known as `Welcome1`'s **class file**) containing the platform-independent Java bytecodes that represent our application. When we use the `java` command to execute the application on a given platform, the JVM will translate these bytecodes into instructions that are understood by the underlying operating system and hardware.

Common Programming Error 2.2

The compiler error message "`class Welcome1 is public, should be declared in a file named Welcome1.java`" indicates that the filename does not match the name of the `public` class in the file or that you typed the class name incorrectly when compiling the class.

Each compilation-error message contains the filename and line number where the error occurred. For example, `Welcome1.java:6` indicates that an error occurred at line 6 in `Welcome1.java`. The rest of the message provides information about the syntax error.

2.2.2 Executing the Application

Now that you've compiled the program, type the following command and press *Enter*:

```
java Welcome1
```

to launch the JVM and load the `Welcome1.class` file. The command *omits* the `.class` filename extension; otherwise, the JVM will *not* execute the program. The JVM calls `Welcome1`'s `main` method. Next, line 7 of `main` displays `"Welcome to Java Programming!"`. Figure 2.2 shows the program executing in a Microsoft Windows **Command Prompt** window. [*Note:* Many environments show command windows with black backgrounds and white text. We adjusted these settings to make our screen captures more readable.]

Error-Prevention Tip 2.1

When attempting to run a Java program, if you receive a message such as "`Exception in thread "main" java.lang.NoClassDefFoundError: Welcome1`," your `CLASSPATH` environment variable has not been set properly. Please carefully review the installation instructions in the Before You Begin section of this book. On some systems, you may need to reboot your computer or open a new command window after configuring the `CLASSPATH`.

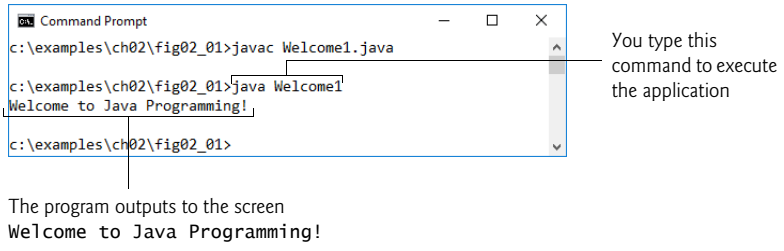


Fig. 2.2 | Executing `Welcome1` from the **Command Prompt**.

2.3 Modifying Your First Java Program

Let's modify the example in Fig. 2.1 to print text on one line by using multiple statements and to print text on several lines by using a single statement.

Displaying a Single Line of Text with Multiple Statements

`Welcome to Java Programming!` can be displayed several ways. Class `Welcome2`, shown in Fig. 2.3, uses two statements (lines 7–8) to produce the output shown in Fig. 2.1. From this point forward, we highlight the new and key features in each code listing. Lines 7–8 in method `main` display *one* line of text. The first statement uses `System.out`'s method `print` to display a string. Each `print` or `println` statement resumes displaying characters from where the last `print` or `println` statement stopped displaying characters. Unlike `println`, after displaying its argument, `print` does *not* position the output cursor at the beginning of the next line—the next character the program displays will appear *immediately after* the last character that `print` displays. So, line 8 positions the first character in its argument (the letter "`J`") immediately after the last character that line 7 displays (the *space character* before the string's closing double-quote character).

```
1  // Fig. 2.3: Welcome2.java
2  // Printing a line of text with multiple statements.
3
4  public class Welcome2 {
5     // main method begins execution of Java application
6     public static void main(String[] args) {
7        System.out.print("Welcome to ");
8        System.out.println("Java Programming!");
9     } // end method main
10 } // end class Welcome2
```

```
Welcome to Java Programming!
```

Fig. 2.3 | Printing a line of text with multiple statements.

Displaying Multiple Lines of Text with a Single Statement

A single statement can display multiple lines by using **newline characters** (`\n`), which indicate to `System.out`'s `print` and `println` methods when to position the output cursor at the beginning of the next line in the command window. Like blank lines, space characters and tab characters, newline characters are white space characters. The program in Fig. 2.4 outputs four lines of text, using newline characters to determine when to begin each new line. Most of the program is identical to those in Figs. 2.1 and 2.3.

```
1  // Fig. 2.4: Welcome3.java
2  // Printing multiple lines of text with a single statement.
3
4  public class Welcome3 {
5     // main method begins execution of Java application
6     public static void main(String[] args) {
7        System.out.println("Welcome\nto\nJava\nProgramming!");
8     } // end method main
9  } // end class Welcome3
```

```
Welcome
to
Java
Programming!
```

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

Line 7 displays four lines of text in the command window. Normally, the characters in a string are displayed *exactly* as they appear in the double quotes. However, the paired characters \ and `n` (repeated three times in the statement) do *not* appear on the screen. The **backslash** (\) is an **escape character**, which has special meaning to `System.out`'s `print` and `println` methods. When a backslash appears in a string, Java combines it with the next character to form an **escape sequence**—`\n` represents the newline character. When a newline character appears in a string being output with `System.out`, the newline character causes the screen's output cursor to move to the beginning of the next line in the command window.

Figure 2.5 lists several escape sequences and describes how they affect the display of characters in the command window. For the complete list of escape sequences, visit

```
http://docs.oracle.com/javase/specs/jls/se8/html/jls-3.html#jls-3.10.6
```

Escape sequence	Description
`\n`	Newline. Position the screen cursor at the beginning of the *next* line.
`\t`	Horizontal tab. Move the screen cursor to the next tab stop.
`\r`	Carriage return. Position the screen cursor at the beginning of the *current* line—do *not* advance to the next line. Any characters output after the carriage return *overwrite* the characters previously output on that line.
`\\`	Backslash. Used to print a backslash character.
`\"`	Double quote. Used to print a double-quote character. For example, `System.out.println("\"in quotes\"");` displays `"in quotes"`.

Fig. 2.5 | Some common escape sequences.

2.4 Displaying Text with `printf`

Method **`System.out.printf`** (`f` means "formatted") displays *formatted* data. Figure 2.6 uses this to output on two lines the strings `"Welcome to"` and `"Java Programming!"`.

```
1  // Fig. 2.6: Welcome4.java
2  // Displaying multiple lines with method System.out.printf.
3
4  public class Welcome4 {
5     // main method begins execution of Java application
6     public static void main(String[] args) {
7        System.out.printf("%s%n%s%n", "Welcome to", "Java Programming!");
8     } // end method main
9  } // end class Welcome4
```

```
Welcome to
Java Programming!
```

Fig. 2.6 | Displaying multiple lines with method `System.out.printf`.

Line 7 calls method `System.out.printf` to display the program's output. The method call specifies three arguments. When a method requires multiple arguments, they're placed in a **comma-separated list**.

Good Programming Practice 2.4

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

Method `printf`'s first argument is a **format string** that may consist of **fixed text** and **format specifiers**. Fixed text is output by `printf` just as it would be by `print` or `println`. Each format specifier is a *placeholder* for a value and specifies the *type of data* to output. Format specifiers also may include optional formatting information.

Format specifiers begin with a percent sign (%) followed by a character that represents the *data type*. For example, the format specifier **`%s`** is a placeholder for a string. The format string specifies that `printf` should output two strings, each followed by a newline character. At the first format specifier's position, `printf` substitutes the value of the first argument after the format string. At each subsequent format specifier's position, `printf` substitutes the value of the next argument. So this example substitutes `"Welcome to"` for the first `%s` and `"Java Programming!"` for the second `%s`. The output shows that two lines of text are displayed on two lines.

Instead of using the escape sequence `\n`, we used the **`%n`** format specifier, which is a line separator that's *portable* across operating systems. You cannot use `%n` in the argument to `System.out.print` or `System.out.println`; however, the line separator output by `System.out.println` *after* it displays its argument *is* portable across operating systems.

2.5 Another Application: Adding Integers

Our next application (Fig. 2.7) reads two **integers** typed by a user at the keyboard, computes their sum and displays it. In the sample output, we use bold text to identify the user's input (i.e., **45** and **72**).

```
1  // Fig. 2.7: Addition.java
2  // Addition program that inputs two numbers then displays their sum.
3  import java.util.Scanner; // program uses class Scanner
4
5  public class Addition {
6     // main method begins execution of Java application
7     public static void main(String[] args) {
8        // create a Scanner to obtain input from the command window
9        Scanner input = new Scanner(System.in);
10
11       System.out.print("Enter first integer: "); // prompt
12       int number1 = input.nextInt(); // read first number from user
13
14       System.out.print("Enter second integer: "); // prompt
15       int number2 = input.nextInt(); // read second number from user
16
17       int sum = number1 + number2; // add numbers, then store total in sum
18
19       System.out.printf("Sum is %d%n", sum); // display sum
20    } // end method main
21 } // end class Addition
```

```
Enter first integer: 45
Enter second integer: 72
Sum is 117
```

Fig. 2.7 | Addition program that inputs two numbers, then displays their sum.

2.5.1 `import` Declarations

A great strength of Java is its rich set of predefined classes that you can *reuse* rather than "reinventing the wheel." These classes are grouped into **packages**—*named groups of related classes*—and are collectively referred to as the **Java class library**, or the **Java Application Programming Interface** (**Java API**). Line 3 is an **`import` declaration** that helps the compiler locate a class that's used in this program. It indicates that the program uses the predefined `Scanner` class (discussed shortly) from the package named **`java.util`**. The compiler then ensures that you use the class correctly.

Common Programming Error 2.3

All `import` declarations must appear before the first class declaration in the file. Placing an `import` declaration inside or after a class declaration is a syntax error.

Common Programming Error 2.4

Forgetting to include an `import` declaration for a class that must be imported results in a compilation error containing a message such as "`cannot find symbol`." When this occurs, check that you provided the proper `import` declarations and that the names in them are correct, including proper capitalization.

2.5.2 Declaring and Creating a Scanner to Obtain User Input from the Keyboard

All Java variables *must* be declared with a **name** and a **type** *before* they can be used. A variable name can be any valid identifier. Like other statements, declaration statements end with a semicolon (`;`).

Line 9 of `main` is a **variable declaration statement** that specifies the *name* (`input`) and *type* (`Scanner`) of a variable that's used in this program. A **`Scanner`** (package `java.util`) enables a program to read data (e.g., numbers and strings) for use in a program. The data can come from many sources, such as the user at the keyboard or a file on disk. Before using a `Scanner`, you must create it and specify the *source* of the data.

The `=` in line 9 indicates that `Scanner` variable `input` should be initialized in its declaration with the result of the expression to the right of the equals sign—`new Scanner(System.in)`. This expression uses the **`new`** keyword to create a `Scanner` object that reads characters typed by the user at the keyboard. The **standard input object**, **`System.in`**, enables applications to read *bytes* of data typed by the user. The `Scanner` translates these bytes into types (like `int`s) that can be used in a program.

Good Programming Practice 2.5

By convention, variable-name identifiers use the camel-case naming convention with a lowercase first letter—for example, `firstNumber`.

2.5.3 Prompting the User for Input

Line 11 uses `System.out.print` to display the message `"Enter first integer: "`. This message is called a **prompt** because it directs the user to take a specific action. Recall from Section 2.2 that identifiers starting with capital letters typically represent class names. Class `System` is part of package **`java.lang`**.

Software Engineering Observation 2.1

By default, package `java.lang` *is imported in every Java program; thus, classes in* `java.lang` *are the only ones in the Java API that do not require an* `import` *declaration.*

2.5.4 Declaring a Variable to Store an Integer and Obtaining an Integer from the Keyboard

The variable declaration statement in line 12 declares that variable `number1` holds data of type **`int`**—that is, *integer* values, The range of values for an `int` is –2,147,483,648 to +2,147,483,647. The `int` values you use in a program may not contain commas; however, for readability, you can place underscores in numbers. So `60_000_000` represents the `int` value 60,000,000.

Some other types of data are **`float`** and **`double`**, for holding real numbers, and **`char`**, for holding character data. Variables of type `char` represent individual characters, such as an uppercase letter (e.g., `A`), a digit (e.g., `7`), a special character (e.g., `*` or `%`) or an escape sequence (e.g., the tab character, `\t`). The types `int`, `float`, `double` and `char` are called **primitive types**. Primitive-type names are keywords and must appear in all lowercase letters. Appendix D summarizes the characteristics of the eight primitive types (`boolean`, `byte`, `char`, `short`, `int`, `long`, `float` and `double`).

The `=` in line 12 initializes the `int` variable `number1` with the result of the expression `input.nextInt()`. This uses the `Scanner` object `input`'s `nextInt` method to obtain an integer from the user at the keyboard. At this point the program *waits* for the user to type the number and press the *Enter* key to submit the number to the program.

Our program assumes that the user enters a valid integer value. If not, a logic error will occur and the program will terminate. Chapter 11, Exception Handling: A Deeper Look, discusses how to make your programs more robust by enabling them to handle such errors. This is also known as making your programs *fault tolerant*.

2.5.5 Obtaining a Second Integer

Line 14 prompts the user to enter the second integer. Line 15 declares the `int` variable `number2` and initializes it with a second integer read from the user at the keyboard.

2.5.6 Using Variables in a Calculation

Line 17 declares the `int` variable `sum` and initializes it with the result of `number1 + number2`. In the preceding statement, the addition operator is a **binary operator**. Portions of statements that contain calculations are called **expressions**. An expression is any portion of a statement that has a *value*. The value of the expression `number1 + number2` is the *sum* of the numbers. Similarly, the value of the expression `input.nextInt()` (lines 12 and 15) is the integer typed by the user.

2.5.7 Displaying the Calculation Result

After the calculation has been performed, line 19 uses method `System.out.printf` to display the `sum`. The format specifier **`%d`** is a *placeholder* for an `int` value (in this case the value of `sum`)—the letter `d` stands for "decimal integer." The remaining characters in the format string are all fixed text. So, method `printf` displays `"Sum is "`, followed by the value of `sum` (in the position of the `%d` format specifier) and a newline.

Calculations also can be performed *inside* `printf` statements. We could have combined the statements at lines 17 and 19 into one statement by replacing sum in line 19 with `number1 + number2`.

2.5.8 Java API Documentation

For each new Java API class we use, we indicate the package in which it's located. This information helps you locate descriptions of each package and class in the Java API documentation. A web-based version of this documentation can be found at

```
http://docs.oracle.com/javase/8/docs/api/index.html
```

You can download it from the Additional Resources section at

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

2.5.9 Declaring and Initializing Variables in Separate Statements

Each variable must have a value *before* you can use the variable in a calculation (or other expression). The variable declaration statement in line 12 both declared `number1` *and* initialized it with a value entered by the user.

Sometimes you declare a variable in one statement, then initialize it in another. For example, line 12 could have been written in two statements as

```
int number1; // declare the int variable number1
number1 = input.nextInt(); // assign the user's input to number1
```

The first statement declares `number1`, but does *not* initialize it. The second statement uses the **assignment operator**, `=`, to assign `number1` the value entered by the user. Everything to the *right* of the assignment operator, `=`, is always evaluated *before* the assignment is performed.

2.6 Arithmetic

The **arithmetic operators** are summarized in Fig. 2.8. The **asterisk** (`*`) indicates multiplication, and the percent sign (`%`) is the **remainder operator**, which we'll discuss shortly. The arithmetic operators in Fig. 2.8 are *binary* operators.

Java operation	Operator	Algebraic expression	Java expression
Addition	`+`	$f + 7$	`f + 7`
Subtraction	`-`	$p - c$	`p - c`
Multiplication	`*`	bm	`b * m`
Division	`/`	x / y or $\frac{x}{y}$ or $x \div y$	`x / y`
Remainder	`%`	$r \bmod s$	`r % s`

Fig. 2.8 | Arithmetic operators.

Integer division yields an integer quotient. For example, the expression `7 / 4` evaluates to `1`, and the expression `17 / 5` evaluates to `3`. Any fractional part in integer division is

simply *truncated*—no *rounding* occurs. Java provides the remainder operator, `%`, which yields the remainder after division. The expression `x % y` yields the remainder after `x` is divided by `y`. Thus, `7 % 4` yields `3`, and `17 % 5` yields `2`. This operator is most commonly used with integer operands but it can also be used with other arithmetic types.

Rules of Operator Precedence

Java applies the arithmetic operators in a precise sequence determined by the **rules of operator precedence**, which are generally the same as those followed in algebra:

1. Multiplication, division and remainder operations are applied first. If an expression contains several such operations, they're applied from left to right. Multiplication, division and remainder operators have the same level of precedence.
2. Addition and subtraction operations are applied next. If an expression contains several such operations, the operators are applied from left to right. Addition and subtraction operators have the same level of precedence.

These rules enable Java to apply operators in the correct *order*.[1] When we say that operators are applied from left to right, we're referring to their **associativity**. Some associate from right to left. Figure 2.9 summarizes these rules of operator precedence. A complete precedence chart is included in Appendix A.

Operator(s)	Operation(s)	Order of evaluation (precedence)

`*`
`/`
`%`
	Multiplication
	Division
	Remainder
		Evaluated first. If there are several operators of this type, they're evaluated from *left to right*.

`+` `-`	Addition Subtraction	Evaluated next. If there are several operators of this type, they're evaluated from *left to right*.
`=`	Assignment	Evaluated last.

Fig. 2.9 | Precedence of arithmetic operators.

2.7 Decision Making: Equality and Relational Operators

A **condition** is an expression that can be **`true`** or **`false`**. This section introduces Java's **`if` selection statement**, which allows a program to make a **decision** based on a condition's value. If an `if` statement's condition is *true*, its body executes. If the condition is *false*, its body does not execute.

Conditions in `if` statements can be formed by using the **equality operators** (`==` and `!=`) and **relational operators** (`>`, `<`, `>=` and `<=`) summarized in Fig. 2.10. Both equality operators have the same level of precedence, which is *lower* than that of the relational operators. The equality operators associate from *left to right*. The relational operators all have the same level of precedence and also associate from *left to right*.

1. Subtle order-of-evaluation issues can occur in expressions. For more information, see Chapter 15 of *The Java® Language Specification* (`https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html`).

Algebraic operator	Java equality or relational operator	Sample Java condition	Meaning of Java condition
Equality operators			
=	==	x == y	x is equal to y
≠	!=	x != y	x is not equal to y
Relational operators			
>	>	x > y	x is greater than y
<	<	x < y	x is less than y
≥	>=	x >= y	x is greater than or equal to y
≤	<=	x <= y	x is less than or equal to y

Fig. 2.10 | Equality and relational operators.

Figure 2.11 uses six `if` statements to compare two integers input by the user. If the condition in any of these `if` statements is *true*, the statement associated with that `if` statement executes; otherwise, the statement is skipped. We use a `Scanner` to input the integers from the user and store them in variables `number1` and `number2`. The program *compares* the numbers and displays the results of the comparisons that are true.

```
1  // Fig. 2.11: Comparison.java
2  // Compare integers using if statements, relational operators
3  // and equality operators.
4  import java.util.Scanner; // program uses class Scanner
5
6  public class Comparison {
7     // main method begins execution of Java application
8     public static void main(String[] args) {
9        // create Scanner to obtain input from command line
10       Scanner input = new Scanner(System.in);
11
12       System.out.print("Enter first integer: "); // prompt
13       int number1 = input.nextInt(); // read first number from user
14
15       System.out.print("Enter second integer: "); // prompt
16       int number2 = input.nextInt(); // read second number from user
17
18       if (number1 == number2)
19          System.out.printf("%d == %d%n", number1, number2);
20       }
21
22       if (number1 != number2) {
23          System.out.printf("%d != %d%n", number1, number2);
24       }
25
```

Fig. 2.11 | Compare integers using `if` statements, relational operators and equality operators. (Part 1 of 2.)

```
26          if (number1 < number2) {
27             System.out.printf("%d < %d%n", number1, number2);
28          }
29
30          if (number1 > number2) {
31             System.out.printf("%d > %d%n", number1, number2);
32          }
33
34          if (number1 <= number2) {
35             System.out.printf("%d <= %d%n", number1, number2);
36          }
37
38          if (number1 >= number2) {
39             System.out.printf("%d >= %d%n", number1, number2);
40          }
41       } // end method main
42    } // end class Comparison
```

```
Enter first integer: 777
Enter second integer: 777
777 == 777
777 <= 777
777 >= 777
```

```
Enter first integer: 1000
Enter second integer: 2000
1000 != 2000
1000 < 2000
1000 <= 2000
```

```
Enter first integer: 2000
Enter second integer: 1000
2000 != 1000
2000 > 1000
2000 >= 1000
```

Fig. 2.11 | Compare integers using `if` statements, relational operators and equality operators. (Part 2 of 2.)

Class `Comparison`'s `main` method (lines 8–41) begins the execution of the program. Line 10 declares `Scanner` variable `input` and assigns it a `Scanner` that inputs data from the standard input (i.e., the keyboard). Lines 12–16 prompt for and read the user's input.

Lines 18–20 compare the values of variables `number1` and `number2` to test for equality. If the values are equal, the statement in line 19 displays a line of text indicating that the numbers are equal. The `if` statements starting in lines 22, 26, 30, 34 and 38 compare `number1` and `number2` using the operators `!=`, `<`, `>`, `<=` and `>=`, respectively. If the conditions are `true` in one or more of those `if` statements, the corresponding body statement displays an appropriate line of text.

Each `if` statement in Fig. 2.11 contains a single body statement that's indented. Also notice that we've enclosed each body statement in a pair of braces, `{ }`, creating a **compound statement** or a **block**.

Common Programming Error 2.5

Placing a semicolon immediately after the right parenthesis after the condition in an `if` statement is often a logic error (although not a syntax error). The semicolon causes the body of the `if` statement to be empty, so the `if` statement performs no action, regardless of whether or not its condition is true. Worse yet, the original body statement of the `if` statement always *executes, often causing the program to produce incorrect results.*

Operators Discussed So Far

Figure 2.12 shows the operators discussed so far in decreasing order of precedence. All but the assignment operator, `=`, associate from *left to right*. The assignment operator, `=`, associates from *right to left*. An assignment expression's value is whatever was assigned to the variable on the `=` operator's left side—for example, the value of the expression `x = 7` is `7`. So an expression like `x = y = 0` is evaluated as if it had been written as `x = (y = 0)`, which first assigns the value `0` to variable `y`, then assigns the result of that assignment, `0`, to `x`.

Operators				Associativity	Type
*	/	%		left to right	multiplicative
+	-			left to right	additive
<	<=	>	>=	left to right	relational
==	!=			left to right	equality
=				right to left	assignment

Fig. 2.12 | Precedence and associativity of operators discussed.

Good Programming Practice 2.6

When writing expressions containing many operators, refer to the operator precedence chart (Appendix A). Confirm that the operations in the expression are performed in the order you expect. If, in a complex expression, you're uncertain about the order of evaluation, use parentheses to force the order, exactly as you'd do in algebraic expressions.

2.8 Wrap-Up

In this chapter, you learned many important features of Java, including displaying data on the screen in a command window, inputting data from the keyboard, performing calculations and making decisions. As you'll see in Chapter 3, Java applications typically contain just a few lines of code in method `main`—these statements normally create the objects that perform the work of the application. In Chapter 3, you'll implement your own classes and use objects of those classes in applications.

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 what primitive types and reference types are.
- Use a constructor to initialize an object's data.
- Represent and use numbers containing decimal points.

Outline

3.1 Introduction[1]

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. 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—this is one of the great strengths of Java as an object-oriented programming language.

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 and you can add the `withdraw` method on your own as an exercise.

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. [Alternatively, you could treat monetary amounts as whole numbers of pennies, then break the result into dollars and cents by using division and remainder operations, respectively, and insert a period between the dollars and the cents.]

1. This chapter depends on the review of terminology and concepts of object-oriented programming in Section 1.2.

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, and *set* and *get* Methods

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     private String name; // instance variable
7
8     // method to set the name in the object
9     public void setName(String name) {
10       this.name = name; // store the name
11    }
12
13    // method to retrieve the name from the object
14    public String getName() {
15       return name; // return value of name to caller
16    }
17 }
```

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:

```
public class Account {
```

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 5 and 17 of Fig. 3.1.

Identifiers and Camel-Case Naming

Recall from Chapter 2 that class names, method names and variable names are all *identifiers* and by convention all use the *camel-case* naming scheme. 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 from Section 1.2 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 6

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

declares instance variable `name` of type `String` *outside* the bodies of methods `setName` (lines 9–11) and `getName` (lines 14–16). `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 6). Like `public`, `private` is an *access modifier*. Variables or methods declared with `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 9–11):

```
public void setName(String name) {
   this.name = name; // store the name
}
```

We refer to the first line of each method declaration (line 9 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. As you'll soon see, the statement in line 19 of `main` (Fig. 3.2) *calls* method `setName`, so `main` is `setName`'s *caller* in this example. The return type `void` (line 9 in Fig. 3.1) 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 see shortly, `Account` method `getName` returns a value.

Method `setName` receives *parameter* `name` of type `String`. Parameters are declared in a 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 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 9 and 11 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 10) 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 9 and 6, 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 10. After line 10 executes, the method has completed its task, so it returns to its *caller*.

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 9, but using the `this` keyword as shown in line 10 is a widely accepted practice to minimize the proliferation of identifier names.

`getName` *Method of Class* `Account`

Method `getName` (lines 14–16)

```
public String getName() {
   return name; // return value of name to 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 14 and 24 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 15 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 23–24 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`

A person drives a car by telling it what to do (go faster, go slower, turn left, turn right, etc.)—without having to know how the car's internal mechanisms work. Similarly, a method (such as `main`) "drives" an `Account` object by calling its methods—without having to know how the class's internal mechanisms work. In this sense, the class containing method `main` is referred to as a **driver class**.

To help you prepare for the larger programs you'll encounter later in this book and in industry, we define class `AccountTest` and its `main` method in the file `AccountTest.java` (Fig. 3.2). 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.

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

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 8 creates a `Scanner` object named `input` for inputting a name from the user. Line 17 prompts the user to enter a name. Line 18 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 11 creates an `Account` object and assigns it to variable `myAccount` of type `Account`. The variable is initialized with the result of `new Account()`—a **class instance creation expression**. Keyword **`new`** creates a new object of the specified class—in this case, `Account`. The parentheses are *required*. As you'll learn in Section 3.3, 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 3.3, 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 8 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 14 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 14 calls `getName` using the `myAccount` object created in line 11, 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 14 in `main`) to method `getName`'s declaration (lines 14–16 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 15 of Fig. 3.1). When the `return` statement executes, program execution continues where `getName` was called (line 14 in Fig. 3.2).
3. `System.out.printf` displays the `String` returned by method `getName`, then the program continues executing at line 17 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 3.5 when we consider *reference types*.

Calling Class `Account`'s `setName` Method

Line 19 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 19 in `main` to `setName` method's declaration (lines 9–11 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 9 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 10 of Fig. 3.1).
3. When program execution reaches `setName`'s closing right brace, it returns to where `setName` was called (line 19 of Fig. 3.2), then continues at line 20 of Fig. 3.2.

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 9 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 20 of Fig. 3.2 outputs a blank line. When the second call to method `getName` (line 24) executes, the name entered by the user in line 18 is displayed. After the statement at lines 23–24 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 *all* files in the *current* directory ending with the file-name 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

We'll often use UML class diagrams to help you visualize 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.

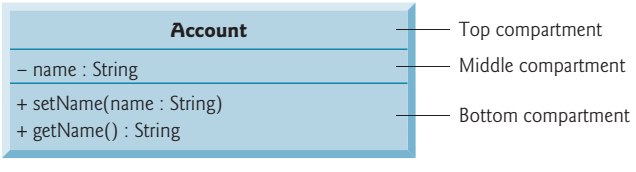


Fig. 3.3 | UML class diagram for class `Account` of Fig. 3.1.

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.

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 Java methods. 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 14 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 `main` method. Recall that `main` is *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. In Chapter 6, you'll learn that method `toString` is commonly invoked *implicitly*.

Lines 6–25 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 6), 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 8 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

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 3.4.

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 *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.

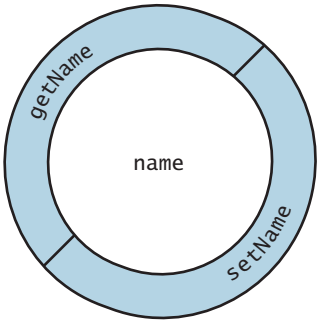


Fig. 3.4 | Conceptual view of an `Account` object with its encapsulated `private` instance variable `name` and protective layer of `public` methods.

3.3 `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).

```
1  // Fig. 3.5: Account.java
2  // Account class with a constructor that initializes the name.
3
4  public class Account {
5     private String name; // instance variable
6
7     // constructor initializes name with parameter name
8     public Account(String name) { // constructor name is class name
9        this.name = name;
10    }
11
```

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

```
12      // method to set the name
13      public void setName(String name) {
14         this.name = name;
15      }
16
17      // method to retrieve the name
18      public String getName() {
19         return name;
20      }
21   }
```

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

3.3.1 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 you'll see in line 8 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.

`Account` Constructor Declaration

Lines 8–10 of Fig. 3.5 declare `Account`'s constructor, which *must* have the *same name* as the class. A constructor's *parameter list* specifies that the constructor requires zero or more pieces of data to perform its task. Line 8 indicates that the constructor has exactly one parameter—a `String` called `name`. When you create a new `Account` object, you'll pass a person's name to the constructor's `name` parameter. The constructor will then assign the `name` parameter's value to the *instance variable* `name` (line 9).

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 8 is a local variable of the constructor that's *not* visible to method `setName`, and the one in line 13 is a local variable of `setName` that's *not* visible to the constructor.

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

The `AccountTest` program (Fig. 3.6) initializes two `Account` objects using the constructor. Line 8 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 constructor 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. Line 8 assigns the new object to the variable `account1`. Line 9 repeats this process, passing the argument `"John Blue"` to initialize the name for `account2`. Lines 12–13 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 the 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     public static void main(String[] args) {
7        // create two Account objects
8        Account account1 = new Account("Jane Green");
9        Account account2 = new Account("John Blue");
10
11       // display initial value of name for each Account
12       System.out.printf("account1 name is: %s%n", account1.getName());
13       System.out.printf("account2 name is: %s%n", account2.getName());
14    }
15 }
```

```
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 11 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 in-

stance 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. As with 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.

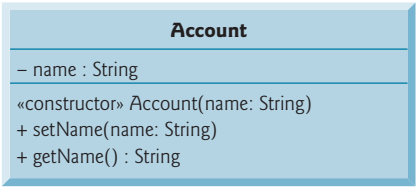


Fig. 3.7 | UML class diagram for `Account` class of Fig. 3.5.

3.4 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.

3.4.1 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 7 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     private String name; // instance variable
7     private double balance; // instance variable
8
9     // Account constructor that receives two parameters
10    public Account(String name, double balance) {
11       this.name = name; // assign name to instance variable name
12
13       // validate that the balance is greater than 0.0; if it's not,
14       // instance variable balance keeps its default initial value of 0.0
15       if (balance > 0.0) { // if the balance is valid
16          this.balance = balance; // assign it to instance variable balance
17       }
18    }
19
20    // method that deposits (adds) only a valid amount to the balance
21    public void deposit(double depositAmount) {
22       if (depositAmount > 0.0) { // if the depositAmount is valid
23          balance = balance + depositAmount; // add it to the balance
24       }
25    }
26
27    // method returns the account balance
28    public double getBalance() {
29       return balance;
30    }
31
32    // method that sets the name
33    public void setName(String name) {
34       this.name = name;
35    }
36
37    // method that returns the name
38    public String getName() {
39       return name;
40    }
41 }
```

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

`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 10–18) now receives a second parameter—`balance` of type `double` that represents the *starting balance*. Lines 15–17 ensure

that initialBalance is greater than 0.0. If so, the balance parameter's value is assigned to the instance variable balance. Otherwise, the instance variable balance remains at 0.0—its *default initial value*.

Account Class deposit Method

Method deposit (lines 21–25) 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 instance variable balance *only* if the parameter value is *valid* (i.e., greater than zero). Line 23 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 23 does *not* modify the balance—that's why the assignment is necessary.

Account Class getBalance Method

Method getBalance (lines 28–30) 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, lines 15, 16, 23 and 29 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.

3.4.2 AccountTest Class to Use Class Account

Class AccountTest (Fig. 3.9) creates two Account objects (lines 7–8) 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 11–14 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;
4
5  public class AccountTest {
6     public static void main(String[] args) {
7        Account account1 = new Account("Jane Green", 50.00);
8        Account account2 = new Account("John Blue", -7.53);
9
10       // display initial balance of each object
11       System.out.printf("%s balance: $%.2f%n",
12          account1.getName(), account1.getBalance());
13       System.out.printf("%s balance: $%.2f%n%n",
14          account2.getName(), account2.getBalance());
15
```

Fig. 3.9 | Inputting and outputting floating-point numbers with Account objects. (Part 1 of 2.)

```
16         // create a Scanner to obtain input from the command window
17         Scanner input = new Scanner(System.in);
18
19         System.out.print("Enter deposit amount for account1: "); // prompt
20         double depositAmount = input.nextDouble(); // obtain user input
21         System.out.printf("%nadding %.2f to account1 balance%n%n",
22            depositAmount);
23         account1.deposit(depositAmount); // add to account1's balance
24
25         // display balances
26         System.out.printf("%s balance: $%.2f%n",
27            account1.getName(), account1.getBalance());
28         System.out.printf("%s balance: $%.2f%n%n",
29            account2.getName(), account2.getBalance());
30
31         System.out.print("Enter deposit amount for account2: "); // prompt
32         depositAmount = input.nextDouble(); // obtain user input
33         System.out.printf("%nadding %.2f to account2 balance%n%n",
34            depositAmount);
35         account2.deposit(depositAmount); // add to account2 balance
36
37         // display balances
38         System.out.printf("%s balance: $%.2f%n",
39            account1.getName(), account1.getBalance());
40         System.out.printf("%s balance: $%.2f%n%n",
41            account2.getName(), account2.getBalance());
42      }
43   }
```

```
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

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 2 of 2.)

Displaying the `Account` *Objects' Initial Balances*

When method `getBalance` is called for `account1` from line 12, the value of `account1`'s `balance` is returned from line 29 of Fig. 3.8 and displayed by the `System.out.printf` statement (Fig. 3.9, lines 11–12). Similarly, when method `getBalance` is called for `account2` from line 14, the value of the `account2`'s `balance` is returned from line 29 of Fig. 3.8 and displayed by the `System.out.printf` statement (Fig. 3.9, lines 13–14). 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 balances 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 19 (Fig. 3.9) prompts the user to enter a deposit amount for account1. Line 20 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.

Error-Prevention Tip 3.3

The Java compiler issues 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 20 obtains the input from the user by calling Scanner object input's **nextDouble** method, which returns a double value entered by the user. Lines 21–22 display the depositAmount. Line 23 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 21 of Fig. 3.8); then method deposit adds that value to the balance. Lines 26–29 (Fig. 3.9) output the names and balances of both Accounts *again* to show that *only* account1's balance has changed.

Line 31 prompts the user to enter a deposit amount for account2. Line 32 obtains the input from the user by calling Scanner object input's nextDouble method. Lines 33–34 display the depositAmount. Line 35 calls object account2's deposit method with depositAmount as the method's *argument*; then method deposit adds that value to the balance. Finally, lines 38–41 output the names and balances of both Accounts *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.

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.

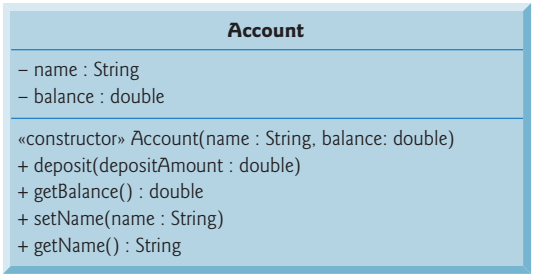


Fig. 3.10 | UML class diagram for `Account` class of Fig. 3.8.

3.5 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`—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 *locations* of objects. Such a variable is said to **refer to an object** in the program. *Objects* that are referenced may each contain *many* instance variables. Line 8 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 11 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 14 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 14 and 24) and `setName` (line 19) to interact with the `Account` object. Primitive-type variables do *not* refer to objects, so such variables *cannot* be used to call methods.

3.6 Wrap-Up

In this chapter, you learned how to create your own 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 a method to tell it to perform its task, passed information to a method as arguments whose values are assigned to the method's parameters and received the value returned by a method. You saw 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 used 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 visually the methods, attributes and constructors of classes. Finally, you used floating-point numbers (numbers with decimal points). [In Chapter 8, we'll begin representing monetary amounts precisely with class `BigDecimal`.] In the next chapter we introduce control statements, which specify the order in which a program's actions are performed.

4

Control Statements: Part 1; Assignment, ++ and -- Operators

Objectives

In this chapter you'll:

- Use the `if` and `if...else` selection statements to choose between alternative actions.
- Use the `while` iteration statement to execute statements in a program repeatedly.
- Use counter-controlled iteration and sentinel-controlled iteration.
- Use the compound assignment operator and the increment and decrement operators.
- See that the primitive data types are portable.

Outline

4.1 Introduction

In this chapter, we discuss Java's `if` statement in additional detail and introduce the `if`...`else` and `while` statements. We present the compound assignment operator and the increment and decrement operators. Finally, we consider the portability of Java's primitive types.

4.2 Control Structures

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The blame was pointed at the **goto statement** (used in most programming languages of the time), which allows you to specify a transfer of control to one of a wide range of destinations in a program. The term **structured programming** became almost synonymous with "`goto` elimination." [*Note:* Java does *not* have a `goto` statement; however, the word `goto` is *reserved* by Java and should *not* be used as an identifier in programs.]

Bohm and Jacopini's work demonstrated that all programs could be written in terms of only three control structures—the **sequence structure**, the **selection structure** and the **repetition structure**.[1] When we introduce Java's control-structure implementations, we'll refer to them in the terminology of the *Java Language Specification* as "control statements."

4.2.1 Sequence Structure in Java

The sequence structure is built into Java. Unless directed otherwise, the computer executes Java statements one after the other in the order in which they're written—that is, in sequence. The UML **activity diagram** in Fig. 4.1 illustrates a typical sequence structure in which two calculations are performed in order. Java lets you have as many actions as you want in sequence. As we'll soon see, anywhere a single action may be placed, we may place several actions in sequence.

1. C. Bohm, and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371.

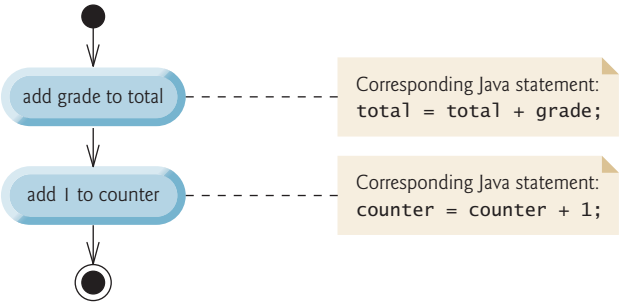


Fig. 4.1 | Sequence-structure activity diagram.

A UML activity diagram models the **workflow** (also called the **activity**) of a portion of a software system. Such workflows may include a portion of an algorithm, like the sequence structure in Fig. 4.1. Activity diagrams are composed of symbols, such as **action-state symbols** (rectangles with their left and right sides replaced with outward arcs), **diamonds** and **small circles**. These symbols are connected by **transition arrows**, which represent the *flow of the activity*—that is, the *order* in which the actions should occur. We use the UML in this chapter and Chapter 5 to show control flow in control statements.

Consider the activity diagram in Fig. 4.1. It contains two **action states**, each containing an **action expression**—"add grade to total" or "add 1 to counter"—that specifies a particular action to perform. Other actions might include calculations or input/output operations. The arrows represent **transitions**, which indicate the order in which the actions represented by the action states occur. The program that implements the activities illustrated by the diagram in Fig. 4.1 first adds `grade` to `total`, then adds `1` to `counter`.

The **solid circle** at the top of the activity diagram represents the **initial state**—the *beginning* of the workflow *before* the program performs the modeled actions. The **solid circle surrounded by a hollow circle** at the bottom of the diagram represents the **final state**—the *end* of the workflow *after* the program performs its actions.

Figure 4.1 also includes rectangles with the upper-right corners folded over. These are UML **notes** (like comments in Java)—explanatory remarks that describe the purpose of symbols in the diagram. Figure 4.1 uses notes to show the Java code associated with each action state. A **dotted line** connects each note with the element it describes. Activity diagrams normally do *not* show the corresponding Java code. We do this here to illustrate how the diagram relates to Java code. For more information on the UML, see our object-oriented design case study (Chapters 25–26) or visit `http://www.uml.org`.

4.2.2 Selection Statements in Java

Java has three types of **selection statements** (discussed in this chapter and Chapter 5). The *`if` statement* either performs (selects) an action, if a condition is *true*, or skips it, if the condition is *false*. The *`if...else` statement* performs an action if a condition is *true* and performs a different action if the condition is *false*. The *`switch` statement* (Chapter 5) performs one of *many* different actions, depending on the value of an expression.

The `if` statement is a **single-selection statement** because it selects or ignores a *single* action (or, as we'll soon see, a *single group of actions*). The `if...else` statement is called a **double-selection statement** because it selects between *two different actions* (or *groups of actions*). The `switch` statement is called a **multiple-selection statement** because it selects among *many different actions* (or *groups of actions*).

4.2.3 Iteration Statements in Java

Java provides four **iteration statements** (also called **repetition statements** or **looping statements**) that enable programs to perform statements repeatedly as long as a condition (called the **loop-continuation condition**) remains *true*. The iteration statements are `while`, `do...while`, `for` and enhanced `for`. (Chapter 5 presents the `do...while` and `for` statements and Chapter 7 presents the enhanced `for` statement.) The `while` and `for` statements perform the action (or group of actions) in their bodies zero or more times—if the loop-continuation condition is initially *false*, the action (or group of actions) will *not* execute. The `do...while` statement performs the action (or group of actions) in its body *one or more* times. The words `if`, `else`, `switch`, `while`, `do` and `for` are Java keywords. A complete list of Java keywords appears in Appendix C.

4.2.4 Summary of Control Statements in Java

Java has only three kinds of control structures, which from this point forward we refer to as *control statements*: the *sequence statement*, *selection statements* (three types) and *iteration statements* (four types). Every program is formed by combining as many of these statements as is appropriate for the algorithm the program implements. We can model each control statement as an activity diagram. Like Fig. 4.1, each diagram contains an initial state and a final state that represent a control statement's entry point and exit point, respectively. **Single-entry/single-exit control statements** make it easy to build programs—we simply connect the exit point of one to the entry point of the next. We call this **control-statement stacking**. There's only one other way in which control statements may be connected—**control-statement nesting**—in which one control statement appears *inside* another. Thus, algorithms in Java programs are constructed from only three kinds of control statements, combined in only two ways. This is the essence of simplicity.

4.3 `if` Single-Selection Statement

Programs use selection statements to choose among alternative courses of action. For example, suppose that the passing grade on an exam is 60. The statement

```
if (studentGrade >= 60) {
   System.out.println("Passed");
}
```

determines whether the *condition* `studentGrade >= 60` is *true*. If so, "`Passed`" is printed, and the next statement in order is performed. If the condition is *false*, the printing statement is ignored, and the next statement in order is performed. The indentation of the second line of this selection statement is optional, but recommended, because it emphasizes the inherent structure of structured programs.

UML Activity Diagram for an `if` Statement

Figure 4.2 illustrates the single-selection `if` statement. This figure contains the most important symbol in an activity diagram—the *diamond*, or **decision symbol**, which indicates that a *decision* is to be made. The workflow continues along a path determined by the symbol's associated **guard conditions**, which can be *true* or *false*. Each transition arrow emerging from a decision symbol has a guard condition (specified in square brackets next to the arrow). If a guard condition is *true*, the workflow enters the action state to which the transition arrow points. In Fig. 4.2, if the grade is greater than or equal to 60, the program prints `"Passed"` then transitions to the activity's final state. If the grade is less than 60, the program immediately transitions to the final state without displaying a message.

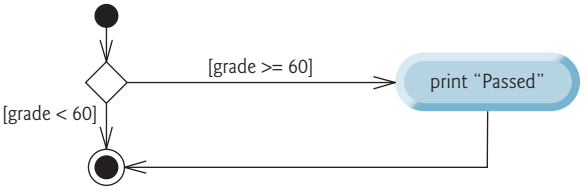


Fig. 4.2 | `if` single-selection statement UML activity diagram.

The `if` statement is a single-entry/single-exit control statement. The activity diagrams for the remaining control statements also contain initial states, transition arrows, action states that indicate actions to perform, decision symbols (with associated guard conditions) that indicate decisions to be made, and final states.

4.4 `if...else` Double-Selection Statement

The `if` single-selection statement performs an indicated action only when the condition is `true`; otherwise, the action is skipped. The **`if...else` double-selection statement** allows you to specify an action to perform when the condition is *true* and another action when the condition is *false*. For example, the statement

```
if (grade >= 60) {
   System.out.println("Passed");
}
else {
   System.out.println("Failed");
}
```

prints "`Passed`" if the student's grade is greater than or equal to 60, but prints "`Failed`" if it's less than 60. In either case, after printing occurs, the next statement in sequence is performed.

UML Activity Diagram for an `if...else` Statement

Figure 4.3 illustrates the flow of control in the `if...else` statement. Once again, the symbols in the UML activity diagram (besides the initial state, transition arrows and final state) represent action states and decisions.