GLOBAL EDITION

Java™ How to Program

Late Objects

ELEVENTH EDITION

Paul Deitel • Harvey Deitel

DIGITAL RESOURCES FOR STUDENTS

Your new textbook provides 12-month access to digital resources that may include VideoNotes (step-by-step video tutorials on programming concepts), source code, web chapters, quizzes, and more. Refer to the preface in the textbook for a detailed list of resources.

Follow the instructions below to register for the Companion Website for Paul Deitel and Harvey Deitel's ***Java™ How to Program, Late Objects, Eleventh Edition, Global Edition.***

1. Go to **www.pearsonglobaleditions.com/deitel.**
2. Enter the title of your textbook or browse by author name.
3. Click Companion Website.
4. Click Register and follow the on-screen instructions to create a login name and password.

ISSLDO-WHIFF-SAURY-LAMBS-DOLBY-LIKES

Use the login name and password you created during registration to start using the digital resources that accompany your textbook.

IMPORTANT

This prepaid subscription does not include access to MyProgrammingLab, which is available at **www.myprogramminglab.com** for purchase.

This access code can only be used once. This subscription is valid for 12 months upon activation and is not transferable.

For technical support go to **https://support.pearson.com/getsupport**

Java™

HOW TO PROGRAM

LATE OBJECTS

ELEVENTH EDITION
GLOBAL EDITION

Introducing
JShell

Use with
Java™ SE 8
or Java™ SE 9

Java™

HOW TO PROGRAM

LATE OBJECTS

ELEVENTH EDITION
GLOBAL EDITION

Introducing JShell

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

Use with
Java™ SE 8
or Java™ SE 9

Pearson

330 Hudson Street, NY, NY, 10013

Senior Vice President Courseware Portfolio Management: ***Marcia J. Horton***
Director, Portfolio Management: Engineering, Computer Science & Global Editions: ***Julian Partridge***
Higher Ed Portfolio Management: ***Tracy Johnson (Dunkelberger)***
Portfolio Management Assistant: ***Kristy Alaura***
Acquisitions Editor, Global Edition: ***Aditee Agarwal***
Managing Content Producer: ***Scott Disanno***
Content Producer: ***Robert Engelhardt***
Senior Project Editor, Global Edition: ***K.K. Neelakantan***
Web Developer: ***Steve Wright***
Rights and Permissions Manager: ***Ben Ferrini***
Manufacturing Buyer, Higher Ed, Lake Side Communications Inc (LSC): ***Maura Zaldivar-Garcia***
Senior Manufacturing Controller, Global Edition: ***Kay Holman***
Inventory Manager: ***Ann Lam***
Product Marketing Manager: ***Yvonne Vannatta***
Field Marketing Manager: ***Demetrius Hall***
Marketing Assistant: ***Jon Bryant***
Manager, Media Production, Global Edition: ***Vikram Kumar***
Cover Designer: ***Lumina Datamatics, Inc.***
Cover Art: ***©MchlSkhrv/Shutterstock***

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on page 6.

Pearson Education Limited
KAO Two
KAO Park
Harlow
CM17 9SR
United Kingdom

and Associated Companies throughout the world

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

Authorized adaptation from the United States edition, entitled Java How to Program, Late Objects, *11th Edition, ISBN 978-0-13-479140-1 by Paul Deitel and Harvey Deitel published by Pearson Education © 2020.*

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library

ISBN 10: 1-292-27373-9
ISBN 13: 978-1-292-27373-0
eBook ISBN 13: 978-1-292-27374-7

eBook formatted by GEX Inc.

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

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

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.

UNIX is a registered trademark of The Open Group.

Apache is a trademark of The Apache Software Foundation.

CSS and XML are registered trademarks of the World Wide Web Consortium.

Firefox is a registered trademark of the Mozilla Foundation.

Google is a trademark of Google, Inc.

Mac and macOS are trademarks of Apple Inc., registered in the U.S. and other countries.

Linux is a registered trademark of Linus Torvalds. All trademarks are property of their respective owners.

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

The online chapters and appendices listed at the end of this Table of Contents are located on the book's Companion Website (`http://www.pearsonglobaleditions.com`)—see the inside front cover of your book for details.

Online Chapters and Appendices

The online chapters and appendices are located on the book's Companion Website. See the book's inside front cover for details.

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 college textbooks, professional books, videos and corporate training. Many Java User Groups have joined together around the Deitels' publications, which are used internationally in university courses and professional training programs. 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 talks for university students and at events with Java professionals. Students 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, especially for those who take the time to learn, practice and master software development. The world needs good, focused expert developers.

So, how do you do it? First, let's be clear: Software development is hard. But do not be discouraged. Mastering 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.

Software development is an amazing skill. It can take you anywhere. You can work in any field. From nonprofits making the world a better place, to bleeding-edge biological technologies. From the frenetic daily run of the financial world to the deep mysteries of religion. From sports to music to acting. Everything has software. The success or failure of initiatives everywhere will depend on developers' knowledge and skills.

The push for you to get the relevant skills is what makes *Java How to Program, 11/e* so compelling. Written for students and new developers, it's easy to follow. It's written by authors who are educators and developers, with input over the years from some of the world's leading academics and 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.

How can this book help you become an expert?

Java was released in 1995—Paul and Harvey had the first edition of *Java How to Program* ready for Fall 1996 classes. Since that groundbreaking book, they've produced ten more editions, 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. Put in your full attention, and consciously "beat" each chapter. You'll soon find

yourself moving nicely along your road to excellence. And with both Java 8 and Java 9 in the same book, you'll have up-to-date skills on the latest Java technologies.

Most importantly, this book is not just meant for you to read—it's meant for you to practice. Be it in the classroom or at home after work, experiment with the abundant sample code and practice with the book's extraordinarily rich and diverse collection of exercises. Take the time to do all that is in here and you'll be well on your way to achieving a level of expertise that will challenge professional developers out there. After working with Java for more than 20 years, I can tell you that this is not an exaggeration.

For example, one of my favorite chapters is Lambdas and Streams. The chapter covers the topic in detail and the exercises shine—many real-world challenges that developers will encounter every day and that will help you sharpen your skills. After solving these exercises, novices and experienced developers alike will deeply understand these important Java features. And if you have a question, don't be shy—the Deitels publish their email address in every book they write to encourage interaction.

That's also why 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 may prove to be the most important tool for leveraging your learning and productivity. Paul and Harvey give a full treatment of JShell that both students and experienced developers will be able to put to use immediately.

I'm impressed with the care that the Deitels always take care to accommodate readers at all levels. They ease you into difficult concepts and deal with the challenges that professionals will encounter in industry projects.

There's lots of 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—be sure to begin with the extraordinary JShell coverage.

Another example is the amazing coverage of JavaFX—Java's latest GUI, graphics and multimedia capabilities. JavaFX is the recommended toolkit for new projects. But if you'll be working on legacy projects that use the older Swing API, those chapters are still available to you.

Make 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 multi-core world.

I encourage you to participate in the worldwide Java community. There are many helpful folks out there who stand ready to help you. Ask questions, get answers and answer your peers' questions. Along with this book, the Internet and the academic and professional communities will help speed you on your way to becoming an expert Java developer. I wish you 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 How to Program, Late Objects, Eleventh Edition***! This book presents leading-edge computing technologies for students, instructors and software developers. It's appropriate for introductory academic and professional course sequences based on the curriculum recommendations of the **ACM** and the **IEEE** professional societies,[1] and for *Advanced Placement (AP) Computer Science* exam preparation.[2] It also will help you prepare for most topics covered by the following Oracle Java Standard Edition 8 (Java SE 8) Certifications:[3]

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

If you haven't already done so, please read the bullet points and reviewer comments on the back cover and inside back cover—these concisely capture the essence of the book. In this Preface we provide more detail for students, instructors and professionals.

Our primary goal is to prepare college students to meet the Java programming challenges they'll encounter in upper-level courses and in industry. 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 complete code examples are accompanied by live sample executions.

New and Updated Features

In the following sections, we discuss the key features and updates we've made for *Java How to Program, 11/e*, including:

- Flexibility Using **Java SE 8 or the New Java SE 9** (which includes Java SE 8)
- *Java How to Program, 11/e*'s **Modular Organization**
- Introduction and Programming Fundamentals
- Flexible Coverage of **Java SE 9**: **JShell**, the **Module System** and Other Java SE 9 Topics
- **Object-Oriented Programming**
- Flexible **JavaFX/Swing** GUI, Graphics, Animation and Video Coverage

1. *Computer Science Curricula 2013 Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*, December 20, 2013, The Joint Task Force on Computing Curricula, Association for Computing Machinery (ACM), IEEE Computer Society.
2. `https://apstudent.collegeboard.org/apcourse/ap-computer-science-a/exam-practice`
3. `http://bit.ly/OracleJavaSE8Certification` (At the time of this writing, the Java SE 9 certification exams were not yet available.)

- **Data Structures** and **than**
- Flexible **Lambdas and Streams** Coverage
- **Concurrency** and **Multi-Core Performance**
- **Database: JDBC** and **JPA**
- **Web-Application Development** and **Web Services**
- Optional Online Object-**Oriented Design Case Study**

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 college and professional courses based on **Java SE 8 or Java SE 9**, which from this point forward we'll refer to simply as Java 8 and Java 9, respectively. Each feature first introduced in Java 8 or Java 9 is accompanied by an 8 or 9 icon in the margin, like those to the left of this paragraph. The new Java 9 capabilities are covered in clearly marked, *easy-to-include-or-omit* chapters and sections—some in the print book and some online. 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	
In the Print Book	***On the Companion Website***
New JShell chapter	Module system
_ is no longer allowed as an identifier	HTML5 Javadoc enhancements
`private` interface methods	`Matcher` class's new method overloads
Effectively `final` variables can be used in `try`-with-resources statements	`CompletableFuture` enhancements
	JavaFX 9 skin APIs and other enhancements
Mention of the Stack Walking API	Mentions of:
Mention of JEP 254, Compact Strings	Overview of Java 9 security enhancements
Collection factory methods	G1 garbage collector
	Object serialization security enhancements
	Enhanced deprecation

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

Java How to Program, Late Objects, 11/e's Modular Organization

The book's modular organization helps instructors plan their syllabi.

Java How to Program, Late Objects, 11/e, is appropriate for programming courses at various levels. Chapters 1–25 are popular in core CS 1 and CS 2 courses and introductory course sequences in related disciplines—these chapters appear in the **print book**. Chapters 26–36 are intended for advanced courses and are located on the book's **Companion Website**.

Part 1: Introduction
Chapter 1, Introduction to Computers, the Internet and Java
Chapter 2, Introduction to Java Applications; Input/Output and Operators

---

Chapter 25, Introduction to JShell: Java 9's REPL for Interactive Java

Part 2: Additional Programming Fundamentals
Chapter 3, Control Statements: Part 1; Assignment, ++ and -- Operators
Chapter 4, Control Statements: Part 2; Logical Operators
Chapter 5, Methods
Chapter 6, Arrays and `ArrayList`s

Chapter 14, Strings, Characters and Regular Expressions
Chapter 15, Files, Input/Output Streams, NIO and XML Serialization

Part 3: Object-Oriented Programming
Chapter 7, Introduction to Classes and Objects
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 and Multimedia
Chapter 12, JavaFX Graphical User Interfaces: Part 1
Chapter 13, JavaFX GUI: Part 2

Chapter 22, JavaFX Graphics and Multimedia

Part 5: Data Structures, Generic Collections, Lambdas and Streams
Chapter 16, Generic Collections
Chapter 17, Lambdas and Streams
Chapter 18, Recursion
Chapter 19, Searching, Sorting and Big O
Chapter 20, Generic Classes and Methods: A Deeper Look
Chapter 21, Custom Generic Data Structures

Part 6: Concurrency; Networking
Chapter 23, Concurrency

Chapter 28, Networking

Part 7: Database-Driven Desktop Development
Chapter 24, Accessing Databases with JDBC

Chapter 29, Java Persistence API (JPA)

Part 8: Web App Development and Web Services
Chapter 30, JavaServer™ Faces Web Apps: Part 1
Chapter 31, JavaServer™ Faces Web Apps: Part 2
Chapter 32, REST Web Services

Part 9: Other Java 9 Topics
Chapter 36, Java Module System and Other Java 9 Features

Part 10: (Optional) Object-Oriented Design
Chapter 33, ATM Case Study, Part 1: Object-Oriented Design with the UML
Chapter 34, ATM Case Study Part 2: Implementing an Object-Oriented Design

Part 11: (Optional) Swing Graphical User Interfaces and Java 2D Graphics
Chapter 26, Swing GUI Components: Part 1
Chapter 27, Graphics and Java 2D

Chapter 35, Swing GUI Components: Part 2

Introduction and Programming Fundamentals (Parts 1 and 2)

Chapters 1 through 7 provide a friendly, example-driven treatment of traditional introductory programming topics. This book features a **late objects approach**—see the section "Object-Oriented Programming" later in this Preface. Note in the preceding outline that Part 1 includes the (optional) Chapter 25 on Java 9's new JShell. Instructors and students who cover JShell will appreciate how its interactivity makes Java "come alive," leveraging the learning process—see the next section.

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

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 25's scores of examples and exercises, you'll see how JShell and its **instant feedback** keep your attention, enhance your performance and speed the learning and software development processes.

As a student you'll find JShell easy and fun to use. It will help you learn Java features faster and more deeply and will help you verify that these features work the way they're supposed to. As an instructor, you'll appreciate how JShell encourages your students to dig in, and that it **leverages the learning process**. As a professional you'll appreciate how JShell helps you rapidly prototype key code segments and how it helps you discover and experiment with new APIs. **If you're staying with Java 8 for a while, you can install JDKs 8 and 9 side-by-side and use JShell on JDK 9 for experimentation. The JDK 9 section of the Before You Begin that follows this Preface shows how to manage multiple JDKs on Windows, macOS and Linux.**

The **JShell** content is packaged modularly in Chapter 25. 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. 3).
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 25.3**. As you do each of them, flip the page and check your answer. This will help you master the basics of JShell quickly. Then as you do each of the **examples** in the remainder of the chapter you'll master the vast majority of JShell's capabilities.

JShell discussions	Can be covered after
Section 25.3 introduces **JShell**, including starting a session, executing statements, declaring variables, evaluating expressions, JShell's **type-inference** capabilities and more. Section 25.4 discusses command-line input with **`Scanner`** in JShell.	Chapter 2, Introduction to Java Applications; Input/Output and Operators
Section 25.5 discusses how to declare and use **classes** in JShell, including how to load a Java source-code file containing an existing class declaration. Section 25.6 shows how to use JShell's **auto-completion** capabilities to discover a class's capabilities and JShell commands.	Chapter 7, Introduction to Classes and Objects

Fig. 3 | Chapter 25 JShell discussions that are designed to be covered after specific earlier chapters. (Part 1 of 2.)

JShell discussions	Can be covered after
Section 25.7 presents additional JShell auto-completion capabilities for **experimentation and discovery**, including viewing method parameters, documentation and method overloads. Section 25.8 shows how to declare and use methods in JShell, including **forward referencing** a method that does not yet exist in the JShell session.	Chapter 5, Methods
Section 25.9 shows how **exceptions** are handled in JShell.	Chapter 6, Arrays and `ArrayLists`
Section 25.10 shows how to add existing **packages** to the classpath and import them for use in JShell.	Chapter 21, Custom Generic Data Structures
The remaining JShell sections are reference material that can be covered after Section 25.10. 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. 3 | Chapter 25 JShell discussions that are designed to be covered after specific earlier chapters. (Part 2 of 2.)

9 ***New Chapter—The Java Module System and Other Java 9 Topics***
Because Java 9 was still under development when this book was published, we included an online chapter on the book's Companion Website that discusses Java 9's module system and various other Java 9 topics. This **online content will be available before Fall 2017 courses**.

Object-Oriented Programming (Part 3)

Object-oriented programming. We use a **late objects approach**, covering programming fundamentals such as data types, variables, operators, control stattements, methods and arrays in the early chapters. Then students develop their first customized classes and objects in Chapter 7. [For courses that require an **early-objects approach**, you may want to consider our sister book ***Java How to Program, Early Objects, 11/e***.]

Real-world case studies. The object-oriented programing presentation in the classes chapters features `Account`, `Time`, `Employee`, `GradeBook` and `Card` shuffling-and-dealing case studies.

Inheritance, Interfaces, Polymorphism and Composition. We use additional real-world case studies—including class `Time`, an `Employee` class hierarchy, and a `Payable` interface implemented in disparate `Employee` and `Invoice` classes—to illustrate these OO concepts and explain situations in which each is preferred in building industrial-strength applications. We also explain the use of current idioms, such as "**programming to an interface not an implementation**" and "**preferring composition to inheritance**."

Exception handling. We integrate basic exception handling beginning in Chapter 6 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.

Classes `Arrays` *and* `ArrayList`. Chapter 6 covers class `Arrays`—which contains methods for performing common array manipulations—and class `ArrayList`—which implements a dynamically resizable array-like data structure. This follows our philosophy of getting lots of practice using existing classes while learning how to define your own. The chapter's rich selection of exercises includes a substantial project on **building your own computer** through the technique of software simulation. Chapter 21 includes a follow-on project on **building your own compiler** that can compile high-level language programs into machine language code that will actually execute on your computer simulator. Students in first and second programming courses, respectively, enjoy these challenges.

Flexible JavaFX GUI, Graphics, Animation and Video Coverage (Part 4) and Optional Swing Coverage (Part 11)

Students enjoy building applications with **GUI**, **graphics**, **animations** and **videos**. Instructors teaching introductory courses can choose the amount of GUI, graphics, animation and video they'd like to cover—from none at all to the four options discussed below. Those who want to use the newer **JavaFX GUI, graphics, animation and video** capabilities (today's most popular options for college courses and professionals) in their courses can choose from:

- a **deep treatment** of **JavaFX GUI**, **graphics (2D and 3D)**, **animation** and **video** in Chapters 12, 13 and 22, or
- a **lighter treatment** of **JavaFX GUI** and **2D graphics** on the Companion Website that can be taught anytime after Chapter 7.

Those who want to continue using the older **Swing GUI** and **Java 2D graphics** in their courses can choose from the following options on the Companion Website:

- a **deep treatment** of **Swing GUI** and **Java 2D graphics** in online Chapters 26, 27 and 35, or
- a **lighter treatment** of **Swing GUI** and **Java 2D graphics** that can be taught anytime after Chapter 7.

Let's consider these four options in more detail.

Deep Treatment of JavaFX GUI, Graphics, Animation and Video in Chapters 12, 13, 22
For this 11th edition, we've significantly updated our **JavaFX** presentation and moved all three chapters into the print book, replacing our **Swing GUI and graphics coverage** (which is now on the book's **Companion Website** for instructors who want to continue with Swing).

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 that demonstrate various JavaFX GUI layouts, controls and event-handling capabilities. In **Swing**, drag-and-drop tools and their generated code are *IDE dependent*. 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 22, we present **JavaFX 2D and 3D graphics**, **animation** and **video** capabilities. We also provide **36 programming exercises and projects** that students will find

challenging and entertaining, including many **game-programming exercises**. **Chapter 22 can be covered immediately after Section 13.3**. We also use JavaFX in several GUI-based examples in Chapter 23, Concurrency and Chapter 24, Accessing Databases with JDBC.

Lighter Treatment of JavaFX GUI and 2D Graphics
For instructors who like to introduce a lighter treatment of JavaFX GUI and graphics earlier than Chapter 12, we've placed on the book's Companion Website a lighter case study (Fig. 4)[1] that can be taught anytime after Chapter 7. The goal is to create a simple polymorphic drawing application in which the user can select a shape to draw and the shape's characteristics (such as its color, stroke thickness and whether it's hollow or filled) then drag the mouse to position and size the shape. The case study builds gradually toward that goal, with the reader implementing a polymorphic drawing app, then adding a more robust user interface. For courses that include these case study sections, instructors can opt to cover none, some or all of the deeper treatment in Chapters 12, 13 and 22.

Part	What you'll do
1. A Simple GUI	Display text and an image.
2. Event Handling and Drawing Lines	In response to a `Button` click, draw lines using JavaFX graphics capabilities.
3. Drawing Rectangles and Ovals	Draw rectangles and ovals.
4. Colors and Filled Shapes	Draw filled shapes in multiple colors.
5. Drawing Arcs	Draw a rainbow of colored arcs.
6. Using Objects with Graphics	Store shapes as objects then have those objects to draw themselves on the screen.
7. Drawing with Polymorphism	Identify the similarities between shape classes and create and use a shape class hierarchy.
8. Interactive Polymorphic Drawing Application	In capstone Exercise 13.9 you'll enable users to select each shape to draw, configure its properties (such as color and fill), and drag the mouse to position and size the shape.

Fig. 4 | Optional JavaFX GUI and Graphics Case Study.

Deep Treatment Swing GUI and 2D Graphics
Swing is still widely used, but Oracle will provide only minor updates going forward. For instructors and readers who wish to continue using Swing, we've moved to the book's **Companion Website** the 10th edition's

- Chapter 26, Swing GUI Components: Part 1
- Chapter 27, Graphics and Java 2D
- Chapter 35, Swing GUI Components: Part 2.

See the "Companion Website" section later in this Preface.

1. The deeper graphics treatment in Chapter 22 uses JavaFX shape types that can be added directly to the GUI using **Scene Builder**.

Lighter Treatment of Swing GUI and 2D Graphics
We've also moved to the Companion Website the older Swing-based version of the lighter JavaFX case study in Fig. 4.

Integrating Swing GUI Components in JavaFX GUIs
If you move to JavaFX, you still can use your favorite Swing capabilities. For example, in Chapter 24, 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.

Data Structures and Generic Collections (Part 5)

Data structures presentation. The chapters of Part 5 form the core of a second programming course emphasizing data structures. We begin with generic collection class `ArrayList` in Chapter 6. Our later data structures discussions (Chapters 16–21) provide a deeper treatment of generic collections—showing how to use many additional built-in collections of the Java API.

We discuss **recursion**, which is important for many reasons including implementing tree-like, data-structure classes. For computer-science majors and students in related disciplines, we discuss **searching and sorting algorithms** for manipulating the contents of collections, and provide a friendly introduction to **Big O**—a means of describing mathematically how hard an algorithm might have to work to solve a problem. Most programmers should use the built-in searching and sorting capabilities of the collections classes.

We then show how to implement **custom generic methods and classes**, including **custom generic data structures** (this, too, is intended for computer-science majors—in industry, 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)—we suggest that students begin by covering Sections 17.1—17.7 after Chapter 6 and that professionals begin by covering Sections 17.1—17.5 after Chapter 4. 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 17.1—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 4, Control Statements: Part 2; Logical Operators
Section 17.6 introduces **method references** and additional streams capabilities.	Chapter 5, Methods
Section 17.7 introduces streams capabilities that process **one-dimensional arrays**.	Chapter 6, Arrays and `ArrayLists`
Sections 17.8—17.10 demonstrate additional streams capabilities and present various **functional interfaces used in streams processing**.	Chapter 10, Object-Oriented Programming: Polymorphism and Interfaces—Section 10.10 introduces Java 8 interface features (**`default` methods**, **`static` methods** and the concept of **functional interfaces**) for the functional interfaces that support lambdas and streams.
Section 17.11 uses lambdas and streams to **process collections of `String` objects**.	Chapter 14, Strings, Characters and Regular Expressions
Section 17.12 uses lambdas and streams to **process a `List<Employee>`**.	Chapter 16, Generic Collections
Section 17.13 uses 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 uses lambdas to implement **JavaFX event-listener interfaces**.	Chapter 12, JavaFX Graphical User Interfaces: Part 1
Chapter 23, Concurrency, 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 *Java How to Program, 10/e* Brian Goetz, co-author of *Java Concurrency in Practice* (Addison-Wesley). We updated Chapter 23, Concurrency, 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 **`Completable-`**

Future enhancements in the online Java 9 chapter. Finally, we added several new exercises, including one that demonstrates the problems with parallelizing Java 8 streams that apply non-associative operations and several that have the reader investigate and use the **Fork/Join framework** to **parallelize recursive algorithms**.

JavaFX concurrency. In this edition, we converted Chapter 23's Swing-based GUI examples to JavaFX. 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 24 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/`).

Java Persistence API. Chapter 29 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.

Web Application Development and Web Services (Part 8)

Java Server Faces (JSF). Chapters 30–31 introduce the **JavaServer™ Faces** (**JSF**) technology for building web-based applications. Chapter 30 includes examples on building web application GUIs, validating forms and session tracking. Chapter 31 discusses data-driven JSF applications—including a **multi-tier web address book application** that allows users to add and search for contacts.

Web services. Chapter 32 now concentrates on creating and consuming **REST-based web services**. Most of today's web services use REST, which is simpler and more flexible than older web-services technologies that often required manipulating data in only XML format. REST can use a variety of formats, such as JSON, HTML, plain text, media files and XML.

Optional Online Object-Oriented Design Case Study (Part 10)

Developing an Object-Oriented Design and Java Implementation of an ATM. Chapters 33–34 include an *optional* case study on object-oriented design using the UML (Unified Modeling Language™)—the industry-standard graphical language for modeling object-oriented systems. We design and implement the software for a simple automated teller machine (ATM). We analyze a typical **requirements document** that specifies the system to be built. We determine the **classes** needed to implement that system, the **attributes** the classes need to have, the **behaviors** the classes need to exhibit and specify how the classes must **interact** with one another to meet the system requirements. From the design we produce a **complete Java implementation**. Students 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 How to Program, 11/e, contains hundreds of complete working code examples. We stress program clarity and concentrate on building well-engineered software.

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

```
comments appear in light gray like this
keywords appear bold black like this
constants and literal values appear in bold dark gray like this
all other code appears in black like this
```

Code Highlighting. We place gray 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** 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 list of chapter objectives provides a high-level overview of the chapter's contents.

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

Summary Bullets. We present a section-by-section bullet-list summary of the chapter. For ease of reference, we generally include the page number of each key term's defining occurrence in the text.

Self-Review Exercises and Answers. Extensive self-review exercises *and* answers are included for self study. All of the exercises in the optional ATM case study are fully solved.

Exercises. The chapter exercises include:

- simple recall of important terminology and concepts
- What's wrong with this code?
- What does this code do?
- writing individual statements and small portions of methods and classes
- writing complete methods, classes and programs
- major projects
- in many chapters, **Making a Difference exercises** that encourage you to use computers and the Internet to research and address significant social problems.
- In this edition, we added new exercises to our **game-programming** set (**SpotOn**, **Horse Race**, **Cannon**, **15 Puzzle**, **Hangman**, **Block Breaker**, **Snake** and **Word Search**), as well as others on the **JavaMoney API**, `final` instance variables, combining composition and inheritance, working with interfaces, drawing fractals, recursively searching directories, visualizing sorting algorithms and implementing parallel recursive algorithms with the Fork/Join framework. Many of these require students to research additional Java features online and use them.

Index. We've included an extensive index. Defining occurrences of key terms are highlighted with a **bold** page number. The print book index mentions only those terms used in the print book. **The online chapters index on the Companion Website includes all the print book terms and the online chapter terms.**

Programming Wisdom

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

Good Programming Practice

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

Common Programming Error

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

Error-Prevention Tip

These tips contain suggestions for exposing 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

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

Portability Tip

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

Software Engineering Observation

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

Look-and-Feel Observation

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

What are 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/CERTOracleSecureJava
```

and adhered to various secure coding practices as appropriate for a textbook 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 *C++ How to Program, 10/e* and *Java How to Program, 11/e*.

Companion Website: Source Code, VideoNotes, Online Chapters and Online Appendices

All the source code for the book's code examples is available at the book's **Companion Website**, which also contains extensive **VideoNotes** and the online chapters and appendices:

```
www.pearsonglobaleditions.com
```

See the book's inside front cover for information on accessing the Companion Website.

In the extensive **VideoNotes**, co-author Paul Deitel patiently explains most of the programs in the book's core chapters. Students like viewing the VideoNotes for reinforcement of core concepts and for additional insights.

Software Used in *Java How to Program, 11/e*

All the software you'll need for this book is available free for download from the Internet. See the **Before You Begin** section that follows this Preface for links to each download. We wrote most of the examples using the free Java Standard Edition Development Kit (JDK) 8. For the optional 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 programs were tested on Windows, macOS and Linux. Several online chapters use the Netbeans IDE.

8
9

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/
```

Java How to Program, Early Objects Version, 11/e

There are several approaches to teaching first courses in Java programming. The two most popular are the **late objects approach** and the **early objects approach**. To meet these diverse needs, we've published two versions of this book:

- *Java How to Program, Late Objects Version, 11/e* (this book), and
- *Java How to Program, Early Objects Version, 11/e*

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. Instructors can request an examination copy of either of these books from their Pearson representative.

Instructor Supplements

The following supplements are available to qualified instructors only through Pearson Education's Instructor Resource Center (`www.pearsonglobaleditions.com`):

- ***PowerPoint® slides*** containing all the code and figures in the text, plus bulleted items that summarize key points.
- ***Test Item File*** of multiple-choice questions and answers (approximately two per book section).
- ***Solutions Manual*** with solutions to most of the end-of-chapter exercises. **Before assigning an exercise for homework, instructors should check the IRC to be sure it includes the solution. Solutions are *not* provided for "project" exercises.**

Please do not write to us requesting access to the Pearson Instructor's Resource Center which contains the book's instructor supplements, including the exercise solutions. Access is limited strictly to college instructors teaching from the book. Instructors may obtain access only through their Pearson representatives. Solutions are *not* provided for "project" exercises. If you're not a registered faculty member, contact your Pearson representative.

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 guidance, wisdom and energy of Tracy Johnson, Executive Editor, Computer Science. Tracy and her team handle all of our academic textbooks. Kristy Alaura recruited the book's reviewers and managed the review process. Bob Engelhardt managed the book's publication. We selected the cover art and Chuti Prasertsith designed the cover.

Reviewers

We wish to acknowledge the efforts of our recent editions reviewers—a distinguished group of academics, 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 three JavaFX chapters.

Eleventh Edition reviewers: Marty Allen (University of Wisconsin-La Crosse), Robert Field (JShell chapter only; JShell Architect, Oracle), Trisha Gee (JetBrains, Java Champion), Jonathan Giles (Consulting Member of Technical Staff, Oracle), Brian Goetz (JShell chapter only; Oracle's Java Language Architect), Edwin Harris (M.S. Instructor at The University of North Florida's School of Computing), Maurice Naftalin (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, Java Champion), Johan Vos (CTO, Cloud Products at Gluon, Java Champion).

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

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

A Special Thank You to 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 the 10th edition. He provided us with an extraordinary collection of insights and constructive comments. For the 11th edition, he did a detailed review of our new JShell chapter 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 hope you enjoy working with *Java How to Program, 11/e*, as much as we enjoyed researching and writing it!

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, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world's best-selling programming-language textbook/professional book/video authors.

Dr. Harvey 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 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 app development, Swift and iOS app development, C++, C, Visual C#®, Visual Basic®, 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 textbooks and professional books in print and e-book formats, **LiveLessons** video courses and **REVEL™** online interactive multimedia courses with integrated **MyProgrammingLab**. Deitel & Associates, Inc. and the authors can be reached at:

```
deitel@deitel.com
```

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

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

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

```
deitel@deitel.com
```

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

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

Acknowledgments for the Global Edition

Pearson would like to thank and acknowledge the following people for their contribution to the Global Edition.

Contributor: Muthuraj M.

Reviewers: Annette Bieniusa (University of Kaiserslautern), Bogdan Oancea (University of Bucharest), and Shaligram Prajapat (Devi Ahilya University)

Before You Begin

This section contains information you should review before using this book. 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 an early access verion. If you're using this book before the final JDK 9 is released, see the section "Installing and Configuring JDK 9 Early Access Version" 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 on your computer designates which directories the computer searches when looking for applications, such as the applications that enable you to compile and run your Java applications (called `javac` and `java`, respectively). *Carefully follow the installation instructions for Java on your platform to ensure that you set the* `PATH` *environment variable correctly.* The steps for setting environment variables differ by operating system. Instructions for various platforms are listed at:

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

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

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

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

JDK Installation Directory and the `bin` *Subdirectory*

The JDK's installation directory varies by platform. The directories listed below are for Oracle's JDK 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 if you're using a different JDK 8 update. 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

The Java DB database software that you'll use in Chapter 24 and several online chapters requires you to set the JAVA_HOME environment variable to your JDK's installation directory. The same steps you used to set the PATH may also be used to set other environment variables, such as JAVA_HOME.

Java Integrated Development Environments (IDEs)

There are many Java integrated development environments that you can use for Java programming. 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. We use NetBeans in several of the book's online chapters.

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

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

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

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

This version supports both Java SE and Java EE development.

Eclipse Downloads
You can download the Eclipse IDE from:

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

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

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

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

The free version supports only Java SE development.

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

Java How to Program, 11/e's examples are available for download at

```
http://www.deitel.com/books/jhtp11LOV/
```

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 and Linux

Installing and Configuring JDK 9 Early Access Version

Throughout the book, we introduce various new Java 9 features in optional sections and chapters. 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, Oracle has changed the numbering scheme. JDK 9 initially will be known as `jdk-9`. Once Java 9 is officially released, there will be future 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 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. In our case

```
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:

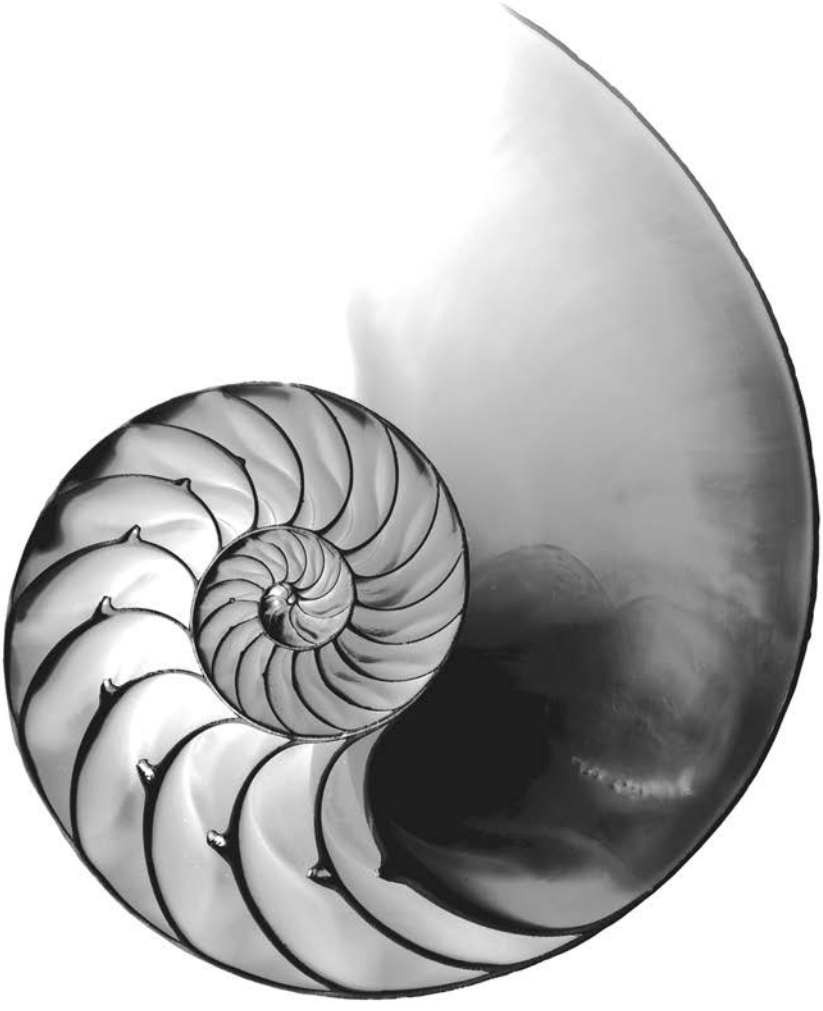
```
export PATH="location:$PATH"
```

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 your Java studies with *Java How to Program, Late Objects, 11/e*. We hope you enjoy the book!

1

Introduction to Computers, the Internet and Java



Objectives

In this chapter you'll:

- Learn about exciting recent developments in the computer field.
- Learn computer hardware, software and networking basics.
- Understand the data hierarchy.
- Understand the different types of programming languages.
- Understand the importance of Java and other leading programming languages.
- Understand object-oriented programming basics.
- Learn Internet and web basics.
- Learn a typical Java program-development environment.
- Test-drive a Java application.
- Learn some key recent software technologies.
- See how to keep up-to-date with information technologies.

Outline

1.1 Introduction

Welcome to Java—one of the world's most widely used computer programming languages and, according to the TIOBE Index, the world's most popular.[1] You're probably familiar with the powerful tasks computers perform. Using this textbook, you'll write instructions in the Java programming language commanding computers to perform those tasks. **Software** (i.e., the instructions you write) controls **hardware** (i.e., computers).

You'll learn *object-oriented programming*—today's key programming methodology. You'll create and work with many *software objects*.

For many organizations, the preferred language for meeting their enterprise programming needs is Java. Java is 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,[2] 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.

1. http://www.tiobe.com/tiobe-index/
2. http://bit.ly/JavaOne2016Keynote

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 this eleventh edition of *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 instructors and professionals 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 and its Companion Website.

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. Later in this book we discuss how to build such web-based applications with Java.

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 Hardware and Software

Computers can perform calculations and make logical decisions phenomenally faster than human beings can. Many of today's personal computers can perform billions of calculations in one second—more than a human can perform in a lifetime. *Supercomputers* are already performing *thousands of trillions (quadrillions)* of instructions per second! China's National Research Center of Parallel Computer Engineering & Technology (NRCPC) has developed the Sunway TaihuLight supercomputer can perform over 93 quadrillion calculations per second (93 *petaflops*)![3] To put that in perspective, *the Sunway TaihuLight supercomputer can perform in one second over 12 million calculations for every person on the planet!* And supercomputing upper limits are growing quickly.

Computers process data under the control of sequences of instructions called **computer programs**. These software programs guide the computer through ordered actions specified by people called computer **programmers**. In this book, you'll learn key programming methodologies that are enhancing programmer productivity, thereby reducing software development costs.

A computer consists of various devices referred to as hardware (e.g., the keyboard, screen, mouse, hard disks, memory, DVD drives and processing units). Computing costs are *dropping dramatically*, owing to rapid developments in hardware and software technologies. Computers that might have filled large rooms and cost millions of dollars decades ago are now inscribed on silicon chips smaller than a fingernail, costing perhaps a few dollars each. Ironically, silicon is one of the most abundant materials on Earth—it's an ingredient in common sand. Silicon-chip technology has made computing so economical that computers have become a commodity.

1.2.1 Moore's Law

Every year, you probably expect to pay at least a little more for most products and services. The opposite has been the case in the computer and communications fields, especially with regard to the hardware supporting these technologies. For many decades, hardware costs have fallen rapidly.

Every year or two, the capacities of computers have approximately *doubled* inexpensively. This remarkable trend often is called **Moore's Law**, named for the person who identified it in the 1960s, Gordon Moore, co-founder of Intel—the leading manufacturer of the processors in today's computers and embedded systems. Moore's Law *and related observations* apply especially to the amount of memory that computers have for programs,

3. https://www.top500.org/lists/2016/06/

the amount of secondary storage (such as solid-state drive storage) they have to hold programs and data over longer periods of time, and their processor speeds—the speeds at which they *execute* their programs (i.e., do their work).

Similar growth has occurred in the communications field—costs have plummeted as enormous demand for communications *bandwidth* (i.e., information-carrying capacity) has attracted intense competition. We know of no other fields in which technology improves so quickly and costs fall so rapidly. Such phenomenal improvement is truly fostering the *Information Revolution*.

1.2.2 Computer Organization

Regardless of differences in *physical* appearance, computers can be envisioned as divided into various **logical units** or sections (Fig. 1.2).

Logical unit	Description
Input unit	This "receiving" section obtains information (data and computer programs) from **input devices** and places it at the disposal of the other units for processing. Most user input is entered into computers through keyboards, touch screens and mouse devices. Other forms of input include receiving voice commands, scanning images and bar codes, reading from secondary storage devices (such as hard drives, DVD drives, Blu-ray Disc™ drives and USB flash drives—also called "thumb drives" or "memory sticks"), receiving video from a webcam and having your computer receive information from the Internet (such as when you stream videos from YouTube® or download e-books from Amazon). Newer forms of input include position data from a GPS device, motion and orientation information from an *accelerometer* (a device that responds to up/down, left/right and forward/backward acceleration) in a smartphone or game controller (such as Microsoft® Kinect® for Xbox®, Wii™ Remote and Sony® PlayStation® Move) and voice input from intelligent assistants like Amazon Echo and Google Home.
Output unit	This "shipping" section takes information the computer has processed and places it on various **output devices** to make it available for use outside the computer. Most information that's output from computers today is displayed on screens (including touch screens), printed on paper ("going green" discourages this), played as audio or video on PCs and media players (such as Apple's iPods) and giant screens in sports stadiums, transmitted over the Internet or used to control other devices, such as robots and "intelligent" appliances. Information is also commonly output to secondary storage devices, such as solid-state drives (SSDs), hard drives, DVD drives and USB flash drives. Popular recent forms of output are smartphone and game-controller vibration, virtual reality devices like Oculus Rift® and Google Cardboard™ and mixed reality devices like Microsoft's HoloLens™.

Fig. 1.2 | Logical units of a computer. (Part 1 of 2.)

Logical unit	Description
Memory unit	This rapid-access, relatively low-capacity "warehouse" section retains information that has been entered through the input unit, making it immediately available for processing when needed. The memory unit also retains processed information until it can be placed on output devices by the output unit. Information in the memory unit is *volatile*—it's typically lost when the computer's power is turned off. The memory unit is often called either **memory**, **primary memory** or **RAM** (Random Access Memory). Main memories on desktop and notebook computers contain as much as 128 GB of RAM, though 2 to 16 GB is most common. GB stands for gigabytes; a gigabyte is approximately one billion bytes. A **byte** is eight bits. A bit is either a `0` or a `1`.
Arithmetic and logic unit (ALU)	This "manufacturing" section performs *calculations*, such as addition, subtraction, multiplication and division. It also contains the *decision* mechanisms that allow the computer, for example, to compare two items from the memory unit to determine whether they're equal. In today's systems, the ALU is implemented as part of the next logical unit, the CPU.
Central processing unit (CPU)	This "administrative" section coordinates and supervises the operation of the other sections. The CPU tells the input unit when information should be read into the memory unit, tells the ALU when information from the memory unit should be used in calculations and tells the output unit when to send information from the memory unit to certain output devices. Many of today's computers have multiple CPUs and, hence, can perform many operations simultaneously. A **multicore processor** implements multiple processors on a single integrated-circuit chip—a *dual-core processor* has two CPUs, a *quad-core processor* has four and an *octa-core processor* has eight. Intel has some processors with up to 72 cores. Today's desktop computers have processors that can execute billions of instructions per second. Chapter 23 explores how to write apps that can take full advantage of multicore architecture.
Secondary storage unit	This is the long-term, high-capacity "warehousing" section. Programs or data not actively being used by the other units normally are placed on secondary storage devices (e.g., your *hard drive*) until they're again needed, possibly hours, days, months or even years later. Information on secondary storage devices is *persistent*—it's preserved even when the computer's power is turned off. Secondary storage information takes much longer to access than information in primary memory, but its cost per unit is much less. Examples of secondary storage devices include solid-state drives (SSDs), hard drives, DVD drives and USB flash drives, some of which can hold over 2 TB (TB stands for terabytes; a terabyte is approximately one trillion bytes). Typical hard drives on desktop and notebook computers hold up to 2 TB, and some desktop hard drives can hold up to 10 TB.

Fig. 1.2 | Logical units of a computer. (Part 2 of 2.)

1.3 Data Hierarchy

Data items processed by computers form a **data hierarchy** that becomes larger and more complex in structure as we progress from the simplest data items (called "bits") to richer ones, such as characters and fields. Figure 1.3 illustrates a portion of the data hierarchy.

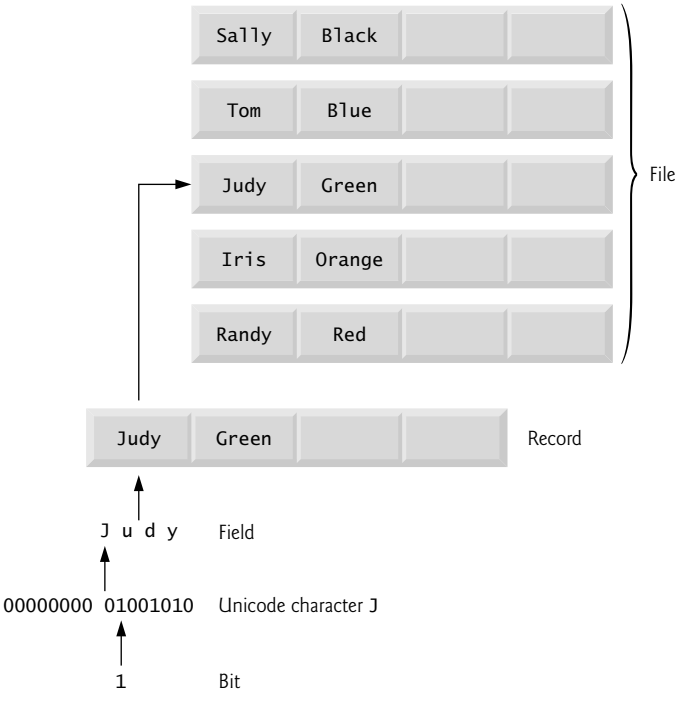


Fig. 1.3 | Data hierarchy.

Bits

The smallest data item in a computer can assume the value `0` or the value `1`. It's called a **bit** (short for "binary digit"—a digit that can assume one of *two* values). Remarkably, the impressive functions performed by computers involve only the simplest manipulations of `0`s and `1`s—*examining a bit's value*, *setting a bit's value* and *reversing a bit's value* (from `1` to `0` or from `0` to `1`).

Characters

It's tedious for people to work with data in the low-level form of bits. Instead, they prefer to work with *decimal digits* (0–9), *letters* (A–Z and a–z), and *special symbols* (e.g., $, @, %, &, *, (,), –, +, ", :, ? and /). Digits, letters and special symbols are known as **characters**. The computer's **character set** is the set of all the characters used to write programs and represent data items. Computers process only `1`s and `0`s, so a computer's character set represents every character as a pattern of `1`s and `0`s. Java uses **Unicode®** characters that are composed of one, two or four bytes (8, 16 or 32 bits). Unicode contains characters for many of the world's languages. See Appendix H for more information on Unicode. See

Appendix B for more information on the **ASCII** (**American Standard Code for Information Interchange**) character set—the popular subset of Unicode that represents uppercase and lowercase letters, digits and some common special characters.

Fields

Just as characters are composed of bits, **fields** are composed of characters or bytes. A field is a group of characters or bytes that conveys meaning. For example, a field consisting of uppercase and lowercase letters can be used to represent a person's name, and a field consisting of decimal digits could represent a person's age.

Records

Several related fields can be used to compose a **record** (implemented as a `class` in Java). In a payroll system, for example, the record for an employee might consist of the following fields (possible types for these fields are shown in parentheses):

- Employee identification number (a whole number)
- Name (a string of characters)
- Address (a string of characters)
- Hourly pay rate (a number with a decimal point)
- Year-to-date earnings (a number with a decimal point)
- Amount of taxes withheld (a number with a decimal point)

Thus, a record is a group of related fields. In the preceding example, all the fields belong to the *same* employee. A company might have many employees and a payroll record for each.

Files

A **file** is a group of related records. [*Note:* More generally, a file contains arbitrary data in arbitrary formats. In some operating systems, a file is viewed simply as a *sequence of bytes*—any organization of the bytes in a file, such as organizing the data into records, is a view created by the application programmer. You'll see how to do that in Chapter 15.] It's not unusual for an organization to have many files, some containing billions, or even trillions, of characters of information.

Database

A **database** is a collection of data organized for easy access and manipulation. The most popular model is the *relational database*, in which data is stored in simple *tables*. A table includes *records* and *fields*. For example, a table of students might include first name, last name, major, year, student ID number and grade point average fields. The data for each student is a record, and the individual pieces of information in each record are the fields. You can *search*, *sort* and otherwise manipulate the data based on its relationship to multiple tables or databases. For example, a university might use data from the student database in combination with data from databases of courses, on-campus housing, meal plans, etc. We discuss databases in Chapter 24, Accessing Databases with JDBC and online Chapter 29, Java Persistence API (JPA).

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,[4] and according to Salesforce.com, as of October 2015 90% of the world's data was created in just the prior 12 months![5] According to an IDC study, the global data supply will reach 40 *zettabytes* (equal to 40 trillion gigabytes) annually by 2020.[6] 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.

Unit	Bytes	Which is approximately
1 kilobyte (KB)	1024 bytes	10^3 (1024) bytes exactly
1 megabyte (MB)	1024 kilobytes	10^6 (1,000,000) bytes
1 gigabyte (GB)	1024 megabytes	10^9 (1,000,000,000) bytes
1 terabyte (TB)	1024 gigabytes	10^{12} (1,000,000,000,000) bytes
1 petabyte (PB)	1024 terabytes	10^{15} (1,000,000,000,000,000) bytes
1 exabyte (EB)	1024 petabytes	10^{18} (1,000,000,000,000,000,000) bytes
1 zettabyte (ZB)	1024 exabytes	10^{21} (1,000,000,000,000,000,000,000) bytes

Fig. 1.4 | Byte measurements.

1.4 Machine Languages, Assembly Languages and High-Level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate *translation* steps. Hundreds of such languages are in use today. These may be divided into three general types:

1. Machine languages
2. Assembly languages
3. High-level languages

Machine Languages
Any computer can directly understand only its own **machine language**, defined by its hardware design. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are *machine dependent* (a particular machine language can be used on only one type of computer). Such languages are cumbersome for humans.

4. http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html
5. https://www.salesforce.com/blog/2015/10/salesforce-channel-ifttt.html
6. http://recode.net/2014/01/10/stuffed-why-data-storage-is-hot-again-really/

For example, here's a section of an early machine-language payroll program that adds overtime pay to base pay and stores the result in gross pay:

```
+1300042774
+1400593419
+1200274027
```

Assembly Languages and Assemblers
Programming in machine language was simply too slow and tedious for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent elementary operations. These abbreviations formed the basis of **assembly languages**. *Translator programs* called **assemblers** were developed to convert early assembly-language programs to machine language at computer speeds. The following section of an assembly-language payroll program also adds overtime pay to base pay and stores the result in gross pay:

```
load   basepay
add    overpay
store  grosspay
```

Although such code is clearer to humans, it's incomprehensible to computers until translated to machine language.

High-Level Languages and Compilers
With the advent of assembly languages, computer usage increased rapidly, but programmers still had to use numerous instructions to accomplish even the simplest tasks. To speed the programming process, **high-level languages** were developed in which single statements could be written to accomplish substantial tasks. Translator programs called **compilers** convert high-level language programs into machine language. High-level languages allow you to write instructions that look almost like everyday English and contain commonly used mathematical notations. A payroll program written in a high-level language might contain a *single* statement such as

```
grossPay = basePay + overTimePay
```

From the programmer's standpoint, high-level languages are preferable to machine and assembly languages. Java is the world's most widely used high-level programming language.

Interpreters
Compiling a large high-level language program into machine language can take considerable computer time. *Interpreter* programs, developed to execute high-level language programs directly, avoid the delay of compilation, although they run slower than compiled programs. We'll say more about interpreters in Section 1.9, where you'll learn that Java uses a clever performance-tuned mixture of compilation and interpretation to run programs.

1.5 Basic Introduction to Object Terminology

[*Note:* In Java, even simple programs, such as those we begin with in Chapter 2, use basic object-oriented concepts like "classes," "objects" and "methods." This section gently introduces those basics. Chapters 7–11 provide our late-objects deep treatment of object-

oriented programming.] As demands for more powerful software soar, 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 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 an object-oriented design-and-implementation approach to be much more productive than with earlier techniques like "structured programming"—object-oriented programs are often easier to understand, correct and modify.

1.5.1 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.

1.5.2 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 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 bank-account class 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.

1.5.3 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.

1.5.4 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.

Software Engineering Observation 1.1

Use a building-block approach to creating your programs. Avoid reinventing the wheel—use existing high-quality pieces wherever possible. This software reuse *is a key benefit of object-oriented programming.*

1.5.5 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.

1.5.6 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**.

1.5.7 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.

1.5.8 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.

1.5.9 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*.

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.

1.5.10 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**. Languages like Java are object oriented. Programming in such a language, called **object-oriented programming (OOP)**, allows you to implement an object-oriented design as a working system.

1.5.11 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 use UML diagrams in our discussions of control statements in Chapters 3 and 4, and in our discussions of object-oriented programming in Chapters 7–11. In our *optional* online ATM Software Engineering Case Study in Chapters 33–34 we present a simple subset of the UML's features as we guide you through an object-oriented design experience.

1.6 Operating Systems

Operating systems are software systems that make using computers more convenient for users, application developers and system administrators. They provide services that allow each application to execute safely, efficiently and *concurrently* (i.e., in parallel) with other applications. The software that contains the core components of the operating system is called the **kernel**. Popular desktop operating systems include Linux, Windows and macOS (formerly called OS X)—we used all three in developing this book. The most popular mobile operating systems used in smartphones and tablets are Google's Android and Apple's iOS (for iPhone, iPad and iPod Touch devices).

1.6.1 Windows—A Proprietary Operating System

In the mid-1980s, Microsoft developed the **Windows operating system**, consisting of a graphical user interface built on top of DOS (Disk Operating System)—an enormously popular personal-computer operating system that users interacted with by typing commands. Windows borrowed from many concepts (such as icons, menus and windows) developed by Xerox PARC and popularized by early Apple Macintosh operating systems. Windows 10 is Microsoft's latest operating system—its features include enhancements to the **Start** menu and user interface, Cortana personal assistant for voice interactions, Action Center for receiving notifications, Microsoft's new Edge web browser, and more. Windows is a *proprietary* operating system—it's controlled by Microsoft exclusively. Windows is by far the world's most widely used desktop operating system.

1.6.2 Linux—An Open-Source Operating System

The **Linux operating system** is perhaps the greatest success of the *open-source* movement. **Open-source software** departs from the *proprietary* software development style that dominated software's early years. With open-source development, individuals and companies *contribute* their efforts in developing, maintaining and evolving software in exchange for the right to use that software for their own purposes, typically at *no charge*. Open-source code is often scrutinized by a much larger audience than proprietary software, so errors often get removed faster. Open source also encourages innovation. Enterprise systems companies, such as IBM, Oracle and many others, have made significant investments in Linux open-source development.

Some key organizations in the open-source community are

- the Eclipse Foundation (the Eclipse Integrated Development Environment helps programmers conveniently develop software)
- the Mozilla Foundation (creators of the Firefox web browser)
- the Apache Software Foundation (creators of the Apache web server used to develop web-based applications)
- GitHub (which provides tools for managing open-source projects—it has millions of them under development).

Rapid improvements to computing and communications, decreasing costs and open-source software have made it much easier and more economical to create a software-based

business now than just a decade ago. A great example is Facebook, which was launched from a college dorm room and built with open-source software.

The **Linux kernel** is the core of the most popular open-source, freely distributed, full-featured operating system. It's developed by a loosely organized team of volunteers and is popular in servers, personal computers and embedded systems (such as the computer systems at the heart of smartphones, smart TVs and automobile systems). Unlike that of proprietary operating systems like Microsoft's Windows and Apple's macOS, Linux source code (the program code) is available to the public for examination and modification and is free to download and install. As a result, Linux users benefit from a huge community of developers actively debugging and improving the kernel, and the ability to customize the operating system to meet specific needs.

A variety of issues—such as Microsoft's market power, the small number of user-friendly Linux applications and the diversity of Linux distributions, such as Red Hat Linux, Ubuntu Linux and many others—have prevented widespread Linux use on desktop computers. Linux has become extremely popular on servers and in embedded systems, such as Google's Android-based smartphones.

1.6.3 Apple's macOS and Apple's iOS for iPhone®, iPad® and iPod Touch® Devices

Apple, founded in 1976 by Steve Jobs and Steve Wozniak, quickly became a leader in personal computing. In 1979, Jobs and several Apple employees visited Xerox PARC (Palo Alto Research Center) to learn about Xerox's desktop computer that featured a graphical user interface (GUI). That GUI served as the inspiration for the Apple Macintosh, launched with much fanfare in a memorable Super Bowl ad in 1984.

The Objective-C programming language, created by Brad Cox and Tom Love at Stepstone in the early 1980s, added capabilities for object-oriented programming (OOP) to the C programming language. Steve Jobs left Apple in 1985 and founded NeXT Inc. In 1988, NeXT licensed Objective-C from StepStone and developed an Objective-C compiler and libraries which were used as the platform for the NeXTSTEP operating system's user interface, and Interface Builder—used to construct graphical user interfaces.

Jobs returned to Apple in 1996 when Apple bought NeXT. Apple's macOS operating system is a descendant of NeXTSTEP. Apple's proprietary operating system, **iOS**, is derived from Apple's macOS and is used in the iPhone, iPad, iPod Touch, Apple Watch and Apple TV devices. In 2014, Apple introduced its new Swift programming language, which became open source in 2015. The iOS app-development community is shifting from Objective-C to Swift.

1.6.4 Google's Android

Android—the fastest growing mobile and smartphone operating system—is based on the Linux kernel and Java. Android apps can also be developed in 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.[7] 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 textbook, *Android How to Program, Third Edition,* and in our professional 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. Just remember—Facebook, Microsoft and Dell were all launched from college dorm rooms.

1.7 Programming Languages

Figure 1.6 provides brief comments on several popular programming languages. In the next section, we introduce Java.

Programming language	Description
Ada	Ada, based on Pascal, was developed under the sponsorship of the U.S. Department of Defense (DOD) during the 1970s and early 1980s. The DOD wanted a single language that would fill most of its needs. The Pascal-based language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. She's credited with writing the world's first computer program in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). Ada also supports object-oriented programming.
Basic	Basic was developed in the 1960s at Dartmouth College to familiarize novices with programming techniques. Many of its latest versions are object oriented.
C	C was developed in the early 1970s by Dennis Ritchie at Bell Laboratories. It initially became widely known as the UNIX operating system's development language. Today, most code for general-purpose operating systems is written in C or C++.
C++	C++, which is based on C, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides several features that "spruce up" the C language, but more important, it provides capabilities for object-oriented programming.
C#	Microsoft's three primary object-oriented programming languages are C# (based on C++ and Java), Visual C++ (based on C++) and Visual Basic (based on the original Basic). C# was developed to integrate the web into computer applications, and is now widely used to develop enterprise applications and for mobile application development.

Fig. 1.5 | Some other programming languages. (Part 1 of 3.)

7. `https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems`

Programming language	Description
COBOL	COBOL (COmmon Business Oriented Language) was developed in the late 1950s by computer manufacturers, the U.S. government and industrial computer users, based on a language developed by Grace Hopper, a career U.S. Navy officer and computer scientist. (She was posthumously awarded the Presidential Medal of Freedom in November of 2016.) COBOL is still widely used for commercial applications that require precise and efficient manipulation of large amounts of data. Its latest version supports object-oriented programming.
Fortran	Fortran (FORmula TRANslator) was developed by IBM Corporation in the mid-1950s to be used for scientific and engineering applications that require complex mathematical computations. It's still widely used, and its latest versions support object-oriented programming.
JavaScript	JavaScript is the most widely used scripting language. It's primarily used to add programmability to web pages—for example, animations and interactivity with the user. All major web browsers support it.
Objective-C	Objective-C is an object-oriented language based on C. It was developed in the early 1980s and later acquired by NeXT, which in turn was acquired by Apple. It became the key programming language for the OS X operating system and all iOS-powered devices (such as iPods, iPhones and iPads).
Pascal	Research in the 1960s resulted in *structured programming*—a disciplined approach to writing programs that are clearer, easier to test and debug and easier to modify than programs produced with previous techniques. The Pascal language, developed by Professor Niklaus Wirth in 1971, grew out of this research. It was popular for teaching structured programming for several decades.
PHP	PHP is an object-oriented, *open-source* "scripting" language supported by a community of developers and used by numerous websites. PHP is platform independent—implementations exist for all major UNIX, Linux, Mac and Windows operating systems.
Python	Python, another object-oriented scripting language, was released publicly in 1991. Developed by Guido van Rossum of the National Research Institute for Mathematics and Computer Science in Amsterdam, Python draws heavily from Modula-3—a systems programming language. Python is "extensible"—it can be extended through classes and programming interfaces.
Ruby on Rails	Ruby—created in the mid-1990s by Yukihiro Matsumoto—is an open-source, object-oriented programming language with a simple syntax that's similar to Python. Ruby on Rails combines the scripting language Ruby with the Rails web-application framework developed by the company 37Signals. Their book, *Getting Real* (free at `http://gettingreal.37signals.com/toc.php`), is a must-read for web developers. Many Ruby on Rails developers have reported productivity gains over other languages when developing database-intensive web applications.

Fig. 1.5 | Some other programming languages. (Part 2 of 3.)

Programming language	Description
Scala	Scala (`http://www.scala-lang.org/what-is-scala.html`)—short for "scalable language"—was designed by Martin Odersky, a professor at École Polytechnique Fédérale de Lausanne (EPFL) in Switzerland. Released in 2003, Scala uses both the object-oriented programming and functional programming paradigms and is designed to integrate with Java. Programming in Scala can reduce the amount of code in your applications significantly.
Swift	Swift, which was introduced in 2014, is Apple's programming language of the future for developing iOS and OS X applications (apps). Swift is a contemporary language that includes popular programming-language features from languages such as Objective-C, Java, C#, Ruby, Python and others. According to the Tiobe Index, Swift has already become one of the most popular programming languages. Swift is now *open source*, so it can be used on non-Apple platforms as well.
Visual Basic	Microsoft's Visual Basic language was introduced in the early 1990s to simplify the development of Microsoft Windows applications. Its features are comparable to those of C#.

Fig. 1.5 | Some other programming languages. (Part 3 of 3.)

1.8 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 textbook, you'll learn 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.

1.9 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. 1.6). 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.

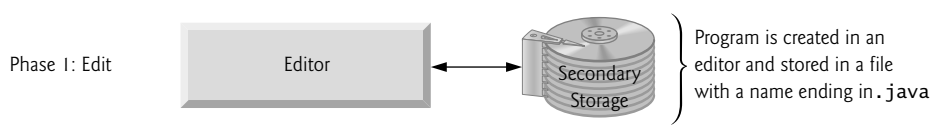


Fig. 1.6 | Typical Java development environment—editing phase.

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/jhtp11LOV
```

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. 1.7). 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 typically 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.

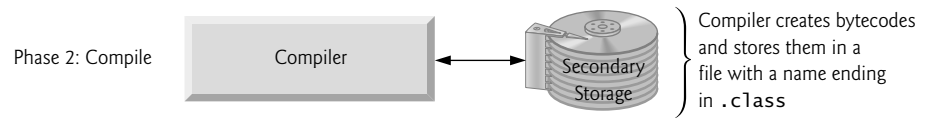


Fig. 1.7 | Typical Java development environment—compilation phase.

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 a software application 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* (that is, dependent on specific computer hardware), bytecode instructions are *platform independent*. So, Java's bytecodes are **portable**—without recompiling the source code, the same bytecode instructions 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. 1.8).The JVM's **class loader** takes the `.class` files containing the program's byte-

codes 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).

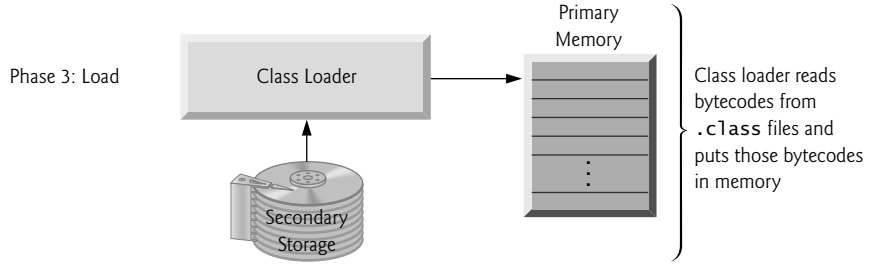


Fig. 1.8 | Typical Java development environment—loading phase.

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. 1.9). 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).

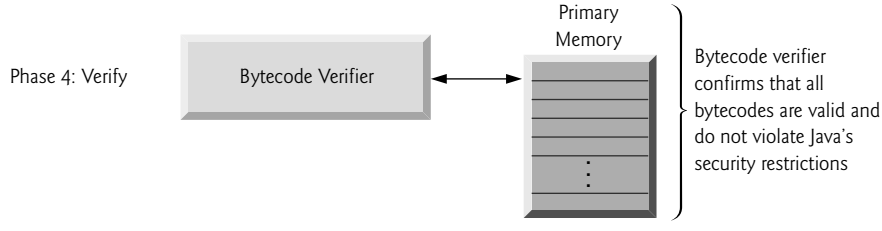


Fig. 1.9 | Typical Java development environment—verification phase.

Phase 5: Execution

In Phase 5, the JVM **executes** the bytecodes to perform the program's specified actions (Fig. 1.10). 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.

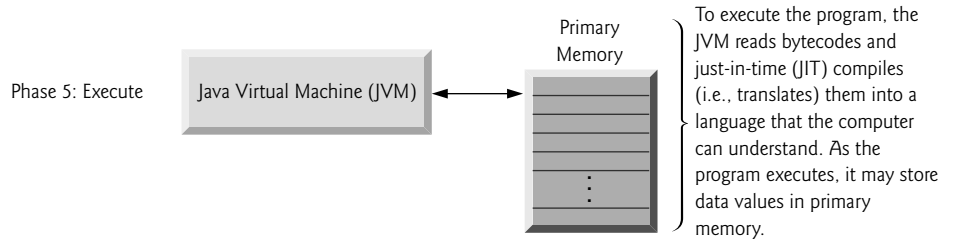


Fig. 1.10 | Typical Java development environment—execution phase.

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.]

Common Programming Error 1.2

Errors such as division by zero occur as a program runs, so they're called ***runtime errors*** *or* ***execution-time errors****.* ***Fatal runtime errors*** *cause programs to terminate immediately without having successfully performed their jobs.* ***Nonfatal runtime errors*** *allow programs to run to completion, often producing incorrect results.*

1.10 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 directory (also called a *folder*) for the **Painter** application:

- On Windows type `cd C:\examples\ch01\Painter`, then press *Enter*.
- On Linux/macOS, type `cd ~/Documents/examples/ch01/Painter`, then press *Enter*.

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 1.9 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 1.11 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.

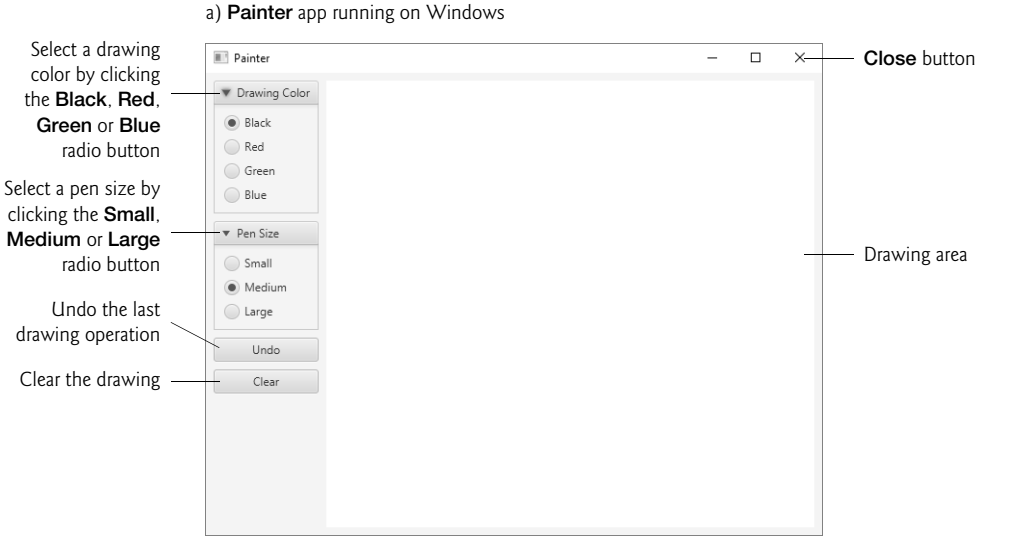
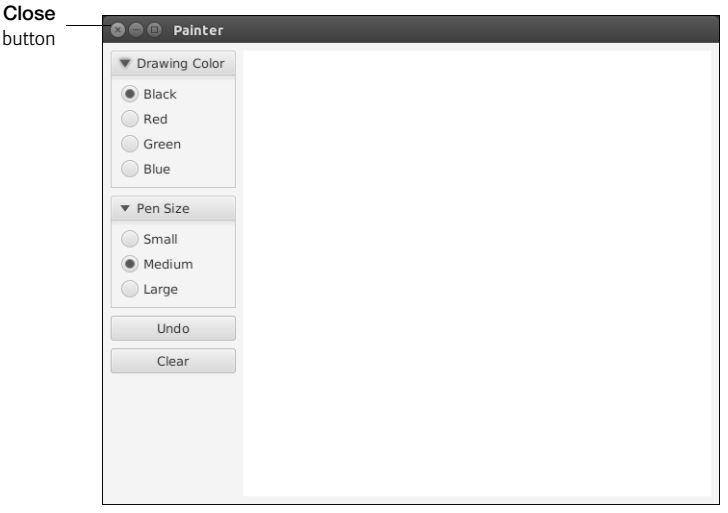


Fig. 1.11 | **Painter** app executing in Windows, Linux and macOS. (Part 1 of 2.)

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



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

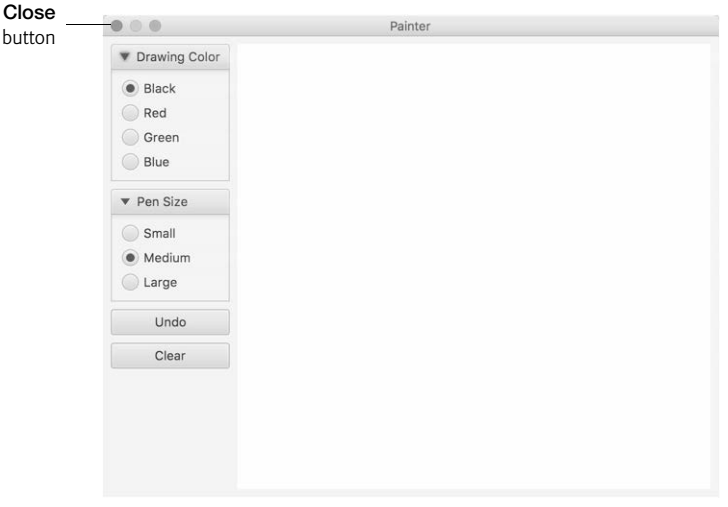


Fig. 1.11 | **Painter** app executing in Windows, Linux and macOS. (Part 2 of 2.)

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. 1.12). 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.

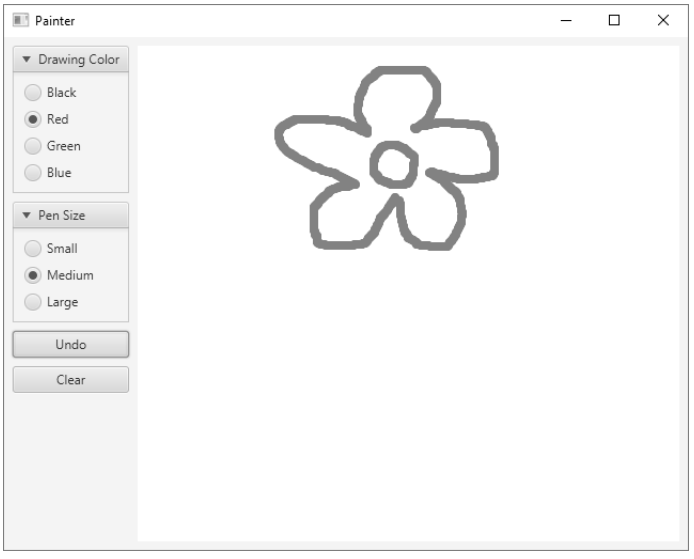


Fig. 1.12 | Drawing the flower petals.

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. 1.13. Next, change the pen size to medium by clicking the **Medium** radio button, then draw the grass as shown in Fig. 1.13.

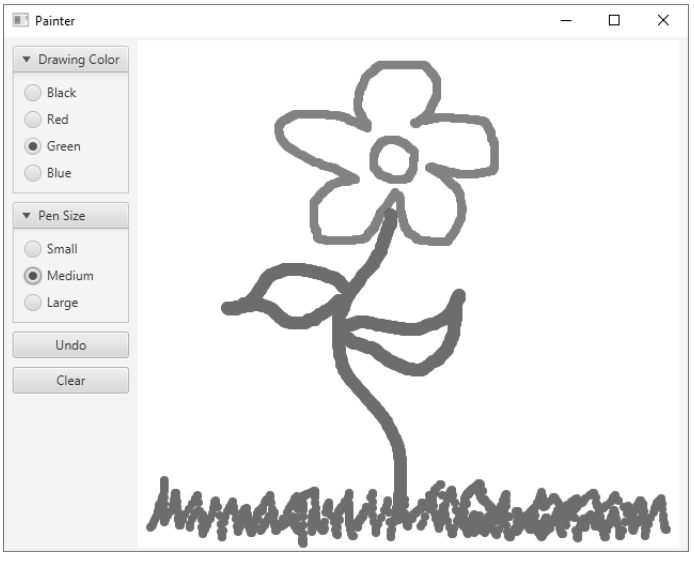


Fig. 1.13 | Drawing the stem and grass.

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. 1.14.

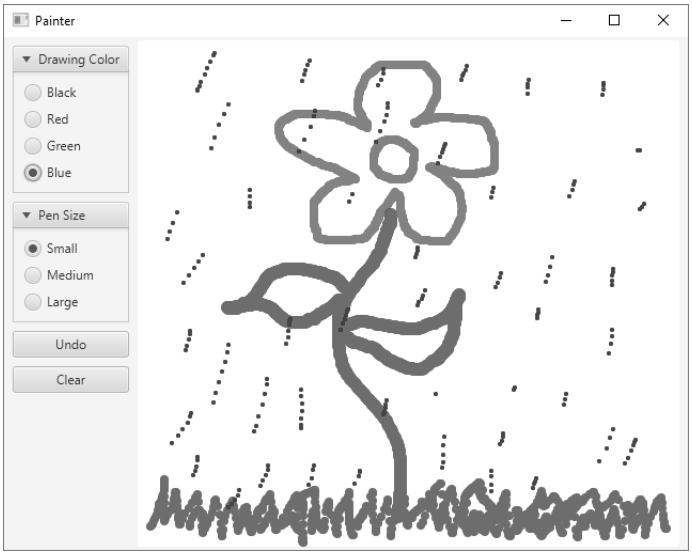


Fig. 1.14 | Drawing the rain.

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. 1.11).

1.11 Internet and World Wide Web

In the late 1960s, ARPA—the Advanced Research Projects Agency of the United States Department of Defense—rolled out plans for networking the main computer systems of approximately a dozen ARPA-funded universities and research institutions. The computers were to be connected with communications lines operating at speeds on the order of 50,000 bits per second, a stunning rate at a time when most people (of the few who even had networking access) were connecting over telephone lines to computers at a rate of 110 bits per second. Academic research was about to take a giant leap forward. ARPA proceeded to implement what quickly became known as the ARPANET, the precursor to today's **Internet**. Today's fastest Internet speeds are on the order of billions of bits per second with trillion-bits-per-second speeds on the horizon!

Things worked out differently from the original plan. Although the ARPANET enabled researchers to network their computers, its main benefit proved to be the capability for quick and easy communication via what came to be known as electronic mail (e-mail). This is true even on today's Internet, with e-mail, instant messaging, file transfer and social media such as Facebook and Twitter enabling billions of people worldwide to communicate quickly and easily.

The protocol (set of rules) for communicating over the ARPANET became known as the **Transmission Control Protocol** (**TCP**). TCP ensured that messages, consisting of sequentially numbered pieces called *packets*, were properly routed from sender to receiver, arrived intact and were assembled in the correct order.

1.11.1 Internet: A Network of Networks

In parallel with the early evolution of the Internet, organizations worldwide were implementing their own networks for both intraorganization (that is, within an organization) and interorganization (that is, between organizations) communication. A huge variety of networking hardware and software appeared. One challenge was to enable these different networks to communicate with each other. ARPA accomplished this by developing the **Internet Protocol** (**IP**), which created a true "network of networks," the current architecture of the Internet. The combined set of protocols is now called **TCP/IP**. Each Internet-connected device has an **IP address**—a unique numerical identifier used by devices communicating via TCP/IP to locate one another on the Internet.

Businesses rapidly realized that by using the Internet, they could improve their operations and offer new and better services to their clients. Companies started spending large amounts of money to develop and enhance their Internet presence. This generated fierce competition among communications carriers and hardware and software suppliers to meet the increased infrastructure demand. As a result, **bandwidth**—the information-carrying capacity of communications lines—on the Internet has increased tremendously, while hardware costs have plummeted.

1.11.2 World Wide Web: Making the Internet User-Friendly

The **World Wide Web** (simply called "the web") is a collection of hardware and software associated with the Internet that allows computer users to locate and view documents (with various combinations of text, graphics, animations, audios and videos) on almost any subject. In 1989, Tim Berners-Lee of CERN (the European Organization for Nuclear Research) began developing **HyperText Markup Language** (**HTML**)—the technology for sharing information via "hyperlinked" text documents. He also wrote communication protocols such as **HyperText Transfer Protocol** (**HTTP**) to form the backbone of his new hypertext information system, which he referred to as the World Wide Web.

In 1994, Berners-Lee founded the **World Wide Web Consortium** (**W3C**, `http://www.w3.org`), devoted to developing web technologies. One of the W3C's primary goals is to make the web universally accessible to everyone regardless of disabilities, language or culture.

1.11.3 Web Services and Mashups

In online Chapter 32, we implement web services (Fig. 1.15). The applications-development methodology of *mashups* enables you to rapidly develop powerful software applications by combining (often free) complementary web services and other forms of information feeds. One of the first mashups combined the real-estate listings provided by `http://www.craigslist.org` with the mapping capabilities of *Google Maps* to offer maps

that showed the locations of homes for sale or rent in a given area. ProgrammableWeb (`http://www.programmableweb.com/`) provides a directory of over 16,500 APIs and 6,300 mashups. Their API University (`https://www.programmableweb.com/api-university`) includes how-to guides and sample code for working with APIs and creating your own mashups. According to their website, some of the most widely used APIs are Facebook, Google Maps, Twitter and YouTube.

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

Fig. 1.15 | Some popular web services (`https://www.programmableweb.com/category/all/apis`).

1.11.4 Internet of Things

The Internet is no longer just a network of computers—it's an **Internet of Things** (**IoT**). A *thing* is any object with an IP address and the ability to send data automatically over the Internet. Such things include:

- a car with a transponder for paying tolls,
- monitors for parking-space availability in a garage,

- a heart monitor implanted in a human,
- monitors for drinkable water quality,
- a smart meter that reports energy usage,
- radiation detectors,
- item trackers in a warehouse,
- mobile apps that can track your movement and location,
- smart thermostats that adjust room temperatures based on weather forecasts and activity in the home
- intelligent home appliances
- and many more.

According to `statista.com`, there are already over 22 billion IoT devices in use today and there are expected to be over 50 billion IoT devices in 2020.[8]

1.12 Software Technologies

Figure 1.16 lists a number of buzzwords that you'll hear in the software development community. We've created Resource Centers on most of these topics, with more on the way.

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.
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 (see online Appendix N, Design Patterns).

Fig. 1.16 | Software technologies. (Part 1 of 2.)

8. `https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/`

Technology	Description
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. 1.16 | Software technologies. (Part 2 of 2.)

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. 1.17).

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. 1.17 | Software product-release terminology.

1.13 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/`

Self-Review Exercises

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

a) Computers process data under the control of sets of instructions called _____.

b) The key logical units of the computer are the _____, _____, _____, _____, _____ and _____.

c) The three types of languages discussed in the chapter are _____, _____ and _____.

d) The programs that translate high-level language programs into machine language are called _____.

e) _____ is an operating system for mobile devices based on the Linux kernel and Java.
f) _____ software is generally feature complete, (supposedly) bug free and ready for use by the community.
g) The Wii Remote, as well as many smartphones, use a(n) _____ which allows the device to respond to motion.

1.2 Fill in the blanks in each of the following sentences about the Java environment:
a) The _____ command from the JDK executes a Java application.
b) The _____ command from the JDK compiles a Java program.
c) A Java source code file must end with the _____ file extension.
d) When a Java program is compiled, the file produced by the compiler ends with the _____ file extension.
e) The file produced by the Java compiler contains _____ that are executed by the Java Virtual Machine.

1.3 Fill in the blanks in each of the following statements (based on Section 1.5):
a) Objects enable the design practice of _____—although they may know how to communicate with one another across well-defined interfaces, they normally are not allowed to know how other objects are implemented.
b) Java programmers concentrate on creating _____, which contain fields and the set of methods that manipulate those fields and provide services to clients.
c) The process of analyzing and designing a system from an object-oriented point of view is called _____.
d) A new class of objects can be created conveniently by _____ —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.
e) _____ is a graphical language that allows people who design software systems to use an industry-standard notation to represent them.
f) The size, shape, color and weight of an object are considered _____ of the object's class.

Answers to Self-Review Exercises

1.1 a) programs. b) input unit, output unit, memory unit, central processing unit, arithmetic and logic unit, secondary storage unit. c) machine languages, assembly languages, high-level languages. d) compilers. e) Android. f) Release candidate. g) accelerometer.

1.2 a) `java`. b) `javac`. c) `.java`. d) `.class`. e) bytecodes.

1.3 a) information hiding. b) classes. c) object-oriented analysis and design (OOAD). d) Inheritance. e) The Unified Modeling Language (UML). f) attributes.

Exercises

1.4 Fill in the blanks in each of the following statements:
a) The logical unit that receives information from outside the computer for use by the computer is the _____.
b) The process of instructing the computer to solve a problem is called _____.
c) _____ is a type of computer language that uses Englishlike abbreviations for machine-language instructions.
d) _____ is a logical unit that sends information which has already been processed by the computer to various devices so that it may be used outside the computer.
e) _____ and _____ are logical units of the computer that retain information.

f) _____ is a logical unit of the computer that performs calculations.
g) _____ is a logical unit of the computer that makes logical decisions.
h) _____ languages are most convenient to the programmer for writing programs quickly and easily.
i) The only language a computer can directly understand is that computer's _____.
j) _____ is a logical unit of the computer that coordinates the activities of all the other logical units.

1.5 Fill in the blanks in each of the following statements:
a) _____ is a platform independent programming language that was built with the objective of allowing programs to be written once and then run on a large variety of electronic devices without modification.
b) _____, _____, and _____ are the names of the three editions of Java that can be used to build different kinds of applications.
c) _____ is the information-carrying capacity of communication lines, and has rapidly increased over the years and become more affordable. Its availability is a cornerstone for building applications that are significantly connected.
d) A(n) _____ is a translator that can convert early assembly-language programs to machine language with reasonable efficiency.

1.6 Fill in the blanks in each of the following statements:
a) Java programs normally go through five phases—_____, _____, _____, _____ and _____.
b) A(n) _____ provides many tools that support the software development process, such as editors for writing and editing programs, debuggers for locating logic errors in programs, and many other features.
c) The command `java` invokes the _____, which executes Java programs.
d) A(n) _____ is a software application that simulates a computer, but hides the underlying operating system and hardware from the programs that interact with it.
e) The _____ takes the `.class` files containing the program's bytecodes and transfers them to primary memory.
f) The _____ examines bytecodes to ensure that they're valid.

1.7 Explain what a just-in-time (JIT) compiler of Java does.

1.8 One of the world's most common objects is a wrist watch. Discuss how each of the following terms and concepts applies to the notion of a watch: object, attributes, behaviors, class, inheritance (consider, for example, an alarm clock), modeling, messages, encapsulation, interface and information hiding.

Making a Difference

The Making-a-Difference exercises will ask you to work on problems that really matter to individuals, communities, countries and the world.

1.9 ***(Test Drive: Carbon Footprint Calculator)*** Some scientists believe that carbon emissions, especially from the burning of fossil fuels, contribute significantly to global warming and that this can be combated if individuals take steps to limit their use of carbon-based fuels. Various organizations and individuals are increasingly concerned about their "carbon footprints." Websites such as TerraPass

```
http://www.terrapass.com/carbon-footprint-calculator-2/
```

and Carbon Footprint

```
http://www.carbonfootprint.com/calculator.aspx
```

provide carbon-footprint calculators. Test drive these calculators to determine your carbon footprint. Exercises in later chapters will ask you to program your own carbon-footprint calculator. To prepare for this, research the formulas for calculating carbon footprints.

1.10 ***(Test Drive: Body-Mass-Index Calculator)*** By recent estimates, two-thirds of the people in the United States are overweight and about half of those are obese. This causes significant increases in illnesses such as diabetes and heart disease. To determine whether a person is overweight or obese, you can use a measure called the body mass index (BMI). The United States Department of Health and Human Services provides a BMI calculator at `http://www.nhlbi.nih.gov/guidelines/obesity/BMI/bmicalc.htm`. Use it to calculate your own BMI. An exercise in Chapter 2 will ask you to program your own BMI calculator. To prepare for this, research the formulas for calculating BMI.

1.11 ***(Attributes of Hybrid Vehicles)*** In this chapter you learned the basics of classes. Now you'll begin "fleshing out" aspects of a class called "Hybrid Vehicle." Hybrid vehicles are becoming increasingly popular, because they often get much better mileage than purely gasoline-powered vehicles. Browse the web and study the features of four or five of today's popular hybrid cars, then list as many of their hybrid-related attributes as you can. For example, common attributes include city-miles-per-gallon and highway-miles-per-gallon. Also list the attributes of the batteries (type, weight, etc.).

1.12 ***(Gender Neutrality)*** Some people want to eliminate sexism in all forms of communication. You've been asked to create a program that can process a paragraph of text and replace gender-specific words with gender-neutral ones. Assuming that you've been given a list of gender-specific words and their gender-neutral replacements (e.g., replace "wife" with "spouse," "man" with "person," "daughter" with "child" and so on), explain the procedure you'd use to read through a paragraph of text and manually perform these replacements. How might your procedure generate a strange term like "woperchild?" In Chapter 3, you'll learn that a more formal term for "procedure" is "algorithm," and that an algorithm specifies the steps to be performed and the order in which to perform them.

1.13 ***(Intelligent Assistants)*** Developments in the field of artificial intelligence have been accelerating in recent years. Many companies now offer computerized intelligent assistants, such as IBM's Watson, Amazon's Alexa, Apple's Siri, Google's Google Now and Microsoft's Cortana. Research these and others and list uses that can improve people's lives.

1.14 ***(Big Data)*** Research the rapidly growing field of big data. List applications that hold great promise in fields such as healthcare and scientific research.

1.15 ***(Internet of Things)*** It's now possible to have a microprocessor at the heart of just about any device and to connect those devices to the Internet. This has led to the notion of the Internet of Things (IoT), which already interconnects tens of billions of devices. Research the IoT and indicate the many ways it's improving people's lives.

Introduction to Java Applications; Input/Output and Operators

2

Objectives

In this chapter you'll:

- Write simple Java applications.
- Use input and output statements.
- Learn about Java's primitive types.
- Understand basic memory concepts.
- Use arithmetic operators.
- Learn 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 learn how to instruct the computer to 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/jhtp11LOV
```

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
```

Fig. 2.1 | Text-printing program. (Part 1 of 2.)

```
Welcome to Java Programming!
```

Fig. 2.1 | Text-printing program. (Part 2 of 2.)

We use line numbers for instructional purposes—they're *not* part of a Java program. This example illustrates several important Java features. We'll see that line 7 does the work—displaying the phrase `"Welcome to Java Programming!"` on the screen.

Commenting Your Programs

We insert **comments** to document programs and improve their readability. The Java compiler *ignores* comments, so they do *not* cause the computer to perform any action when the program is run.

By convention, we begin every program with a comment indicating the figure number and the program's filename. The comment in line 1

```
// Fig. 2.1: Welcome1.java
```

begins with `//`, indicating that it's an **end-of-line comment**—it terminates at the end of the line on which the `//` appears. An end-of-line comment need not begin a line; it also can begin in the middle of a line and continue until the end (as in lines 5, 8 and 9). Line 2,

```
// Text-printing program.
```

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. We demonstrate Javadoc comments and the `javadoc` utility in online Appendix , Creating Documentation with `javadoc`.

Common Programming Error 2.1

Forgetting one of the delimiters of a traditional or Javadoc comment is a syntax error. A ***syntax error*** *occurs when the compiler encounters code that violates Java's language rules (i.e., its syntax). These rules are similar to natural-language grammar rules specifying sentence structure, such as those in English, French, Spanish, etc. Syntax errors are also called* ***compiler errors***, ***compile-time errors*** *or* ***compilation errors****, because the compiler detects them when compiling the program. When a syntax error is encountered, the compiler issues an error message. You must eliminate all compilation errors before your program will compile properly.*

Good Programming Practice 2.1

Some organizations require that every program begin with a comment that states the purpose of the program and the author, date and time when the program was last modified.

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.

Good Programming Practice 2.2

Use white space to enhance program readability.

Declaring a Class

Line 4

```
public class Welcome1 {
```

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 2–6, 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 2.2

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 characters 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 2.3

*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.*

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

9

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 2.4
Indent the entire body of each class declaration one "level" between the braces that delimit the class's. 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.

Good Programming Practice 2.5
IDEs typically indent code for you. The Tab *key may also be used to indent code. You can configure each IDE to specify the number of spaces inserted when you press* Tab.

Common Programming Error 2.3
It's a syntax error if braces do not occur in matching pairs.

Error-Prevention Tip 2.1
When you type an opening left brace, {, immediately type the closing right brace, }, then reposition the cursor between the braces and indent to begin typing the body. This practice helps prevent errors due to missing braces. Many IDEs do this for you.

Declaring a Method
Line 5

```
    // main method begins execution of Java application
```

is a comment indicating the purpose of lines 6–8 of the program. Line 6

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

is the starting point of every Java application. The **parentheses** after the identifier `main` indicate that it's a program building block called a **method**. 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.

Methods perform tasks and can return information when they complete their tasks. We'll explain the purpose of keyword `static` in Section 7.2.5. Keyword **`void`** indicates that this method will *not* return any information. Later, we'll see how a method can return information. For now, simply mimic `main`'s first line in your programs. The `String[] args` in parentheses is a required part of `main`'s declaration—we discuss this in Chapter 6.

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 2.6
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

```
System.out.println("Welcome to Java Programming!");
```

instructs the computer to perform an action—namely, to display 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 characters 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.

Using End-of-Line Comments on Right Braces for Readability
As an aid to programming novices, we include an end-of-line comment after a closing brace that ends a method declaration and after a closing brace that ends a class declaration. For example, line 8

```
    } // end method main
```

indicates the closing brace of method `main`, and line 9

```
} // end class Welcome1
```

indicates the closing brace of class `Welcome1`. Each comment indicates the method or class that the right brace terminates. We'll omit such ending comments after this chapter.

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.

To prepare to compile the program, open a command window and change to the directory where the program is stored. Many operating systems use the command `cd` to change directories (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 2.4

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.

When learning how to program, sometimes it's helpful to "break" a working program to get familiar with the compiler's error messages. These messages do not always state the exact problem in the code. When you encounter an error, it will give you an idea of what caused it. Try removing a semicolon or brace from the program of Fig. 2.1, then recompiling to see the error messages generated by the omission.

Error-Prevention Tip 2.2

When the compiler reports a syntax error, it may not be on the line that the error message indicates. First, check the line for which the error was reported. If you don't find an error on that line, check several preceding lines.

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 2.3

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`.

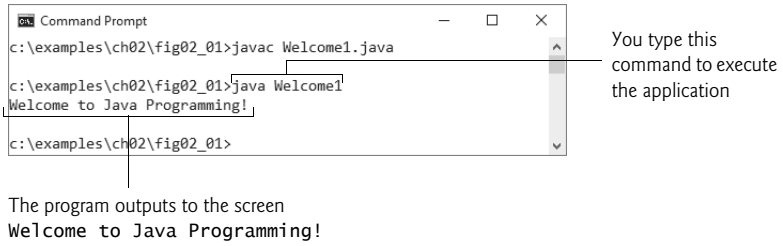


Fig. 2.2 | Executing `Welcome1` from the **Command Prompt**.

2.3 Modifying Your First Java Program

In this section, we 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. [*Note:* From this point forward, we highlight the new and key features in each code listing, as we've done for lines 7–8.]

```
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.

The program is similar to Fig. 2.1, so we discuss only the changes here. Line 2

```
      // Printing a line of text with multiple statements.
```

is an end-of-line comment stating the purpose of the program. Line 4 begins the `Welcome2` class declaration. Lines 7–8 in method `main`

```
      System.out.print("Welcome to ");
      System.out.println("Java Programming!");
```

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).

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

```
System.out.println("Welcome\nto\nJava\nProgramming!");
```

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

```
System.out.printf("%s%n%s%n", "Welcome to", "Java Programming!");
```

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**. Calling a method is also referred to as **invoking** a method.

Good Programming Practice 2.7

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.

Notice that 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. Online Appendix presents more details of formatting output with `printf`.

2.5 Another Application: Adding Integers

Our next application reads (or inputs) two **integers** (whole numbers, such as –22, 7, 0 and 1024) typed by a user at the keyboard, computes their sum and displays it. This program must keep track of the numbers supplied by the user for the calculation later in the program. Programs remember numbers and other data in the computer's memory and access that data through program elements called variables. The program of Fig. 2.7 demonstrates these concepts. In the sample output, we use bold text to identify the user's input (i.e., **45** and **72**). As per our convention in prior programs, lines 1–2 state the figure number, filename and purpose of the program.

```
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
```

Fig. 2.7 | Addition program that inputs two numbers then displays their sum. (Part 1 of 2.)

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

Fig. 2.7 | Addition program that inputs two numbers then displays their sum. (Part 2 of 2.)

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

```
import java.util.Scanner; // program uses class Scanner
```

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 2.5

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 2.6

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

A **variable** is a location in the computer's memory where a value can be stored for use later in a program. All Java variables *must* be declared with a **name** and a **type** *before* they can be used. A variable's *name* enables the program to access the variable's *value* in memory. A variable name can be any valid identifier—again, a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does *not* begin with a digit and does *not* contain spaces. A variable's *type* specifies what kind of information is stored at that location in memory. Like other statements, declaration statements end with a semicolon (;).

Line 9 of `main`

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

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** (i.e., prepared for use in the program) in its declaration with the result of the expression to the right

of the equals sign—`new Scanner(System.in)`. This expression uses the **`new`** keyword to create a `Scanner` object that reads characters typed by the user at the keyboard. The **standard input object**, **`System.in`**, enables applications to read *bytes* of 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 2.8

Choosing meaningful variable names helps a program to be self-documenting (i.e., one can understand the program simply by reading it rather than by reading associated documentation or creating and viewing an excessive number of comments).

Good Programming Practice 2.9

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

```
System.out.print("Enter first integer: "); // prompt
```

uses `System.out.print` to display the message `"Enter first integer: "`. This message is called a **prompt** because it directs the user to take a specific action. We use method `print` here rather than `println` so that the user's input appears on the same line as the prompt. Recall from Section 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

```
int number1 = input.nextInt(); // read first number from user
```

declares that variable `number1` holds data of type **`int`**—that is, *integer* values, which are whole numbers such as `72`, `-1127` and `0`. 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. Real numbers contain decimal points, such as in `3.4`, `0.0` and `-11.19`. 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 indicates that `int` variable `number1` should be initialized in its declaration with the result of `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 program *fault tolerant*.

2.5.5 Obtaining a Second Integer

Line 14

```
      System.out.print("Enter second integer: "); // prompt
```

prompts the user to enter the second integer. Line 15

```
      int number2 = input.nextInt(); // read second number from user
```

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

```
      int sum = number1 + number2; // add numbers then store total in sum
```

declares the `int` variable `sum` and initializes it with the result of `number1 + number2`. When the program encounters the addition operation, it performs the calculation using the values stored in the variables `number1` and `number2`.

In the preceding statement, the addition operator is a **binary operator**, because it has *two* **operands**—`number1` and `number2`. Portions of statements that contain calculations are called **expressions**. In fact, an expression is any portion of a statement that has a *value*. 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.

Good Programming Practice 2.10

Place spaces on either side of a binary operator for readability.

2.5.7 Displaying the Calculation Result

After the calculation has been performed, line 19

```
      System.out.printf("Sum is %d%n", sum); // display sum
```

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 the statement

```
System.out.printf("Sum is %d%n", (number1 + number2));
```

The parentheses around the expression `number1 + number2` are optional—they're included to emphasize that the value of the *entire* expression is output in the position of the `%d` format specifier. Such parentheses are said to be **redundant**.

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
```

Online Appendix shows how to use this documentation.

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 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* (that is, give) `number1` the value entered by the user. You can read this statement as "`number1` *gets* the value of `input.nextInt()`." Everything to the *right* of the assignment operator, `=`, is always evaluated *before* the assignment is performed.

2.6 Memory Concepts

Variable names such as `number1`, `number2` and `sum` actually correspond to *locations* in the computer's memory. Every variable has a **name**, a **type**, a **size** (in bytes) and a **value**.

In the addition program of Fig. 2.7, when the following statement (line 12) executes:

```
int number1 = input.nextInt(); // read first number from user
```

the number typed by the user is placed into a memory location corresponding to the name `number1`. Suppose that the user enters `45`. The computer places that integer value into location `number1` (Fig. 2.8), replacing the previous value (if any) in that location. The previous value is lost, so this process is said to be *destructive*.

Fig. 2.8 | Memory location showing the name and value of variable `number1`.

When the statement (line 15)

```
int number2 = input.nextInt(); // read second number from user
```

executes, suppose that the user enters `72`. The computer places that integer value into location `number2`. The memory now appears as shown in Fig. 2.9.

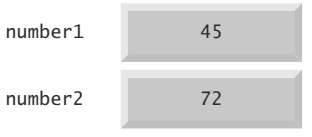


Fig. 2.9 | Memory locations after storing values for `number1` and `number2`.

After the program of Fig. 2.7 obtains values for `number1` and `number2`, it adds the values and places the total into variable `sum`. The statement (line 17)

```
int sum = number1 + number2; // add numbers, then store total in
sum
```

performs the addition, then replaces any previous value in `sum`. After `sum` has been calculated, memory appears as shown in Fig. 2.10. The values of `number1` and `number2` appear exactly as they did before they were used in the calculation of `sum`. These values were used, but *not* destroyed, as the computer performed the calculation. When a value is read from a memory location, the process is *nondestructive*.

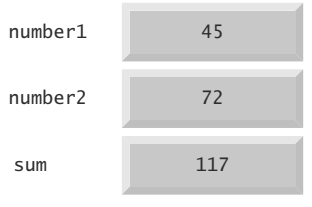


Fig. 2.10 | Memory locations after storing the sum of `number1` and `number2`.

2.7 Arithmetic

Most programs perform arithmetic calculations. The **arithmetic operators** are summarized in Fig. 2.11. Note the use of various special symbols not used in algebra. The **asterisk** (`*`) indicates multiplication, and the percent sign (`%`) is the **remainder operator**, which we'll discuss shortly. The arithmetic operators in Fig. 2.11 are *binary* operators, because each operates on *two* operands. For example, the expression `f + 7` contains the binary operator `+` and the two operands `f` and `7`.

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

Fig. 2.11 | Arithmetic operators.

Integer division yields an integer quotient. For example, the expression `7 / 4` evaluates to `1`, and the expression `17 / 5` evaluates to `3`. Any fractional part in integer division is simply *truncated* (i.e., *discarded*)—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. In this chapter's exercises and in later chapters, we consider several interesting applications of the remainder operator, such as determining whether one number is a multiple of another.

Arithmetic Expressions in Straight-Line Form

Arithmetic expressions in Java must be written in **straight-line form** to facilitate entering programs into computers. Thus, expressions such as "a divided by `b`" must be written as `a / b`, so that all constants, variables and operators appear in a straight line. The following algebraic notation is generally not acceptable to compilers:

$$\frac{a}{b}$$

Parentheses for Grouping Subexpressions

Parentheses are used to group terms in Java expressions in the same manner as in algebraic expressions. For example, to multiply `a` times the quantity `b + c`, we write

```
a * (b + c)
```

If an expression contains **nested parentheses**, such as

```
((a + b) * c)
```

the expression in the *innermost* set of parentheses (`a + b` in this case) is evaluated *first*.

Rules of Operator Precedence

Java applies the 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 2.12 summarizes these rules of operator precedence. A complete precedence chart is included in Appendix .

Operator(s)	Operation(s)	Order of evaluation (precedence)
*	Multiplication	Evaluated first. If there are several operators of this type, they're evaluated from *left to right*.
/	Division	
%	Remainder	
+	Addition	Evaluated next. If there are several operators of this type, they're evaluated from *left to right*.
-	Subtraction	
=	Assignment	Evaluated last.

Fig. 2.12 | Precedence of arithmetic operators.

Sample Algebraic and Java Expressions

Let's consider several sample expressions. Each example shows an algebraic expression and its Java equivalent. The following is an example of an average of five terms:

Algebra: $m = \frac{a + b + c + d + e}{5}$

Java:
```
m = (a + b + c + d + e) / 5;
```

The parentheses are required because division has higher precedence than addition. The entire quantity `(a + b + c + d + e)` is to be divided by `5`. If the parentheses are erroneously omitted, we obtain `a + b + c + d + e / 5`, which evaluates to the different expression

$$a + b + c + d + \frac{e}{5}$$

Here's an example of the equation of a straight line:

Algebra: $y = mx + b$

Java:
```
y = m * x + b;
```

No parentheses are required. The multiplication operator is applied first because multiplication has a higher precedence than addition. The assignment occurs last because it has a lower precedence than multiplication or addition.

The following example contains remainder (%), multiplication, division, addition and subtraction operations:

Algebra: $z = pr\%q + w/x - y$

Java:
```
z = p * r % q + w / x - y;
```

6 1 2 4 3 5

1. We use simple examples to explain the order of evaluation. Subtle order-of-evaluation issues occur in the more complex expressions. For more information, see Chapter 15 of The Java™ Language Specification (`https://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html`).

The circled numbers under the statement indicate the *order* in which Java applies the operators. The *, % and / operations are evaluated first in *left-to-right* order (i.e., they associate from left to right), because they have higher precedence than + and -. The + and - operations are evaluated next. These operations are also applied from *left to right*. The assignment (=) operation is evaluated last.

Evaluation of a Second-Degree Polynomial

To develop a better understanding of the rules of operator precedence, consider the evaluation of an assignment expression that includes a second-degree polynomial $ax^2 + bx + c$:

```
y = a * x * x + b * x + c;
    6   1   2   4   3   5
```

The multiplication operations are evaluated first in left-to-right order (i.e., they associate from left to right), because they have higher precedence than addition. (Java has no arithmetic operator for exponentiation, so x^2 is represented as `x * x`. Section 4.4 shows an alternative for performing exponentiation.) The addition operations are evaluated next from *left to right*. Suppose that `a`, `b`, `c` and `x` are initialized (given values) as follows: `a = 2`, `b = 3`, `c = 7` and `x = 5`. Figure 2.13 illustrates the order in which the operators are applied.

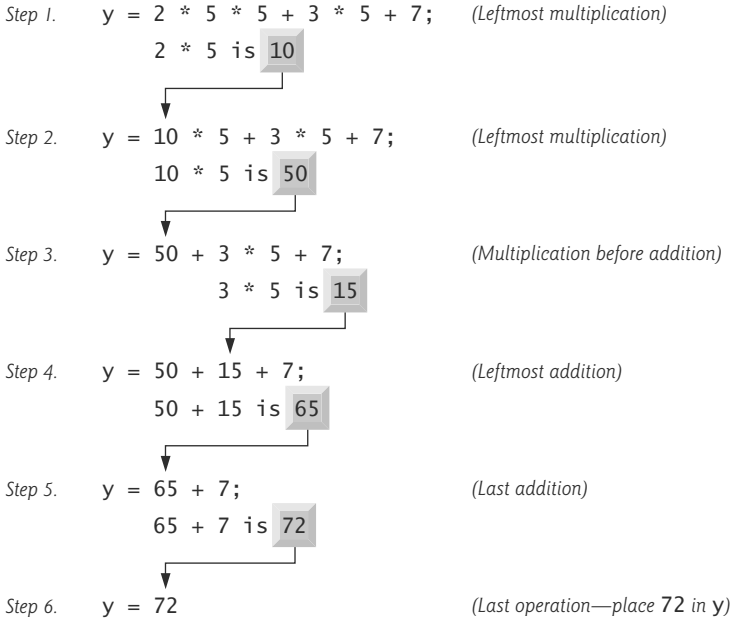


Fig. 2.13 | Order in which a second-degree polynomial is evaluated.

You can use redundant parentheses to make an expression clearer. For example, the preceding statement might be parenthesized as follows:

```
y = (a * x * x) + (b * x) + c;
```

2.8 Decision Making: Equality and Relational Operators

A **condition** is an expression that can be **`true`** or **`false`**. This section introduces Java's **`if` selection statement**, which allows a program to make a **decision** based on a condition's value. For example, the condition "grade is greater than or equal to 60" determines whether a student passed a test. If 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. 2.14. Both equality operators have the same level of precedence, which is *lower* than that of the relational operators. The equality operators associate from *left to right*. The relational operators all have the same level of precedence and also associate from *left to right*.

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. 2.14 | Equality and relational operators.

Figure 2.15 uses six `if` statements to compare two integers input by the user. If the condition in any of these `if` statements is *true*, the statement associated with that `if` statement executes; otherwise, the statement is skipped. We use a `Scanner` to input the integers from the user and store them in variables `number1` and `number2`. The program *compares* the numbers and displays the results of the comparisons that are true. We show three sample outputs for different values entered by the user.

```
1  // Fig. 2.15: 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);
```

Fig. 2.15 | Compare integers using `if` statements, relational operators and equality operators. (Part 1 of 2.)

```
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
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. 2.15 | 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

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

declares `Scanner` variable `input` and assigns it a `Scanner` that inputs data from the standard input (i.e., the keyboard).

Lines 12–13

```
System.out.print("Enter first integer: "); // prompt
int number1 = input.nextInt(); // read first number from user
```

prompt the user to enter the first integer and input the value, respectively. The value is stored in the `int` variable `number1`.

Lines 15–16

```
System.out.print("Enter second integer: "); // prompt
int number2 = input.nextInt(); // read second number from user
```

prompt the user to enter the second integer and input the value, respectively. The value is stored in the `int` variable `number2`.

Lines 18–20

```
if (number1 == number2) {
   System.out.printf("%d == %d%n", number1, number2);
}
```

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

Good Programming Practice 2.11

Indent the statement(s) in the body of an `if` statement to enhance readability. IDEs typically do this for you, allowing you to specify the indent size.

Error-Prevention Tip 2.4

You don't need to use braces, `{ }`, around single-statement bodies, but you must *include the braces around multiple-statement bodies. You'll see later that forgetting to enclose multiple-statement bodies in braces leads to errors. To avoid errors, as a rule, always enclose an `if` statement's body statement(s) in braces.*

Common Programming Error 2.7

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.*

White Space

Note the use of white space in Fig. 2.15. Recall that the compiler normally ignores white space. So, statements may be split over several lines and may be spaced according to your preferences without affecting a program's meaning. It's incorrect to split identifiers and strings. Ideally, statements should be kept small, but this is not always possible.

Error-Prevention Tip 2.5

A lengthy statement can be spread over several lines. If a single statement must be split across lines, choose natural breaking points, such as after a comma in a comma-separated list, or after an operator in a lengthy expression. If a statement is split across two or more lines, indent all subsequent lines until the end of the statement.

Operators Discussed So Far

Figure 2.16 shows the operators discussed so far in decreasing order of precedence. All but the assignment operator, `=`, associate from *left to right*. The assignment operator, `=`, associates from *right to left*. 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`.

Good Programming Practice 2.12

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.

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

Fig. 2.16 | Precedence and associativity of operators discussed so far.

2.9 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. The applications presented here introduced you to many basic programming concepts. In the next chapter we begin our introduction to control statements, which specify the order in which a program's actions are performed. You'll use these in your methods to specify how they should order their tasks.

Summary

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

- A Java application (p. 88) executes when you use the `java` command to launch the JVM.
- Comments (p. 89) document programs and improve their readability. The compiler ignores them.
- An end-of-line comment begins with `//` and terminates at the end of the line on which it appears.
- Traditional comments (p. 89) can be spread over several lines and are delimited by `/*` and `*/`.
- Javadoc comments (p. 89), delimited by `/**` and `*/`, enable you to embed program documentation in your code. The `javadoc` program generates web pages based on these comments.
- A syntax error (p. 89) occurs when the compiler encounters code that violates Java's language rules. It's similar to a grammar error in a natural language.
- Blank lines, space characters and tab characters are known as white space (p. 90). White space makes programs easier to read and is normally ignored by the compiler.
- Keywords (p. 90) are reserved for use by Java and are always spelled with all lowercase letters.
- Keyword `class` (p. 90) introduces a class declaration.
- By convention, all class names in Java begin with a capital letter and capitalize the first letter of each word they include (e.g., `SampleClassName`).
- A Java class name is an identifier—a series of characters consisting of letters, digits, underscores (_) and dollar signs ($) that does not begin with a digit and does not contain spaces.
- A `public` (p. 90) class declaration must be saved in a file with the same name as the class followed by the "`.java`" filename extension.
- Java is case sensitive (p. 90)—that is, uppercase and lowercase letters are distinct.
- The body of every class declaration (p. 91) is delimited by braces, `{` and `}`.
- Method `main` (p. 91) is the starting point of every Java application and must begin with

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

  otherwise, the JVM will not execute the application.
- Methods perform tasks and return information when they complete them. Keyword `void` (p. 91) indicates that a method will perform a task but return no information.
- Statements instruct the computer to perform actions.
- A string (p. 92) in double quotes is sometimes called a character string or a string literal.
- The standard output object (`System.out`; p. 92) displays characters in the command window.
- Method `System.out.println` (p. 92) displays its argument (p. 92) in the command window followed by a newline character to position the output cursor to the beginning of the next line.

Section 2.2.1 Compiling the Application

- You compile a program with the command `javac`. If the program contains no syntax errors, a class file (p. 93) containing the Java bytecodes that represent the application is created. These bytecodes are interpreted by the JVM when you execute the program.

Section 2.2.2 Executing the Application

- To run an application, type `java` followed by the name of the class that contains method `main`.

Section 2.3 Modifying Your First Java Program

- `System.out.print` (p. 94) displays its argument and positions the output cursor immediately after the last character displayed.

- A backslash (\) in a string is an escape character (p. 95). Java combines it with the next character to form an escape sequence (p. 95)—`\n` (p. 95) represents the newline character.

Section 2.4 Displaying Text with `printf`

- `System.out.printf` method (p. 96; `f` means "formatted") displays formatted data.
- Method `printf`'s first argument is a format string (p. 96) containing fixed text and/or format specifiers. Each format specifier (p. 96) indicates the type of data to output and is a placeholder for a corresponding argument that appears after the format string.
- Format specifiers begin with a percent sign (`%`) and are followed by a character that represents the data type. The format specifier `%s` (p. 97) is a placeholder for a string.
- The `%n` format specifier (p. 97) is a portable line separator. 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.

Section 2.5.1 `import` *Declarations*

- An `import` declaration (p. 98) helps the compiler locate a class that's used in a program.
- Java's rich set of predefined classes are grouped into packages (p. 98)—named groups of classes. These are referred to as the Java class library, or the Java Application Programming Interface (Java API; p. 98).

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

- A variable (p. 98) is a location in the computer's memory where a value can be stored for use later in a program. All variables must be declared with a name and a type before they can be used.
- A variable's name enables the program to access the variable's value in memory.
- A `Scanner` (package `java.util`; p. 98) enables a program to read data that the program will use. Before a `Scanner` can be used, the program must create it and specify the source of the data.
- Variables should be initialized (p. 98) to prepare them for use in a program.
- The expression `new Scanner(System.in)` creates a `Scanner` that reads from the standard input object (`System.in`; p. 99)—normally the keyboard.

Section 2.5.3 Prompting the User for Input

- A prompt (p. 99) directs the user to take a specific action.

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

- Data type `int` (p. 99) is used to declare variables that will hold 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 (e.g., `60_000_000`).
- Types `float` and `double` (p. 99) specify real numbers with decimal points, such as `-11.19` and `3.4`.
- Variables of type `char` (p. 99) 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., tab, `\t`).
- Types such as `int`, `float`, `double` and `char` are primitive types (p. 99). Primitive-type names are keywords; thus, they must appear in all lowercase letters.
- `Scanner` method `nextInt` obtains an integer for use in a program.

Section 2.5.6 Using Variables in a Calculation

- Portions of statements that have values are called expressions (p. 100).

Section 2.5.7 Displaying the Calculation Result

- The format specifier `%d` (p. 100) is a placeholder for an `int` value.

Section 2.5.9 Declaring and Initializing Variables in Separate Statements

- A variable must be assigned a value before it's used in a program.
- The assignment operator, `=` (p. 101), enables the program to give a value to a variable.

Section 2.6 Memory Concepts

- Variable names (p. 101) correspond to locations in the computer's memory. Every variable has a name, a type, a size and a value.
- A value that's placed in a memory location replaces the location's previous value, which is lost.

Section 2.7 Arithmetic

- The arithmetic operators (p. 102) are `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division) and `%` (remainder).
- Integer division (p. 103) yields an integer quotient.
- The remainder operator, `%` (p. 103), yields the remainder after division.
- Arithmetic expressions must be written in straight-line form (p. 103).
- If an expression contains nested parentheses (p. 103), the innermost set is evaluated first.
- Java applies the operators in arithmetic expressions in a precise sequence determined by the rules of operator precedence (p. 103).
- When we say that operators are applied from left to right, we're referring to their associativity (p. 104). Some operators associate from right to left.
- Redundant parentheses (p. 105) can make an expression clearer.

Section 2.8 Decision Making: Equality and Relational Operators

- The `if` statement (p. 106) makes a decision based on a condition's value (true or false).
- Conditions in `if` statements can be formed by using the equality (`==` and `!=`) and relational (`>`, `<`, `>=` and `<=`) operators (p. 106).
- An `if` statement begins with keyword `if` followed by a condition in parentheses and expects one statement in its body. You must include braces around multiple-statement bodies.

Self-Review Exercises

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

a) A(n) _____ and a(n) _____ begin and end the body of every method.
b) You can use the _____ statement to make decisions.
c) _____ begins an end-of-line comment.
d) _____, _____ and _____ are called white space.
e) _____ are reserved for use by Java.
f) Java applications begin execution at method _____.
g) Methods _____, _____ and _____ display information in a command window.

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

a) Comments cause the computer to display the text after the `//` on the screen when the program executes.
b) All variables must be given a type when they're declared.
c) Java considers the variables `number` and `NuMbEr` to be identical.

d) The remainder operator (%) can be used only with integer operands.
e) The arithmetic operators *, /, %, + and - all have the same level of precedence.
f) The identifier _ (underscore) is valid in Java 9.

2.3 Write statements to accomplish each of the following tasks:
a) Declare variables `c`, `thisIsAVariable`, `q76354` and `number` to be of type `int` and initialize each to `0`.
b) Prompt the user to enter an integer.
c) Input an integer and assign the result to `int` variable `value`. Assume `Scanner` variable `input` can be used to read a value from the keyboard.
d) Print `"This is a Java program"` on one line in the command window. Use method `System.out.println`.
e) Print `"This is a Java program"` on two lines in the command window. The first line should end with `Java`. Use method `System.out.printf` and two `%s` format specifiers.
f) If the variable `number` is not equal to `7`, display `"The variable number is not equal to 7"`.

2.4 Identify and correct the errors in each of the following statements:
a)
```
if (c < 7); {
   System.out.println("c is less than 7");
}
```
b)
```
if (c => 7) {
   System.out.println("c is equal to or greater than 7");
}
```

2.5 Write declarations, statements or comments that accomplish each of the following tasks:
a) State that a program will calculate the product of three integers.
b) Create a `Scanner` called `input` that reads values from the standard input.
c) Prompt the user to enter the first integer.
d) Read the first integer from the user and store it in the `int` variable `x`.
e) Prompt the user to enter the second integer.
f) Read the second integer from the user and store it in the `int` variable `y`.
g) Prompt the user to enter the third integer.
h) Read the third integer from the user and store it in the `int` variable `z`.
i) Compute the product of the three integers contained in variables `x`, `y` and `z`, and store the result in the `int` variable `result`.
j) Use `System.out.printf` to display the message `"Product is"` followed by the value of the variable `result`.

2.6 Using the statements you wrote in Exercise 2.5, write a complete program that calculates and prints the product of three integers.

Answers to Self-Review Exercises

2.1 a) left brace ({), right brace (}). b) `if`. c) `//`. d) Space characters, newlines and tabs. e) Keywords. f) `main`. g) `System.out.print`, `System.out.println` and `System.out.printf`.

2.2 The answers to Self-Review Exercise 2.2 are:
a) False. Comments do not cause any action to be performed when the program executes. They're used to document programs and improve their readability.
b) True.
c) False. Java is case sensitive, so these variables are distinct.
d) False. The remainder operator can also be used with noninteger operands in Java.
e) False. The operators *, / and % have higher precedence than operators + and -.
f) False. As of Java 9, _ (underscore) by itself is no longer a valid identifier.

2.3 The answers to Self-Review Exercise 2.3 are:

a)
```
int c = 0;
int thisIsAVariable = 0;
int q76354 = 0;
int number = 0;
```

b)
```
System.out.print("Enter an integer: ");
```

c)
```
int value = input.nextInt();
```

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

e)
```
System.out.printf("%s%n%s%n", "This is a Java", "program");
```

f)
```
if (number != 7) {
   System.out.println("The variable number is not equal to 7");
}
```

2.4 The answers to Self-Review Exercise 2.4 are:

a) Error: Semicolon after the right parenthesis of the condition `(c < 7)` in the `if`. As a result, the output statement executes regardless of whether the condition in the `if` is true.
Correction: Remove the semicolon after the right parenthesis.

b) Error: The relational operator `=>` is incorrect.
Correction: Change `=>` to `>=`.

2.5 The answers to Self-Review Exercise 2.5 are:

a)
```
// Calculate the product of three integers
```

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

c)
```
System.out.print("Enter first integer: ");
```

d)
```
int x = input.nextInt();
```

e)
```
System.out.print("Enter second integer: ");
```

f)
```
int y = input.nextInt();
```

g)
```
System.out.print("Enter third integer: ");
```

h)
```
int z = input.nextInt();
```

i)
```
int result = x * y * z;
```

j)
```
System.out.printf("Product is %d%n", result);
```

2.6 The answer to Self-Review Exercise 2.6 is:

```
1  // Ex. 2.6: Product.java
2  // Calculate the product of three integers.
3  import java.util.Scanner; // program uses Scanner
4
5  public class Product {
6     public static void main(String[] args) {
7        // create Scanner to obtain input from command window
8        Scanner input = new Scanner(System.in);
9
10       System.out.print("Enter first integer: "); // prompt for input
11       int x = input.nextInt(); // read first integer
12
13       System.out.print("Enter second integer: "); // prompt for input
14       int y = input.nextInt(); // read second integer
15
16       System.out.print("Enter third integer: "); // prompt for input
17       int z = input.nextInt(); // read third integer
18
19       int result = x * y * z; // calculate product of numbers
20
21       System.out.printf("Product is %d%n", result);
22    } // end method main
23 } // end class Product
```

```
Enter first integer: 10
Enter second integer: 20
Enter third integer: 30
Product is 6000
```

Exercises

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

a) _____ are used to document a program and improve its readability.
b) A decision can be made in a Java program with a(n) _____.
c) The arithmetic operators with the same precedence as multiplication are and _____.
d) When parentheses in an arithmetic expression are nested, the _____ set of parentheses is evaluated first.
e) A location in the computer's memory that may contain different values at various times throughout the execution of a program is called a(n) _____.

2.8 Write Java statements that accomplish each of the following tasks:

a) Display the message "`Enter an integer: `", leaving the cursor on the same line.
b) Assign the product of variables `b` and `c` to the `int` variable `a`.
c) Use a comment to state that a program performs a sample payroll calculation.

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

a) Addition is executed first in the following expression: `a * b / (c + d) * 5`.
b) The following are all valid variable names: `AccountValue`, `$value`, `value_in_$`, `account_no_1234`, `US$`, `her_sales_in_$`, `his_$checking_account`, `X!`, `_$_`, `a@b`, and `_name`.
c) In `2 + 3 + 5 / 4`, addition has the highest precedence.
d) The following are all invalid variable names: `name@email.com`, `87`, `x%`, `99er`, and `2_`.

2.10 Assuming that x = 5 and y = 1, what does each of the following statements display?

a) `System.out.printf("x = %d\n", x + 5);`
b) `System.out.printf("Value of %d * %d is %d\n", x, y, (x * y));`
c) `System.out.printf("x is %d and y is %d", x, y);`
d) `System.out.printf("%d is not equal to %d\n", (x + y), (x * y));`

2.11 Which of the following Jav statements contain variables whose values are not modified?

a) `int m = (p + 2) + 3;`
b) `System.out.println("m = m + 1");`
c) `int m = p / 2;`
d) `int j = k + 2;`

2.12 Given that $y = ax^2 + 5x + 2$, which of the following are correct Java statements for this equation?

a) `y = a * x * x + 5 * x + 2;`
b) `y = a * x * x + (5 * x) + 2;`
c) `y = a * x * x + 5 * (x + 2);`
d) `y = a * (x * x) + 5 * x + 2;`
e) `y = a * x * (x + 5 * x) + 2;`
f) `y = a * (x * x + 5 * x + 2);`

2.13 What is the output that will be printed after execution of the following Java code snippet? Explain why.

```
int p = 5;
System.out.printf("%d", p + 2 * 4);
System.out.printf("%d", p * 2 + 4);
```

2.14 Write an application that displays the numbers 1 to 4 on the same line, with each pair of adjacent numbers separated by one space. Use the following techniques:

a) Use one `System.out.println` statement.
b) Use four `System.out.print` statements.
c) Use one `System.out.printf` statement.

2.15 *(Arithmetic)* Write an application that asks the user to enter two integers, obtains them from the user and prints the square of each, the sum of their squares, and the difference of the squares (first number squared minus the second number squared). Use the techniques shown in Fig. 2.7.

2.16 *(Comparing Integers)* Write an application that asks the user to enter one integer, obtains it from the user and displays whether the number and its square are greater than, equal to, not equal to, or less than the number 100. Use the techniques shown in Fig. 2.15.

2.17 *(Arithmetic, Smallest and Largest)* Write an application that inputs three integers from the user and displays the sum, average, product, smallest and largest of the numbers. Use the techniques shown in Fig. 2.15. [*Note:* The calculation of the average in this exercise should result in an integer representation of the average. So, if the sum of the values is 7, the average should be 2, not 2.3333….]

2.18 *(Displaying Shapes with Asterisks)* Write an application that displays a box, an oval, an arrow and a diamond using asterisks (*), as follows:

```
*********     ***       *         *
*       *   *     *    ***       * *
*       *  *       *  *****     *   *
*       *  *       *    *      *     *
*       *  *       *    *     *       *
*       *  *       *    *      *     *
*       *  *       *    *       *   *
*       *   *     *     *        * *
*********     ***       *         *
```

2.19 What does the following code print?

```
System.out.printf("  ****%n ******%n*******%n ******%n  ****%n");
```

2.20 What does the following code print?

```
System.out.println("*");
System.out.println("***");
System.out.println("*****");
System.out.println("****");
System.out.println("**");
```

2.21 What does the following code print?

```
System.out.print("*");
System.out.print("***");
System.out.print("*****");
System.out.print("****");
System.out.println("**");
```

2.22 What does the following code print?

```
System.out.print("*");
System.out.println("***");
System.out.println("*****");
System.out.print("****");
System.out.println("**");
```

2.23 What does the following code print?

```
System.out.printf("%s%n%s%n%s%n%s%n", "  *", " ***", "*****", " ***", "  *");
```

2.24 ***(Largest and Smallest Integers)*** Write an application that reads five integers and determines and prints the largest and smallest integers in the group. Use only the programming techniques you learned in this chapter.

2.25 ***(Divisible by 3)*** Write an application that reads an integer and determines and prints whether it's divisible by 3 or not. [*Hint*: Use the remainder operator. A number is divisible by 3 if it's divided by 3 with a remainder of 0.]

2.26 ***(Multiples)*** Write an application that reads two integers, determines whether the first number tripled is a multiple of the second number doubled, and prints the result. [*Hint*: Use the remainder operator.]

2.27 ***(Checkerboard Pattern of Asterisks)*** Write an application that displays a checkerboard pattern, as follows:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

2.28 ***(Diameter, Circumference and Area of a Circle)*** Here's a peek ahead. In this chapter, you learned about integers and the type `int`. Java can also represent floating-point numbers that contain decimal points, such as 3.14159. Write an application that inputs from the user the radius of a circle as an integer and prints the circle's diameter, circumference and area using the floating-point value 3.14159 for π. Use the techniques shown in Fig. 2.7. [*Note:* You may also use the predefined constant `Math.PI` for the value of π. This constant is more precise than the value 3.14159. Class `Math` is defined in package `java.lang`. Classes in that package are imported automatically, so you do not need to `import` class `Math` to use it.] Use the following formulas (r is the radius):

$diameter = 2r$
$circumference = 2\pi r$
$area = \pi r^2$

Do not store the results of each calculation in a variable. Rather, specify each calculation as the value that will be output in a `System.out.printf` statement. The values produced by the circumference and area calculations are floating-point numbers. Such values can be output with the format specifier `%f` in a `System.out.printf` statement. You'll learn more about floating-point numbers in Chapter 3.

2.29 ***(Integer Value of a Character)*** Here's another peek ahead. In this chapter, you learned about integers and the type `int`. Java can also represent uppercase letters, lowercase letters and a considerable variety of special symbols. Every character has a corresponding integer representation. The set of characters a computer uses together with the corresponding integer representations for those characters is called that computer's character set. You can indicate a character value in a program simply by enclosing that character in single quotes, as in `'A'`.

You can determine a character's integer equivalent by preceding that character with `(int)`, as in

```
(int) 'A'
```

An operator of this form is called a cast operator. (You'll learn about cast operators in Chapter 3.) The following statement outputs a character and its integer equivalent:

```
System.out.printf("The character %c has the value %d%n", 'A', ((int) 'A'));
```

When the preceding statement executes, it displays the character `A` and the value `65` (from the Unicode® character set) as part of the string. The format specifier `%c` is a placeholder for a character (in this case, the character `'A'`).

Using statements similar to the one shown earlier in this exercise, write an application that displays the integer equivalents of some uppercase letters, lowercase letters, digits and special symbols. Display the integer equivalents of the following: `A B C a b c 0 1 2 $ * + /` and the blank character.

2.30 ***(Separating the Digits in an Integer)*** Write an application that inputs one number consisting of five digits from the user, separates the number into its individual digits and prints the digits separated from one another by three spaces each. For example, if the user types in the number `42339`, the program should print

```
4   2   3   3   9
```

Assume that the user enters the correct number of digits. What happens when you enter a number with more than five digits? What happens when you enter a number with fewer than five digits? [*Hint:* It's possible to do this exercise with the techniques you learned in this chapter. You'll need to use both division and remainder operations to "pick off" each digit.]

2.31 ***(Table of Squares and Cubes)*** Using only the programming techniques you learned in this chapter, write an application that calculates the squares and cubes of the numbers from 0 to 10 and prints the resulting values in table format, as shown below.

```
number  square  cube
0       0       0
1       1       1
2       4       8
3       9       27
4       16      64
5       25      125
6       36      216
7       49      343
8       64      512
9       81      729
10      100     1000
```

2.32 ***(Negative, Positive and Zero Values)*** Write a program that inputs five numbers and determines and prints the number of negative numbers input, the number of positive numbers input and the number of zeros input.

Making a Difference

2.33 ***(Body Mass Index Calculator)*** We introduced the body mass index (BMI) calculator in Exercise 1.10. The formulas for calculating BMI are

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

or

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

Create a BMI calculator that reads the user's weight in pounds and height in inches (or, if you prefer, the user's weight in kilograms and height in meters), then calculates and displays the user's body mass index. Also, display the BMI categories and their values from the National Heart Lung and Blood Institute

`http://www.nhlbi.nih.gov/health/educational/lose_wt/BMI/bmicalc.htm`

so the user can evaluate his/her BMI.

[*Note:* In this chapter, you learned to use the `int` type to represent whole numbers. The BMI calculations when done with `int` values will both produce whole-number results. In Chapter 7 you'll learn to use the `double` type to represent numbers with decimal points. When the BMI calculations are performed with `double`s, they'll both produce numbers with decimal points—these are called "floating-point" numbers.]

2.34 ***(World Population Growth Calculator)*** Search the Internet to determine the current world population and the annual world population growth rate. Write an application that inputs these values, then displays the estimated world population after one, two, three, four and five years.

2.35 ***(Statistics for the Great Pyramid of Giza)*** The Great Pyramid of Giza is considered an engineering marvel of its time. Use the web to get statistics related to the Great Pyramid of Giza, and find the estimated number of stones used to build it, the average weight of each stone, and the number of years it took to build. Create an application that calculates an estimate of how much, by weight, of the pyramid was built each year, each hour, and each minute as it was being built. The application should input the following information:

a) Estimated number of stones used.
b) Average weight of each stone.
c) Number of years taken to build the pyramid (assuming a year comprises 365 days).

3

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

Objectives

In this chapter you'll:

- Learn basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- Use the `if` and `if...else` selection statements to choose 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.
- Learn about the portability of primitive data types.

3.1 Introduction

Before writing a program to solve a problem, you should have a thorough understanding of the problem and a carefully planned approach to solving it. When writing a program, you also should understand the available building blocks and employ proven program-construction techniques. In this chapter and the next, we discuss these issues in presenting the theory and principles of structured programming. The concepts presented here are crucial in building classes and manipulating objects. We discuss Java's `if` statement in additional detail and introduce the `if...else` and `while` statements—all of these building blocks allow you to specify the logic required for methods to perform their tasks. We also introduce the compound assignment operator and the increment and decrement operators. Finally, we consider the portability of Java's primitive types.

3.2 Algorithms

Any computing problem can be solved by executing a series of actions in a specific order. A *procedure* for solving a problem in terms of

1. the **actions** to execute and
2. the **order** in which these actions execute

is called an **algorithm**. The following example demonstrates that correctly specifying the order in which the actions execute is important.

Consider the "rise-and-shine algorithm" followed by one executive for getting out of bed and going to work: (1) Get out of bed; (2) take off pajamas; (3) take a shower; (4) get dressed; (5) eat breakfast; (6) carpool to work. This routine gets the executive to work well prepared to make critical decisions. Suppose that the same steps are performed in a slightly different order: (1) Get out of bed; (2) take off pajamas; (3) get dressed; (4) take a shower; (5) eat breakfast; (6) carpool to work. In this case, our executive shows up for work soaking wet. Specifying the order in which statements (actions) execute in a program is called **program control**. This chapter investigates program control using Java's **control statements**.

3.3 Pseudocode

Pseudocode is an informal language that helps you develop algorithms without having to worry about the strict details of Java language syntax. The pseudocode we present is particularly useful for developing algorithms that will be converted to structured portions of Java programs. The pseudocode we use in this book is similar to everyday English—it's convenient and user friendly, but it's not an actual computer programming language. You'll see an algorithm written in pseudocode in Fig. 3.5. You may, of course, use your own native language(s) to develop your own pseudocode.

Pseudocode does not execute on computers. Rather, it helps you "think out" a program before attempting to write it in a programming language, such as Java. This chapter provides several examples of using pseudocode to develop Java programs.

The style of pseudocode we present consists purely of characters, so you can type pseudocode conveniently, using any text-editor program. A carefully prepared pseudocode program can easily be converted to a corresponding Java program.

Pseudocode normally describes only statements representing the *actions* that occur after you convert a program from pseudocode to Java and the program is run on a computer. Such actions might include *input*, *output* or *calculations*. In our pseudocode, we typically do *not* include variable declarations, but some programmers choose to list variables and mention their purposes.

3.4 Control Structures

Normally, statements in a program are executed one after the other in the order in which they're written. This process is called **sequential execution**. Various Java statements, which we'll soon discuss, enable you to specify that the next statement to execute is *not* necessarily the *next* one in sequence. This is called **transfer of control**.

During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of much difficulty experienced by software development groups. The blame was pointed at the **`goto` statement** (used in most programming languages of the time), which allows you to specify a transfer of control to one of a wide range of destinations in a program. [*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.]

The research of Bohm and Jacopini[1] demonstrated that programs could be written *without* any `goto` statements. The challenge of the era for programmers was to shift their styles to "`goto`-less programming." The term **structured programming** became almost synonymous with "`goto` elimination." Not until the 1970s did most programmers start taking structured programming seriously. The results were impressive. Software development groups reported shorter development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects. The key to these successes was that structured programs were clearer, easier to debug and modify, and more likely to be bug free in the first place.

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

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.

iteration structure. When we introduce Java's control-structure implementations, we'll refer to them in the terminology of the *Java Language Specification* as "control statements."

3.4.1 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. 3.1 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.

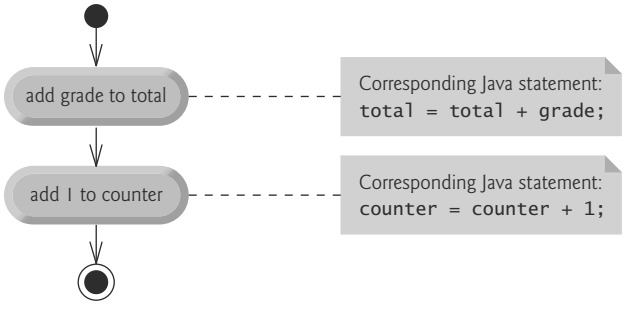


Fig. 3.1 | Sequence-structure activity diagram.

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. 3.1. 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.

Like pseudocode, activity diagrams help you develop and represent algorithms. Activity diagrams clearly show how control structures operate. We use the UML in this chapter and Chapter 4 to show control flow in control statements. Online Chapters 33–34 use the UML in a real-world automated-teller-machine case study.

Consider the activity diagram in Fig. 3.1. 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. 3.1 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 3.1 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