Joyce Farrell

Java Programming

Ninth Edition

NINTH EDITION

JAVA™ PROGRAMMING

JOYCE FARRELL

CENGAGE

Australia • Brazil • Mexico • Singapore • United Kingdom • United States

CENGAGE

***Java™ Programming*, Ninth Edition**
Joyce Farrell

SVP, GM Skills: Jonathan Lau

Product Team Manager: Kristin McNary

Associate Product Manager:
Kate Mason

Executive Director of Content Design,
Skills: Marah Bellegarde

Director, Learning Design –
Skills Computing: Leigh Hefferon

Learning Designer:
Natalie Onderdonk

Product Assistant: Jake Toth

Marketing Director: Michele McTighe

Marketing Manager: Stephanie Albracht

Content Project Manager:
Michele Stulga

Senior Designer: Diana Graham

Production Service/Composition:
SPi Global

Cover image: Colormos/Photodisc
/Getty Images

For product information and technology assistance, contact us at
Cengage Customer & Sales Support, 1-800-354-9706
or support.cengage.com.

For permission to use material from this text or product, submit all requests online at **www.cengage.com/permissions.**

Library of Congress Control Number: 2018933919

Softbound ISBN: 978-1-337-39707-0

Loose Leaf ISBN: 978-1-337-68590-0

Cengage
20 Channel Center Street
Boston, MA 02210
USA

Cengage is a leading provider of customized learning solutions with employees residing in nearly 40 different countries and sales in more than 125 countries around the world. Find your local representative at **www.cengage.com.**

Cengage products are represented in Canada by Nelson Education, Ltd.

To learn more about Cengage platforms and services, visit **www.cengage.com.**

To register or access your online learning solution or purchase materials for your course, visit **www.cengagebrain.com.**

Notice to the Reader

Publisher does not warrant or guarantee any of the products described herein or perform any independent analysis in connection with any of the product information contained herein. Publisher does not assume, and expressly disclaims, any obligation to obtain and include information other than that provided to it by the manufacturer. The reader is expressly warned to consider and adopt all safety precautions that might be indicated by the activities described herein and to avoid all potential hazards. By following the instructions contained herein, the reader willingly assumes all risks in connection with such instructions. The publisher makes no representations or warranties of any kind, including but not limited to, the warranties of fitness for particular purpose or merchantability, nor are any such representations implied with respect to the material set forth herein, and the publisher takes no responsibility with respect to such material. The publisher shall not be liable for any special, consequential, or exemplary damages resulting, in whole or part, from the readers' use of, or reliance upon, this material.

Printed in the United States of America
Print Number: 01 Print Year: 2018

Brief Contents

Table of Contents

Preface

Java Programming, Ninth Edition, provides the beginning programmer with a guide to developing applications using the Java programming language. Java is popular among professional programmers because it can be used to build visually interesting graphical user interface (GUI) and Web-based applications. Java also provides an excellent environment for the beginning programmer—a student can quickly build useful programs while learning the basics of structured and object-oriented programming techniques.

This textbook assumes that you have little or no programming experience. It provides a solid background in good object-oriented programming techniques and introduces terminology using clear, familiar language. The programming examples are business examples; they do not assume a mathematical background beyond high school business math. In addition, the examples illustrate only one or two major points; they do not contain so many features that you become lost following irrelevant and extraneous details. Complete, working programs appear frequently in each chapter; these examples help students make the transition from the theoretical to the practical. The code presented in each chapter also can be downloaded from the publisher's website, so students easily can run the programs and experiment with changes to them.

The student using *Java Programming, Ninth Edition*, builds applications from the bottom up rather than starting with existing objects. This facilitates a deeper understanding of the concepts used in object-oriented programming and engenders appreciation for the existing objects students use as their knowledge of the language advances. When students complete this book, they will know how to modify and create simple Java programs, and they will have the tools to create more complex examples. They also will have a fundamental knowledge about object-oriented programming, which will serve them well in advanced Java courses or in studying other object-oriented languages such as C++, C#, and Visual Basic.

Organization and Coverage

Java Programming, Ninth Edition, presents Java programming concepts, enforcing good style, logical thinking, and the object-oriented paradigm. Objects are covered right from the beginning, earlier than in many other textbooks. You create your first Java program in Chapter 1. Chapters 2, 3, and 4 increase your understanding about how data, classes, objects, and methods interact in an object-oriented environment.

Chapters 5 and 6 explore input and repetition structures, which are the backbone of programming logic and essential to creating useful programs in any language. You learn the special considerations of string and array manipulation in Chapters 7, 8, and 9.

Chapters 10, 11, and 12 thoroughly cover inheritance and exception handling. Inheritance is the object-oriented concept that allows you to develop new objects quickly by adapting the features of existing objects; exception handling is the object-oriented approach to handling errors. Both are important concepts in object-oriented design. Chapter 13 provides information about handling files so you can store and retrieve program output.

Chapter 14 introduces GUI Swing components, which are used to create visually pleasing, user-friendly, interactive applications.

Chapter 15 introduces JavaFX, which is the newest platform for creating and delivering applications for the desktop and the Internet. Chapter 15 is written by Sandra Lavallee, a professor and Computer and Design Technologies Department chairperson at Lakes Region Community College in Laconia, New Hampshire.

New in This Edition

The following features are new for the Ninth Edition:

- **Java 9e:** All programs have been tested using Java 9e, the newest edition of Java.
- **Windows 10:** All programs have been tested in Windows 10, and all screen shots have been taken in this environment.
- **Programming exercises:** Each chapter contains several new programming exercises not seen in previous editions. All exercises and their solutions from the previous edition that were replaced in this edition are still available on the Instructor Companion site.
- **Anonymous inner classes and lambda expressions:** These two new topics are introduced in this edition of the book.
- **JavaFX:** This edition includes coverage of JavaFX.

Additionally, *Java Programming, Ninth Edition*, includes the following features:

- **OBJECTIVES:** Each chapter begins with a list of objectives so you know the topics that will be presented in the chapter. In addition to providing a quick reference to topics covered, this feature provides a useful study aid.
- **YOU DO IT:** In each chapter, step-by-step exercises help students create multiple working programs that emphasize the logic a programmer uses in choosing statements to include. These sections provide a means for students to achieve success on their own—even those in online or distance learning classes.
- **NOTES:** These highlighted tips provide additional information—for example, an alternative method of performing a procedure, another term for a concept, background information about a technique, or a common error to avoid.
- **EMPHASIS ON STUDENT RESEARCH:** The student frequently is directed to the Java website to investigate classes and methods. Computer languages evolve, and programming professionals must understand how to find the latest language improvements. This book encourages independent research.

- **FIGURES:** Each chapter contains many figures. Code figures are most frequently 25 lines or fewer, illustrating one concept at a time. Frequent screen shots show exactly how program output appears. Callouts appear where needed to emphasize a point.
- **COLOR:** The code figures in each chapter contain all Java keywords in blue. This helps students identify keywords more easily, distinguishing them from programmer-selected names.
- **FILES:** More than 200 student files can be downloaded from the publisher's website. Most files contain the code presented in the figures in each chapter; students can run the code for themselves, view the output, and make changes to the code to observe the effects. Other files include debugging exercises that help students improve their programming skills.
- **TWO TRUTHS & A LIE:** A short quiz reviews each chapter section, with answers provided. This quiz contains three statements based on the preceding section of text—two statements are true, and one is false. Over the years, students have requested answers to problems, but we have hesitated to distribute them in case instructors want to use problems as assignments or test questions. These true-false quizzes provide students with immediate feedback as they read, without "giving away" answers to the multiple-choice questions and programming exercises.
- **DON'T DO IT:** This section at the end of each chapter summarizes common mistakes and pitfalls that plague new programmers while learning the current topic.
- **KEY TERMS:** Each chapter includes a list of newly introduced vocabulary, shown in the order of appearance in the text. The list of key terms provides a short review of the major concepts in the chapter.
- **SUMMARIES:** Following each chapter is a summary that recaps the programming concepts and techniques covered in the chapter. This feature provides a concise means for students to check their understanding of the main points in each chapter.
- **REVIEW QUESTIONS:** Each chapter includes 20 multiple-choice questions that serve as a review of chapter topics.
- **GAME ZONE:** Each chapter provides one or more exercises in which students can create interactive games using the programming techniques learned up to that point; 50 game programs are suggested in the book. The games are fun to create and play; writing them motivates students to master the necessary programming techniques. Students might exchange completed game programs with each other, suggesting improvements and discovering alternate ways to accomplish tasks.
- **CASES:** Each chapter contains two running case problems. These cases represent projects that continue to grow throughout a semester using concepts learned in each new chapter. Two cases allow instructors to assign different cases in alternate semesters or to divide students in a class into two case teams.
- **GLOSSARY:** A glossary contains definitions for all key terms in the book.

- **APPENDICES:** This edition includes useful appendices on working with the Java platform, data representation, formatting output, generating random numbers, and creating Javadoc comments.
- **QUALITY:** Every program example, exercise, and game solution was tested by the author and then tested again by a quality assurance team using Java Standard Edition (SE) 9, the most recent version available.

Instructor Resources

MindTap

MindTap activities for *Java Programming, Ninth Edition* are designed to help students master the skills they need in today's workforce. Research shows employers need critical thinkers, troubleshooters, and creative problem-solvers to stay relevant in our fast-paced, technology-driven world. MindTap helps you achieve this with assignments and activities that provide hands-on practice and real-life relevance. Students are guided through assignments that help them master basic knowledge and understanding before moving on to more challenging problems.

All MindTap activities and assignments are tied to defined unit learning objectives. Hands-on coding labs provide real-life application and practice. Readings and dynamic visualizations support the lecture, while a post-course assessment measures exactly how much a student has learned. MindTap provides the analytics and reporting to easily see where the class stands in terms of progress, engagement, and completion rates. Use the content and learning path as-is, or pick-and-choose how our materials will wrap around yours. You control what the students see and when they see it. Learn more at *http://www.cengage.com/mindtap/*.

The *Java Programming* MindTap also includes:

- **Unit Quizzes:** Students apply what they have learned in each unit by taking the quizzes provided in the learning path.
- **Video Lessons:** Each unit is accompanied by video lessons that help to explain important unit concepts. These videos were created and narrated by the author.
- **Interactive Study Aids:** Flashcards and crossword puzzles help users review main concepts from the units and coding Snippets allow students to practice key coding concepts.

Instructor Companion Site

The following teaching tools are available for download at the Companion Site for this text. Simply search for this text at *www.cengagebrain.com* and choose "Instructor Downloads." An instructor login is required.

- **Instructor's Manual:** The Instructor's Manual that accompanies this textbook includes additional instructional material to assist in class preparation, including items such as Overviews, Chapter Objectives, Teaching Tips, Quick Quizzes, Class Discussion Topics, Additional Projects, Additional Resources, and Key Terms. A sample syllabus also is available.
- **Test Bank:** Cengage Testing Powered by Cognero is a flexible, online system that allows you to:
    - Author, edit, and manage test bank content from multiple Cengage solutions.
    - Create multiple test versions in an instant.
    - Deliver tests from your LMS, your classroom, or wherever you want.
- **PowerPoint Presentations:** This text provides PowerPoint slides to accompany each chapter. Slides can be used to guide classroom presentations, to make available to students for chapter review, or to print as classroom handouts.
- **Student Files:** Files are provided for every figure in the text. Instructors can use the files to customize PowerPoint slides, illustrate quizzes, or create handouts.
- **Solutions:** Solutions to all programming exercises are available. If an input file is needed to run a programming exercise, it is included with the solution file.
- **Data Files:** Data files necessary to complete the steps and projects in the book are available at *www.cengagebrain.com*, or your instructor will provide the data files to you.

Acknowledgments

I would like to thank all of the people who helped to make this book a reality, including Natalie Onderdonk, Learning Designer; Michele Stulga, Content Project Manager; and John Freitas, Quality Assurance Tester. I am lucky to work with these professionals who are dedicated to producing high-quality instructional materials.

I am also grateful to the reviewers who provided comments and encouragement during this book's development, including Cliff Brozo, Monroe College; Fred D'Angelo, University of Arizona; Cassandra Henderson, Albany Technical College; Zack Hubbard, Rowan-Cabarrus Community College; and Sandra Lavallee, Lakes Region Community College.

Thanks, too, to my husband, Geoff, for his constant support, advice, and encouragement. Finally, this book is dedicated to George Edward Farrell Peterson and Clifford Geoffrey Farrell Peterson. You each had a book dedicated to you earlier, but those books were published before I knew your names. Now you are here, and I love you!

Joyce Farrell

Read This Before You Begin

The following information will help you as you prepare to use this textbook.

To the User of the Data Files

To complete the steps and projects in this book, you need data files that have been created specifically for this book. Your instructor will provide the data files to you. You also can obtain the files electronically from *www.CengageBrain.com*. Find the ISBN of your title on the back cover of your book, then enter the ISBN in the search box at the top of the Cengage Brain home page. You can find the data files on the product page that opens. Note that you can use a computer in your school lab or your own computer to complete the exercises in this book.

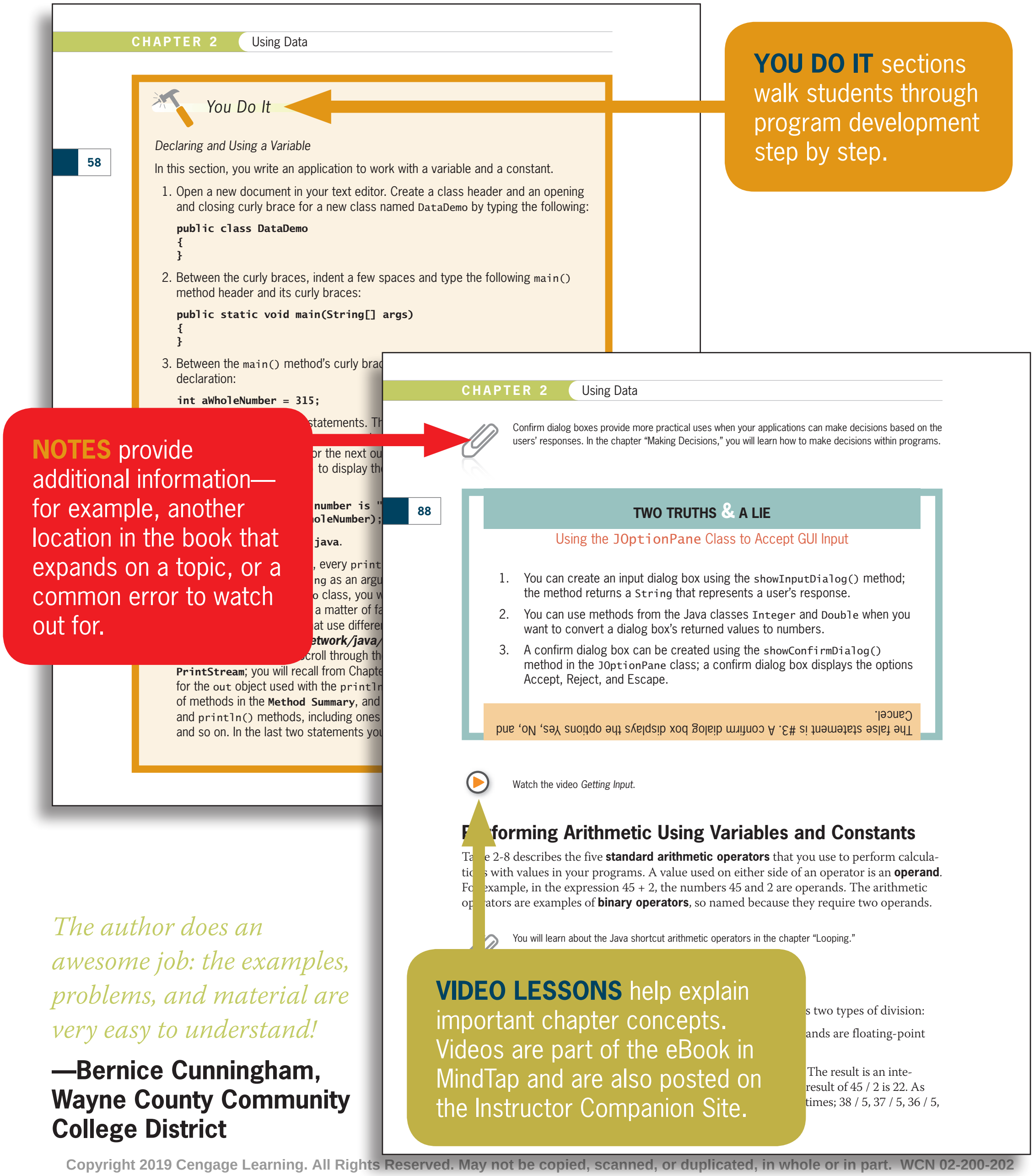
Using Your Own Computer

To use your own computer to complete the steps and exercises, you need the following:

- **Software:** Java SE 9, available from *www.oracle.com/technetwork/java/index.html*. Although almost all of the examples in this book will work with earlier versions of Java, this book was created using Java 9e. You also need a text editor, such as Notepad. A few exercises ask you to use a browser for research. Chapter 15 uses NetBeans to develop JavaFX programs; you can downoad this software from *Https:netbens.org*.
- **Hardware:** For operating system requirements (memory and disk space), see *http://java.com/en/download/help*.

Features

This text focuses on helping students become better programmers and understand Java program development through a variety of key features. In addition to Chapter Objectives, Summaries, and Key Terms, these useful features will help students regardless of their learning styles.



CHAPTER 2 Using Data

58

You Do It

Declaring and Using a Variable

In this section, you write an application to work with a variable and a constant.

1. Open a new document in your text editor. Create a class header and an opening and closing curly brace for a new class named `DataDemo` by typing the following:

```
public class DataDemo
{
}
```

2. Between the curly braces, indent a few spaces and type the following `main()` method header and its curly braces:

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

3. Between the `main()` method's curly brac declaration:

```
int aWholeNumber = 315;
```

statements. Th

or the next ou
to display th

```
number is "
oleNumber);
```

java.

, every `print`
ng as an argu
o class, you w
a matter of fa
at use differe
etwork/java/
croll through th
`PrintStream`; you will recall from Chapte
for the `out` object used with the `printl`
of methods in the **`Method Summary`**, and
and `println()` methods, including ones
and so on. In the last two statements you

YOU DO IT sections walk students through program development step by step.

NOTES provide additional information—for example, another location in the book that expands on a topic, or a common error to watch out for.

CHAPTER 2 Using Data

Confirm dialog boxes provide more practical uses when your applications can make decisions based on the users' responses. In the chapter "Making Decisions," you will learn how to make decisions within programs.

88

TWO TRUTHS & A LIE

Using the `JOptionPane` Class to Accept GUI Input

1. You can create an input dialog box using the `showInputDialog()` method; the method returns a `String` that represents a user's response.
2. You can use methods from the Java classes `Integer` and `Double` when you want to convert a dialog box's returned values to numbers.
3. A confirm dialog box can be created using the `showConfirmDialog()` method in the `JOptionPane` class; a confirm dialog box displays the options Accept, Reject, and Escape.

The false statement is #3. A confirm dialog box displays the options Yes, No, and Cancel.

Watch the video *Getting Input.*

P forming Arithmetic Using Variables and Constants

Ta e 2-8 describes the five **standard arithmetic operators** that you use to perform calcula- tic s with values in your programs. A value used on either side of an operator is an **operand**. Fo example, in the expression 45 + 2, the numbers 45 and 2 are operands. The arithmetic op ators are examples of **binary operators**, so named because they require two operands.

You will learn about the Java shortcut arithmetic operators in the chapter "Looping."

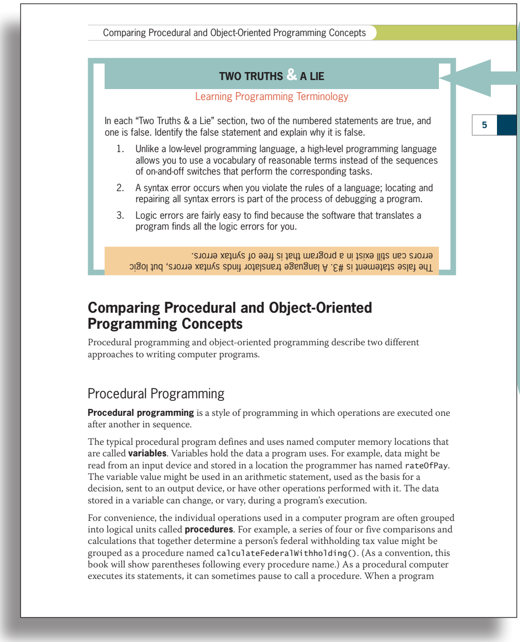
s two types of division:
ands are floating-point

The result is an inte-
result of 45 / 2 is 22. As
times; 38 / 5, 37 / 5, 36 / 5,

VIDEO LESSONS help explain important chapter concepts. Videos are part of the eBook in MindTap and are also posted on the Instructor Companion Site.

The author does an awesome job: the examples, problems, and material are very easy to understand!

—Bernice Cunningham, Wayne County Community College District



TWO TRUTHS & A LIE

Learning Programming Terminology

In each "Two Truths & a Lie" section, two of the numbered statements are true, and one is false. Identify the false statement and explain why it is false.

1. Unlike a low-level programming language, a high-level programming language allows you to use a vocabulary of reasonable terms instead of the sequences of on-and-off switches that perform the corresponding tasks.
2. A syntax error occurs when you violate the rules of a language; locating and repairing all syntax errors is part of the process of debugging a program.
3. Logic errors are fairly easy to find because the software that translates a program finds all the logic errors for you.

The false statement is #3. A language translator finds syntax errors, but logic errors can still exist in a program that is free of syntax errors.

Comparing Procedural and Object-Oriented Programming Concepts

Procedural programming and object-oriented programming describe two different approaches to writing computer programs.

Procedural Programming

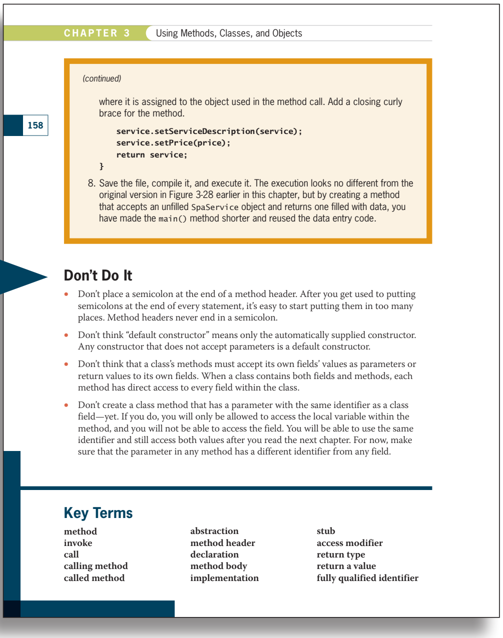
Procedural programming is a style of programming in which operations are executed one after another in sequence.

The typical procedural program defines and uses named computer memory locations that are called **variables**. Variables hold the data a program uses. For example, data might be read from an input device and stored in a location the programmer has named `rateOfPay`. The variable value might be used in an arithmetic statement, used as the basis for a decision, sent to an output device, or have other operations performed with it. The data stored in a variable can change, or vary, during a program's execution.

For convenience, the individual operations used in a computer program are often grouped into logical units called **procedures**. For example, a series of four or five comparisons and calculations that together determine a person's federal withholding tax value might be grouped as a procedure named `calculateFederalWithholding()`. (As a convention, this book will show parentheses following every procedure name.) As a procedural computer executes its statements, it can sometimes pause to call a procedure. When a program

TWO TRUTHS & A LIE quizzes appear after each chapter section, with answers provided. The quiz contains three statements based on the preceding section of text—two statements are true and one is false. Answers give immediate feedback without "giving away" answers to the multiple-choice questions and programming problems later in the chapter. Students also have the option to take these quizzes in MindTap.

DON'T DO IT sections at the end of each chapter list advice for avoiding common programming errors.



(continued)

where it is assigned to the object used in the method call. Add a closing curly brace for the method.

```
    service.setServiceDescription(service);
    service.setPrice(price);
    return service;
}
```

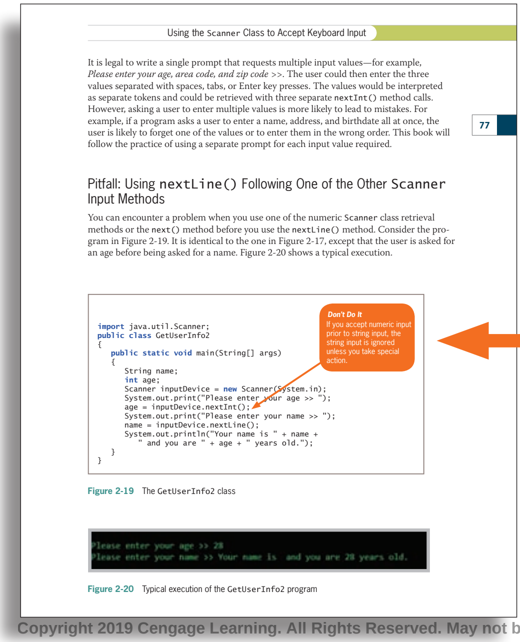
8. Save the file, compile it, and execute it. The execution looks no different from the original version in Figure 3-28 earlier in this chapter, but by creating a method that accepts an unfilled `SpaService` object and returns one filled with data, you have made the `main()` method shorter and reused the data entry code.

Don't Do It

- Don't place a semicolon at the end of a method header. After you get used to putting semicolons at the end of every statement, it's easy to start putting them in too many places. Method headers never end in a semicolon.
- Don't think "default constructor" means only the automatically supplied constructor. Any constructor that does not accept parameters is a default constructor.
- Don't think that a class's methods must accept its own fields' values as parameters or return values to its own fields. When a class contains both fields and methods, each method has direct access to every field within the class.
- Don't create a class method that has a parameter with the same identifier as a class field—yet. If you do, you will only be allowed to access the local variable within the method, and you will not be able to access the field. You will be able to use the same identifier and still access both values after you read the next chapter. For now, make sure that the parameter in any method has a different identifier from any field.

Key Terms

method
invoke
call
calling method
called method
abstraction
method header
declaration
method body
implementation
stub
access modifier
return type
return a value
fully qualified identifier



It is legal to write a single prompt that requests multiple input values—for example, *Please enter your age, area code, and zip code >>*. The user could then enter the three values separated with spaces, tabs, or Enter key presses. The values would be interpreted as separate tokens and could be retrieved with three separate `nextInt()` method calls. However, asking a user to enter multiple values is more likely to lead to mistakes. For example, if a program asks a user to enter a name, address, and birthdate all at once, the user is likely to forget one of the values or to enter them in the wrong order. This book will follow the practice of using a separate prompt for each input value required.

Pitfall: Using `nextLine()` Following One of the Other `Scanner` Input Methods

You can encounter a problem when you use one of the numeric `Scanner` class retrieval methods or the `next()` method before you use the `nextLine()` method. Consider the program in Figure 2-19. It is identical to the one in Figure 2-17, except that the user is asked for an age before being asked for a name. Figure 2-20 shows a typical execution.

```
import java.util.Scanner;
public class GetUserInfo2
{
   public static void main(String[] args)
   {
      String name;
      int age;
      Scanner inputDevice = new Scanner(System.in);
      System.out.print("Please enter your age >> ");
      age = inputDevice.nextInt();
      System.out.print("Please enter your name >> ");
      name = inputDevice.nextLine();
      System.out.println("Your name is " + name +
         " and you are " + age + " years old.");
   }
}
```

Don't Do It
If you accept numeric input prior to string input, the string input is ignored unless you take special action.

Figure 2-19 The `GetUserInfo2` class

```
Please enter your age >> 28
Please enter your name >> Your name is  and you are 28 years old.
```

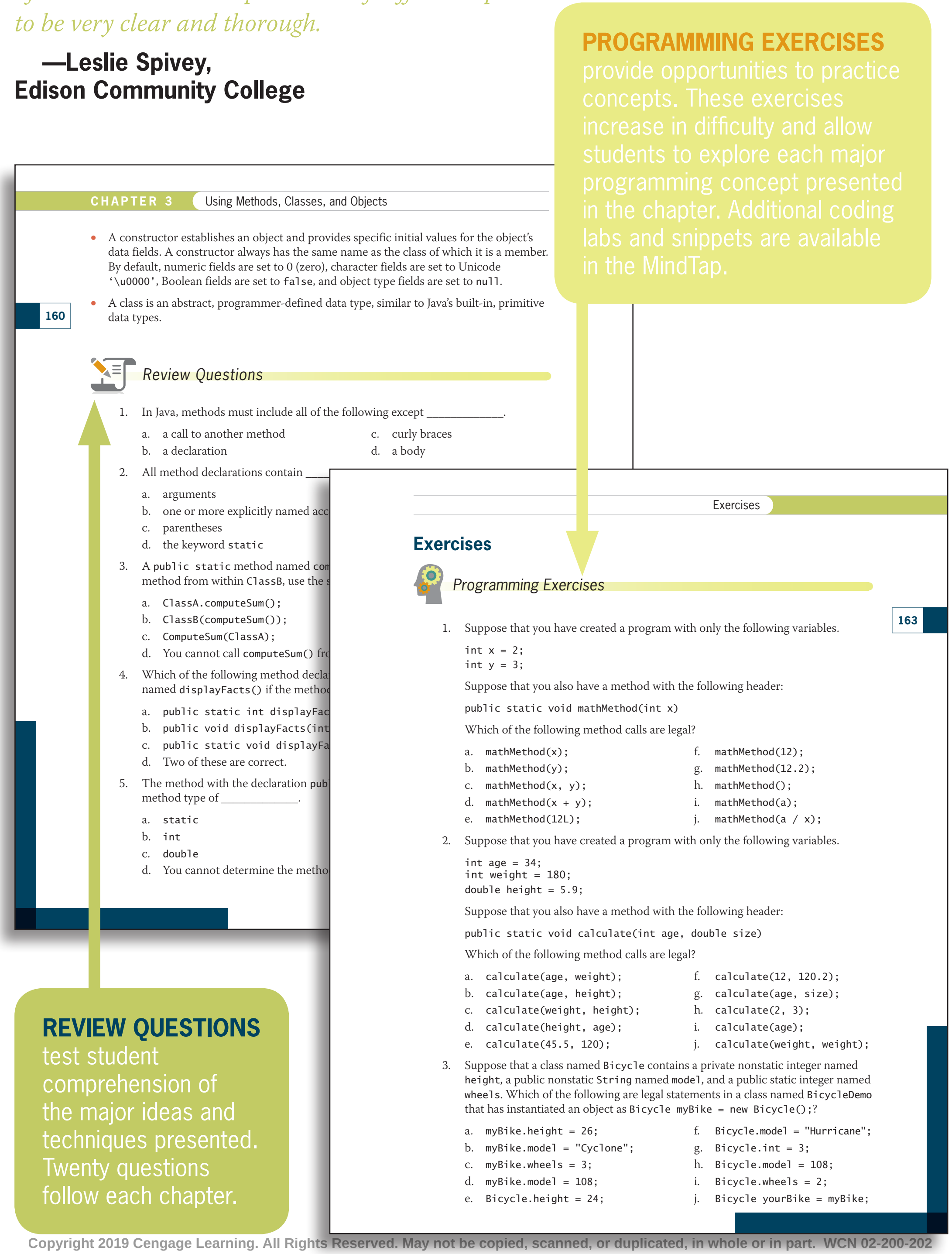
Figure 2-20 Typical execution of the `GetUserInfo2` program

THE DON'T DO IT ICON illustrates how NOT to do something—for example, having a dead code path in a program. This icon provides a visual jolt to the student, emphasizing that particular figures are NOT to be emulated and making students more careful to recognize problems in existing code.

Assessment

I found the author's explanation of difficult topics to be very clear and thorough.

**—Leslie Spivey,
Edison Community College**



PROGRAMMING EXERCISES provide opportunities to practice concepts. These exercises increase in difficulty and allow students to explore each major programming concept presented in the chapter. Additional coding labs and snippets are available in the MindTap.

CHAPTER 3 Using Methods, Classes, and Objects

- A constructor establishes an object and provides specific initial values for the object's data fields. A constructor always has the same name as the class of which it is a member. By default, numeric fields are set to 0 (zero), character fields are set to Unicode '\u0000', Boolean fields are set to `false`, and object type fields are set to `null`.
- A class is an abstract, programmer-defined data type, similar to Java's built-in, primitive data types.

160

Review Questions

1. In Java, methods must include all of the following except ____________.
   a. a call to another method c. curly braces
   b. a declaration d. a body
2. All method declarations contain ____
   a. arguments
   b. one or more explicitly named acc
   c. parentheses
   d. the keyword `static`
3. A `public static` method named `com` method from within `ClassB`, use the s
   a. `ClassA.computeSum();`
   b. `ClassB(computeSum());`
   c. `ComputeSum(ClassA);`
   d. You cannot call `computeSum()` fr
4. Which of the following method decla named `displayFacts()` if the metho
   a. `public static int displayFac`
   b. `public void displayFacts(int`
   c. `public static void displayFa`
   d. Two of these are correct.
5. The method with the declaration `pub` method type of ____________.
   a. `static`
   b. `int`
   c. `double`
   d. You cannot determine the metho

Exercises

Exercises

Programming Exercises

163

1. Suppose that you have created a program with only the following variables.

```
int x = 2;
int y = 3;
```

Suppose that you also have a method with the following header:

```
public static void mathMethod(int x)
```

Which of the following method calls are legal?

a. `mathMethod(x);` f. `mathMethod(12);`
b. `mathMethod(y);` g. `mathMethod(12.2);`
c. `mathMethod(x, y);` h. `mathMethod();`
d. `mathMethod(x + y);` i. `mathMethod(a);`
e. `mathMethod(12L);` j. `mathMethod(a / x);`

2. Suppose that you have created a program with only the following variables.

```
int age = 34;
int weight = 180;
double height = 5.9;
```

Suppose that you also have a method with the following header:

```
public static void calculate(int age, double size)
```

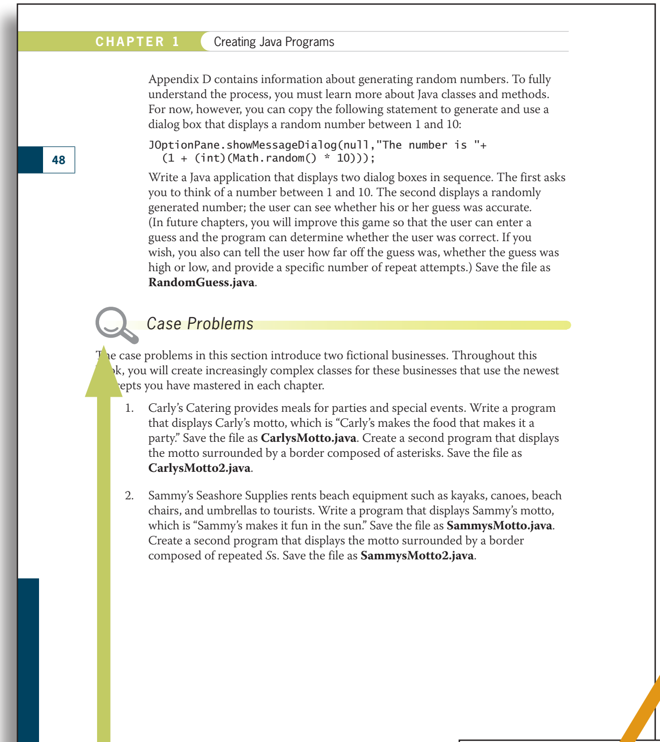
Which of the following method calls are legal?

a. `calculate(age, weight);` f. `calculate(12, 120.2);`
b. `calculate(age, height);` g. `calculate(age, size);`
c. `calculate(weight, height);` h. `calculate(2, 3);`
d. `calculate(height, age);` i. `calculate(age);`
e. `calculate(45.5, 120);` j. `calculate(weight, weight);`

3. Suppose that a class named `Bicycle` contains a private nonstatic integer named `height`, a public nonstatic `String` named `model`, and a public static integer named `wheels`. Which of the following are legal statements in a class named `BicycleDemo` that has instantiated an object as `Bicycle myBike = new Bicycle();`?

a. `myBike.height = 26;` f. `Bicycle.model = "Hurricane";`
b. `myBike.model = "Cyclone";` g. `Bicycle.int = 3;`
c. `myBike.wheels = 3;` h. `Bicycle.model = 108;`
d. `myBike.model = 108;` i. `Bicycle.wheels = 2;`
e. `Bicycle.height = 24;` j. `Bicycle yourBike = myBike;`

REVIEW QUESTIONS test student comprehension of the major ideas and techniques presented. Twenty questions follow each chapter.



Appendix D contains information about generating random numbers. To fully understand the process, you must learn more about Java classes and methods. For now, however, you can copy the following statement to generate and use a dialog box that displays a random number between 1 and 10:

```
JOptionPane.showMessageDialog(null,"The number is "+
   (1 + (int)(Math.random() * 10)));
```

Write a Java application that displays two dialog boxes in sequence. The first asks you to think of a number between 1 and 10. The second displays a randomly generated number; the user can see whether his or her guess was accurate. (In future chapters, you will improve this game so that the user can enter a guess and the program can determine whether the user was correct. If you wish, you also can tell the user how far off the guess was, whether the guess was high or low, and provide a specific number of repeat attempts.) Save the file as **RandomGuess.java**.

Case Problems

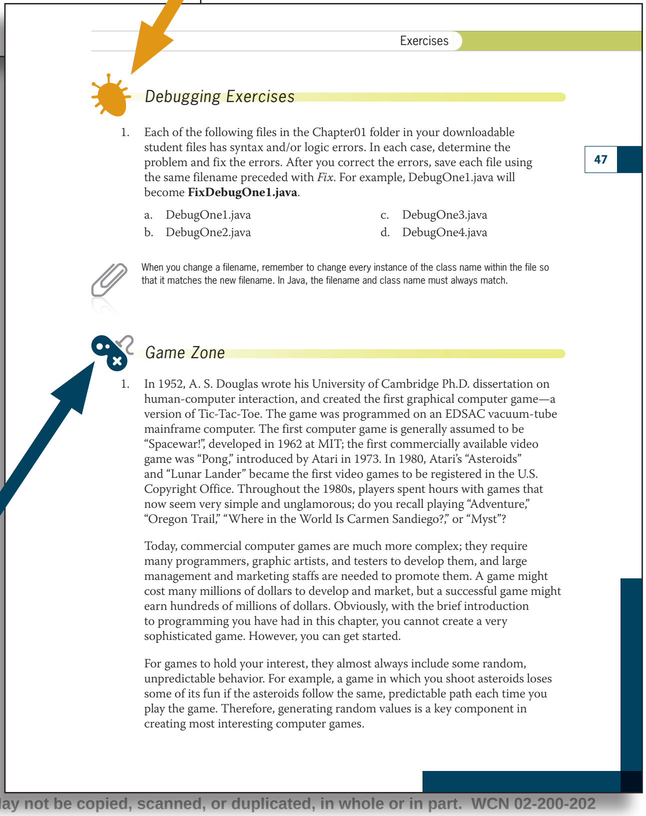
The case problems in this section introduce two fictional businesses. Throughout this book, you will create increasingly complex classes for these businesses that use the newest concepts you have mastered in each chapter.

1. Carly's Catering provides meals for parties and special events. Write a program that displays Carly's motto, which is "Carly's makes the food that makes it a party." Save the file as **CarlysMotto.java**. Create a second program that displays the motto surrounded by a border composed of asterisks. Save the file as **CarlysMotto2.java**.

2. Sammy's Seashore Supplies rents beach equipment such as kayaks, canoes, beach chairs, and umbrellas to tourists. Write a program that displays Sammy's motto, which is "Sammy's makes it fun in the sun." Save the file as **SammysMotto.java**. Create a second program that displays the motto surrounded by a border composed of repeated *S*s. Save the file as **SammysMotto2.java**.

DEBUGGING EXERCISES are included with each chapter because examining programs critically and closely is a crucial programming skill. Students can download these exercises at *www.Cengagebrain.com*. These files are also available to instructors through *sso.cengage.com*.

CASE PROBLEMS provide opportunities to build more detailed programs that continue to incorporate increasing functionality throughout the book.



Debugging Exercises

1. Each of the following files in the Chapter01 folder in your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the errors. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, DebugOne1.java will become **FixDebugOne1.java**.

   a. DebugOne1.java
   b. DebugOne2.java
   c. DebugOne3.java
   d. DebugOne4.java

When you change a filename, remember to change every instance of the class name within the file so that it matches the new filename. In Java, the filename and class name must always match.

Game Zone

1. In 1952, A. S. Douglas wrote his University of Cambridge Ph.D. dissertation on human-computer interaction, and created the first graphical computer game—a version of Tic-Tac-Toe. The game was programmed on an EDSAC vacuum-tube mainframe computer. The first computer game is generally assumed to be "Spacewar!", developed in 1962 at MIT; the first commercially available video game was "Pong," introduced by Atari in 1973. In 1980, Atari's "Asteroids" and "Lunar Lander" became the first video games to be registered in the U.S. Copyright Office. Throughout the 1980s, players spent hours with games that now seem very simple and unglamorous; do you recall playing "Adventure," "Oregon Trail," "Where in the World Is Carmen Sandiego?," or "Myst"?

   Today, commercial computer games are much more complex; they require many programmers, graphic artists, and testers to develop them, and large management and marketing staffs are needed to promote them. A game might cost many millions of dollars to develop and market, but a successful game might earn hundreds of millions of dollars. Obviously, with the brief introduction to programming you have had in this chapter, you cannot create a very sophisticated game. However, you can get started.

   For games to hold your interest, they almost always include some random, unpredictable behavior. For example, a game in which you shoot asteroids loses some of its fun if the asteroids follow the same, predictable path each time you play the game. Therefore, generating random values is a key component in creating most interesting computer games.

GAME ZONE EXERCISES are included at the end of each chapter. Students can create games as an additional entertaining way to understand key programming concepts.

CHAPTER 1

Creating Java Programs

Upon completion of this chapter, you will be able to:

- Define basic programming terminology
- Compare procedural and object-oriented programming
- Describe the features of the Java programming language
- Analyze a Java application that produces console output
- Compile a Java class and correct syntax errors
- Run a Java application and correct logic errors
- Add comments to a Java class
- Create a Java application that produces GUI output
- Find help

Learning Programming Terminology

A **computer program** is a set of instructions that you write to tell a computer what to do. Computer equipment, such as a monitor or keyboard, is **hardware**, and programs are **software**. A program that performs a task for a user (such as calculating and producing paychecks, word processing, or playing a game) is **application software**; a program that manages the computer itself (such as Windows or Linux) is **system software**. The **logic** behind any computer program, whether it is an application or system program, determines the exact order of instructions needed to produce desired results. Much of this book describes how to develop the logic to create programs that are application software, called *applications* (or, especially if used on a mobile device, *apps*) for short.

You can write computer programs in a **high-level programming language** such as Java, Visual Basic, C++, or C#. A high-level programming language allows you to use English-like, easy-to-remember terms such as *read*, *write*, and *add*. These languages are called high-level languages to distinguish them from **low-level languages** that correspond closely to a computer's circuitry and are not as easily read or understood. Because they correspond to circuitry, low-level languages must be customized for every type of machine on which a program runs.

All computer programs ultimately are converted to the lowest level language, which is machine language. **Machine language**, or **machine code**, is the most basic set of instructions that a computer can execute. Each type of processor (the internal hardware that handles computer instructions) has its own set of machine language instructions. Programmers often describe machine language using 1s and 0s to represent the on-and-off circuitry of computer systems.

The system that uses only 1s and 0s is the *binary numbering system*. Appendix B describes the binary system in detail. Later in this chapter, you will learn that *bytecode* is the name for the binary code created when Java programs are converted to machine language.

Every programming language has its own **syntax**, or rules about how language elements are combined correctly to produce usable statements. For example, depending on the specific high-level language, you might use the verb *print* or *write* to produce output. All languages have a specific, limited vocabulary (the language's **keywords**) and a specific set of rules for using that vocabulary. When you are learning a computer programming language, such as Java, C++, or Visual Basic, you are learning the vocabulary and syntax for that language.

Using a programming language, programmers write a series of **program statements**, which are similar to English sentences. The statements carry out the program's tasks. Program statements are also known as **commands** because they are orders to the computer, such as *Output this word* or *Add these two numbers*.

After the program statements are written in a high-level programming language, a computer program called a **compiler** or **interpreter** translates the statements into machine language. A compiler translates an entire program before carrying out any statements, or **executing** them, whereas an interpreter translates one program statement at a time, executing a statement as soon as it is translated.

Whether you use a compiler or interpreter often depends on the programming language you use. For example, C++ is a compiled language, and Visual Basic is an interpreted language. Each type of translator has its supporters; programs written in compiled languages execute more quickly, whereas programs written in interpreted languages can be easier to develop and debug. Java uses the best of both technologies: a compiler to translate your programming statements and an interpreter to read the compiled code line by line when the program executes (also called **at run time**).

Compilers and interpreters issue one or more error messages each time they encounter an invalid program statement—that is, a statement containing a **syntax error**, or misuse of the language. Examples of syntax errors include misspelling a keyword or omitting a word that a statement requires. When a syntax error is detected, the programmer can correct the error and attempt another translation. Repairing all syntax errors is the first part of the process of **debugging** a program—freeing the program of all flaws or errors, also known as **bugs**. Figure 1-1 illustrates the steps a programmer takes while developing an executable program. You will learn more about debugging Java programs later in this chapter.

As Figure 1-1 shows, you might write a program that compiles successfully (that is, it contains no syntax errors), but it still might not be a correct program because it might contain one or more logic errors. A **logic error** is a bug that allows a program to run, but that causes it to operate incorrectly. Correct logic requires that all the right commands be issued in the appropriate order. Examples of logic errors include multiplying two values when you meant to divide them or producing output prior to obtaining the appropriate input. When you develop a program of any significant size, you should plan its logic before you write any program statements.

Correcting logic errors is much more difficult than correcting syntax errors. Syntax errors are discovered by the language translator when you compile a program, but a program can be free of syntax errors and execute while still retaining logic errors. Sometimes you can find logic errors by carefully examining the structure of your program (when a group of programmers do this together, it is called a *structured walkthrough*), but sometimes you can identify logic errors only when you examine a program's output. For example, if you know an employee's paycheck should contain the value $4,000, but when you examine a payroll program's output you see that it holds $40, then a logic error has occurred. Perhaps an incorrect calculation was performed, or maybe the hours worked value was output by mistake instead of the net pay value. When output is incorrect, the programmer must carefully examine all the statements within the program, revise or move the offending statements, and translate and test the program again.

Just because a program produces correct output does not mean it is free from logic errors. For example, suppose that a program should multiply two values entered by the user, that the user enters two 2s, and the output is 4. The program might actually be adding the values by mistake. The programmer would discover the logic error only by entering different values, such as 5 and 7, and examining the result.

Programmers call some logic errors **semantic errors**. For example, if you misspell a programming language word, you commit a syntax error, but if you use a correct word in the wrong context, you commit a semantic error.

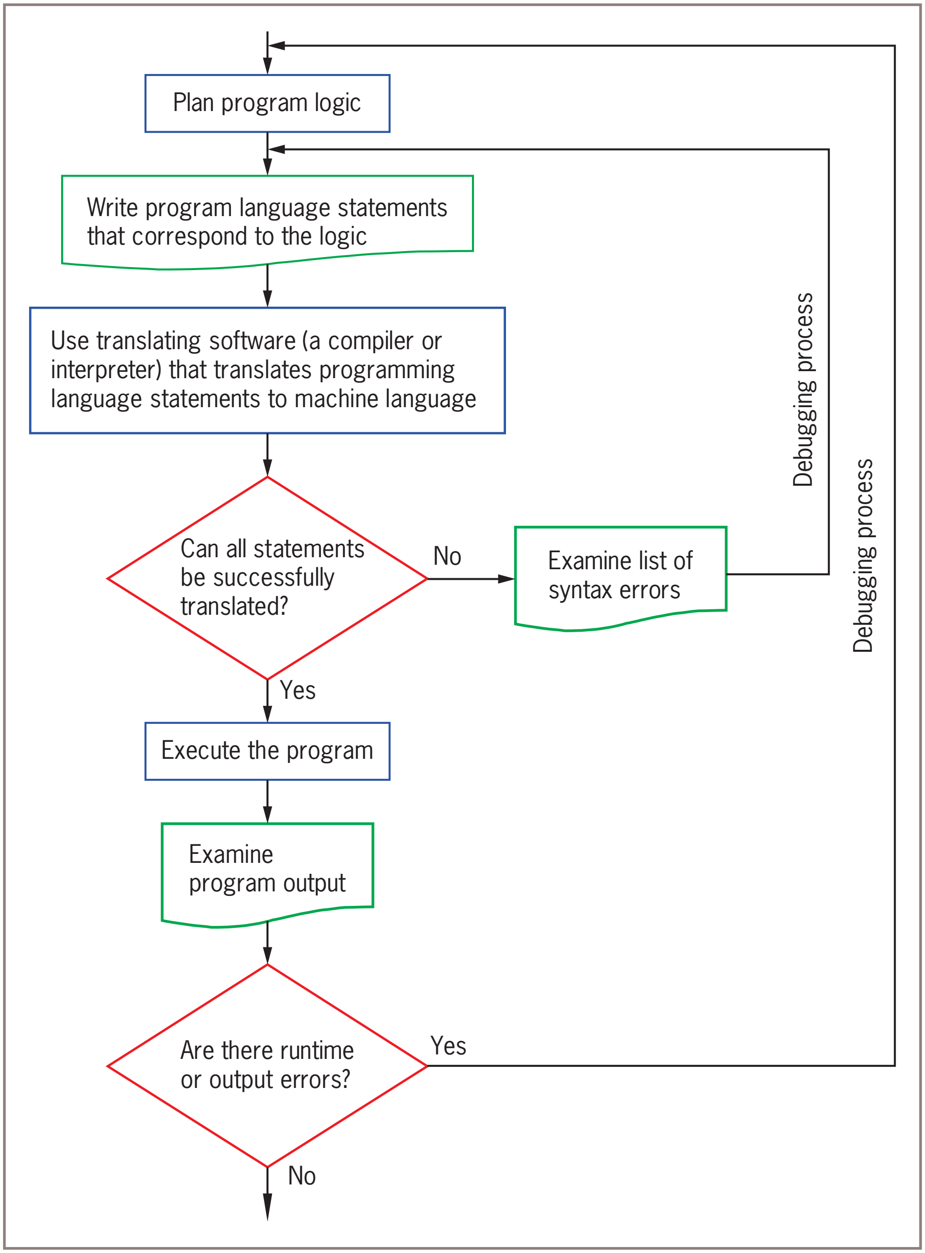

Plan program logic
Write program language statements that correspond to the logic
Use translating software (a compiler or interpreter) that translates programming language statements to machine language
Can all statements be successfully translated?
No
Examine list of syntax errors
Debugging process
Debugging process
Yes
Execute the program
Examine program output
Are there runtime or output errors?
Yes
No

Figure 1-1 The program development process

TWO TRUTHS & A LIE

Learning Programming Terminology

In each “Two Truths & a Lie” section, two of the numbered statements are true, and one is false. Identify the false statement and explain why it is false.

1. Unlike a low-level programming language, a high-level programming language allows you to use a vocabulary of reasonable terms instead of the sequences of on-and-off switches that perform the corresponding tasks.
2. A syntax error occurs when you violate the rules of a language; locating and repairing all syntax errors is part of the process of debugging a program.
3. Logic errors are fairly easy to find because the software that translates a program finds all the logic errors for you.

The false statement is #3. A language translator finds syntax errors, but logic errors can still exist in a program that is free of syntax errors.

Comparing Procedural and Object-Oriented Programming Concepts

Procedural programming and object-oriented programming describe two different approaches to writing computer programs.

Procedural Programming

Procedural programming is a style of programming in which operations are executed one after another in sequence.

The typical procedural program defines and uses named computer memory locations that are called **variables**. Variables hold the data a program uses. For example, data might be read from an input device and stored in a location the programmer has named `rateOfPay`. The variable value might be used in an arithmetic statement, used as the basis for a decision, sent to an output device, or have other operations performed with it. The data stored in a variable can change, or vary, during a program’s execution.

For convenience, the individual operations used in a computer program are often grouped into logical units called **procedures**. For example, a series of four or five comparisons and calculations that together determine a person’s federal withholding tax value might be grouped as a procedure named `calculateFederalWithholding()`. (As a convention, this book will show parentheses following every procedure name.) As a procedural computer executes its statements, it can sometimes pause to call a procedure. When a program

calls a procedure, the current logic is temporarily suspended so that the procedure's commands can execute. A single procedural program might contain any number of procedure calls. Procedures are also called *modules, methods, functions*, and *subroutines*. Users of different programming languages tend to use different terms. As you will learn later in this chapter, Java programmers most frequently use the term *method*.

Object-Oriented Programming

Object-oriented programming is an extension of procedural programming in which you take a slightly different approach to writing computer programs. Writing **object-oriented programs** involves:

- Creating classes, which are blueprints for objects
- Creating objects, which are specific instances of those classes
- Creating applications that manipulate or use those objects

Programmers use *OO* as an abbreviation for *object-oriented*; it is pronounced "oh oh." Object-oriented programming is abbreviated *OOP*, and pronounced to rhyme with *soup*.

Originally, object-oriented programming was used most frequently for two major types of applications:

- **Computer simulations**, which attempt to mimic real-world activities so that their processes can be improved or so that users can better understand how the real-world processes operate
- **Graphical user interfaces**, or **GUIs** (pronounced *gooeys*), which allow users to interact with a program in a graphical environment

Thinking about objects in these two types of applications makes sense. For example, a city might want to develop a program that simulates traffic patterns and controls traffic signals to help prevent tie-ups. Programmers would create classes for objects such as cars and pedestrians that contain their own data and rules for behavior. For example, each car has a speed and a method for changing that speed. The specific instances of cars could be set in motion to create a simulation of a real city at rush hour.

Creating a GUI environment for users is also a natural use for object orientation. It is easy to think of the components a user manipulates on a computer screen, such as buttons and scroll bars, as similar to real-world objects. Each GUI object contains data—for example, a button on a screen has a specific size and color. Each object also contains behaviors—for example, each button can be clicked and reacts in a specific way when clicked. Some people consider the term *object-oriented programming* to be synonymous with GUI programming, but object-oriented programming means more. Although many GUI programs are object oriented, not all object-oriented programs use GUI objects. Modern businesses use object-oriented design techniques when developing all sorts of business applications, whether they are GUI applications or not. In the first 13 chapters of this book, you will learn object-oriented techniques that are appropriate for any program type; in the last chapters, you will apply what you have learned about those techniques specifically to GUI applications.

Understanding object-oriented programming requires grasping three basic concepts:

- Encapsulation as it applies to classes as objects
- Inheritance
- Polymorphism

Understanding Classes, Objects, and Encapsulation

In object-oriented terminology, a **class** is a group or collection of objects with common properties. In the same way that a blueprint exists before any houses are built from it, and a recipe exists before any cookies are baked from it, a class definition exists before any objects are created from it. A **class definition** describes what attributes its objects will have and what those objects will be able to do. **Attributes** are the characteristics that define an object; they are **properties** of the object. When you learn a programming language such as Java, you learn to work with two types of classes: those that have already been developed by the language's creators and your own new, customized classes.

An **object** is a specific, concrete **instance** of a class. Creating an instance is called **instantiation**. You can create objects from classes that you write and from classes written by other programmers, including Java's creators. The values contained in an object's properties often differentiate instances of the same class from one another. For example, the class `Automobile` describes what `Automobile` objects are like. Some properties of the `Automobile` class are make, model, year, and color. Each `Automobile` object possesses the same attributes, but not necessarily the same values for those attributes. One `Automobile` might be a 2014 white Ford Taurus and another might be a 2018 red Chevrolet Camaro. Similarly, your dog has the properties of all `Dogs`, including a breed, name, age, and whether the dog's shots are current. The values of the properties of an object are referred to as the object's **state**. In other words, you can think of objects as roughly equivalent to nouns (words that describe a person, place, or thing), and of their attributes as similar to adjectives that describe the nouns.

When you understand an object's class, you understand the characteristics of the object. If your friend purchases an `Automobile`, you know it has a model name, and if your friend gets a `Dog`, you know the dog has a breed. Knowing what attributes exist for classes allows you to ask appropriate questions about the states or values of those attributes. For example, you might ask how many miles the car gets per gallon, but you would not ask whether the car has had shots. Similarly, in a GUI operating environment, you expect each component to have specific, consistent attributes and methods, such as a window having a title bar and a close button, because each component gains these properties as a member of the general class of GUI components. Figure 1-2 shows the relationship of some `Dog` objects to the `Dog` class.

By convention, programmers using Java begin their class names with an uppercase letter. Thus, the class that defines the attributes and methods of an automobile probably would be named `Automobile`, and the class for dogs probably would be named `Dog`. This convention, however, is not required to produce a workable program.

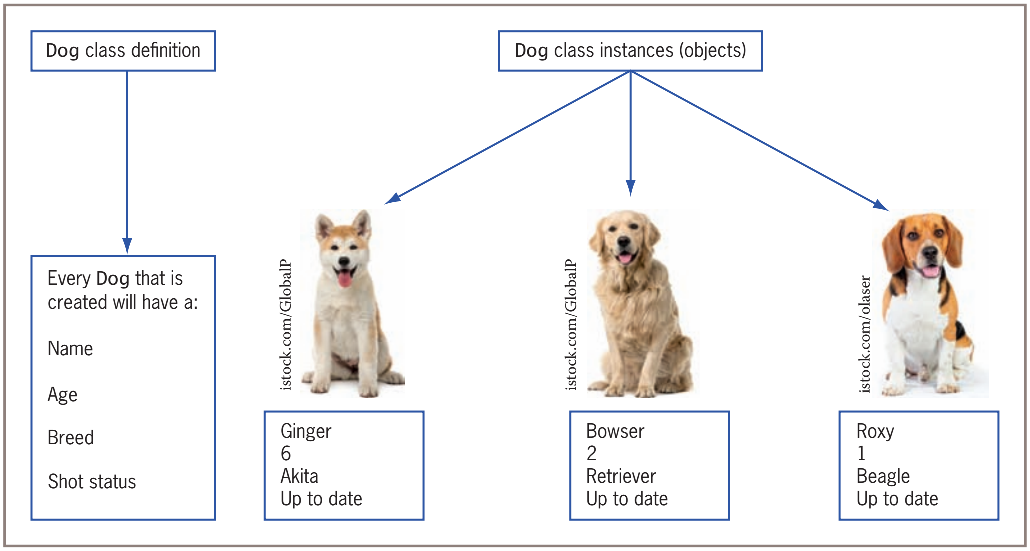


Figure 1-2 `Dog` class definition and some objects created from it

Besides defining properties, classes define methods their objects can use. A **method** is a self-contained block of program code that carries out some action, similar to a procedure in a procedural program. An `Automobile`, for example, might have methods for moving forward, moving backward, and determining the status of its gas tank. Similarly, a `Dog` might have methods for walking, eating, and determining its name, and a program's GUI components might have methods for maximizing and minimizing them as well as determining their size. In other words, if objects are similar to nouns, then methods are similar to verbs.

In object-oriented classes, attributes and methods are encapsulated into objects. **Encapsulation** refers to two closely related object-oriented notions:

- Encapsulation is the enclosure of data and methods within an object. Encapsulation allows you to treat all of an object's methods and data as a single entity. Just as an actual dog contains all of its attributes and abilities, so would a program's `Dog` object.
- Encapsulation also refers to the concealment of an object's data and methods from outside sources. Concealing data is sometimes called *information hiding*, and concealing how methods work is *implementation hiding*; you will learn more about both terms in the chapter "Using Methods, Classes, and Objects." Encapsulation lets you hide specific object attributes and methods from outside sources and provides the security that keeps data and methods safe from inadvertent changes.

If an object's methods are well written, the user can be unaware of the low-level details of how the methods are executed, and the user must simply understand the interface or interaction between the method and the object. For example, if you can fill your `Automobile` with gasoline, it is because you understand the interface between the gas pump nozzle and the vehicle's gas tank opening. You don't need to understand how the pump works mechanically or where the gas tank is located inside your vehicle. If you can read your speedometer, it does not matter how the displayed figure is calculated. As a matter of fact, if someone produces a superior, more accurate speed-determining device and inserts it in your `Automobile`, you don't have to know or care how it operates, as long as your interface remains the same. The same principles apply to well-constructed classes used in object-oriented programs—programs that use classes only need to work with interfaces.

Understanding Inheritance and Polymorphism

An important feature of object-oriented program design that differentiates it from procedural program design is **inheritance**—the ability to create classes that share the attributes and methods of existing classes, but with more specific features. For example, `Automobile` is a class, and all `Automobile` objects share many traits and abilities. `Convertible` is a class that inherits from the `Automobile` class; a `Convertible` is a type of `Automobile` that has and can do everything a "plain" `Automobile` does—but with an added ability to lower its top. (In turn, `Automobile` inherits from the `Vehicle` class.) `Convertible` is not an object—it is a class. A specific `Convertible` is an object—for example, `my1967BlueMustangConvertible`.

Inheritance helps you understand real-world objects. For example, the first time you encounter a convertible, you already understand how the ignition, brakes, door locks, and other systems work because you realize that a convertible is a type of automobile. Therefore, you need to be concerned only with the attributes and methods that are "new" with a convertible. The advantages in programming are the same—you can build new classes based on existing classes and concentrate on the specialized features you are adding.

A final important concept in object-oriented terminology (that does not exist in procedural programming terminology) is **polymorphism**. Literally, polymorphism means *many forms*—it describes the feature of languages that allows the same word or symbol to be interpreted correctly in different situations based on the context. For example, although the classes `Automobile`, `Sailboat`, and `Airplane` all inherit from `Vehicle`, methods such as `turn` and `stop` work differently for instances of those classes. The advantages of polymorphism will become more apparent when you begin to create GUI applications containing features such as windows, buttons, and menu bars. In a GUI application, it is convenient to remember one method name, such as `setColor` or `setHeight`, and have it work correctly no matter what type of object you are modifying.

When you see a plus sign (+) between two numbers, you understand they are being added. When you see it carved in a tree between two names, you understand that the names are linked romantically. Because the symbol has diverse meanings based on context, it is polymorphic. Chapters 10 and 11 provide more information about inheritance and polymorphism and how they are implemented in Java. Using Java, you can write either procedural or object-oriented programs. In this book, you will learn about how to do both.