JAVA™ PROGRAMMING

JOYCE FARRELL

SEVENTH EDITION

CENGAGE brain .com

Buy. Rent. Access.

Access student data files and other study tools on **cengagebrain.com**.

For detailed instructions visit **www.cengage.com/ct/studentdownload**.

Store your Data Files on a USB drive for maximum efficiency in organizing and working with the files.

Macintosh users should use a program to expand WinZip or PKZip archives. Ask your instructor or lab coordinator for assistance.

JAVA™ PROGRAMMING

SEVENTH EDITION

JAVA™ PROGRAMMING

JOYCE FARRELL

COURSE TECHNOLOGY
CENGAGE Learning®

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

COURSE TECHNOLOGY
CENGAGE Learning®

Java Programming, Seventh Edition
Joyce Farrell

Executive Editor: Kathleen McMahon

Senior Product Manager: Alyssa Pratt

Development Editor: Dan Seiter

Editorial Assistant: Sarah Ryan

Brand Manager: Kay Stefanski

Print Buyer: Julio Esperas

Art and Design Direction, Production Management, and Composition: Integra Software Services Pvt. Ltd.

Cover Designer: Lisa Kuhn/Curio Press, LLC www.curiopress.com

Cover Photo: © Leigh Prather/Veer

Copyeditor: Mark Goodin

Proofreader: Pamela Hunt

Indexer: Alexandra Nickerson

Library of Congress Control Number: 2012953690

Student Edition:

ISBN-13: 978-1-285-08195-3
ISBN-10: 1-285-08195-1

Course Technology
20 Channel Center Street
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil and Japan. Locate your local office at **international.cengage.com/region**

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

For your course and learning solutions, visit **www.cengage.com**.

Purchase any of our products at your local college store or at our preferred online store: **www.CengageBrain.com**.

Instructors: Please visit **login.cengage.com** and log in to access instructor-specific resources.

Printed in the United States of America
1 2 3 4 5 6 18 17 16 15 14 13

Brief Contents

Contents

Preface

Java Programming, Seventh 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. This book provides a solid background in good object-oriented programming techniques and introduces terminology using clear, familiar language. The writing is nontechnical and emphasizes good programming practices. 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. The explanations in this textbook are written clearly in straightforward sentences, making it easier for native and non-native English speakers alike to master the programming concepts. 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 can also be downloaded from the publisher's Web site, so students can easily run the programs and experiment with changes to them.

The student using *Java Programming, Seventh 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 of 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, Seventh 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 of 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 on handling files so you can permanently store and retrieve program output.

Chapters 14 and 15 introduce GUI Swing components—Java's visually pleasing, user-friendly widgets—and their layout managers. Chapters 16 and 17 show you ways to provide interactive excitement using graphics, applets, images, and sound.

Features

The following features are new for the Seventh Edition:

- **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. Previous editions of the book contained a long, multipart "You Do It" section at the end of each chapter, but in this edition, more and shorter sections follow important chapter topics so the student can focus on one new concept at a time.
- **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.
- **PROGRAMMING EXERCISES:** Each chapter concludes with meaningful programming exercises that provide additional practice of the skills and concepts learned in the chapter. These exercises vary in difficulty and are designed to allow exploration of logical programming concepts. Each chapter contains several new programming exercises not seen in previous editions.
- **INCREASED EMPHASIS ON STUDENT RESEARCH:** In this edition, the student frequently is directed to the Java Web site 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.

Additionally, *Java Programming, Seventh 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.

- **NOTES:** These highlighted tips provide additional information—for example, an alternative method of performing a procedure, another term for a concept, background information on a technique, or a common error to avoid.
- **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 Web site. 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 AND 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 create interactive games using the programming techniques learned up to that point; 70 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.
- **GLOSSARY:** A glossary contains definitions for all key terms in the book, presented in alphabetical order.

- **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) 7, the most recent version available.

CourseMate

The more you study, the better the results. Make the most of your study time by accessing everything you need to succeed in one place. Read your textbook, take notes, review flashcards, watch videos, and take practice quizzes online. CourseMate goes beyond the book to deliver what you need! Learn more at *www.cengage.com/coursemate*.

The *Java Programming* CourseMate includes:

- **Debugging Exercises:** Four error-filled programs accompany each chapter. By debugging these programs, students can gain expertise in program logic in general and the Java programming language in particular.
- **Video Lessons:** Each chapter is accompanied by at least three video lessons that help to explain important chapter concepts. These videos were created and narrated by the author.
- **Interactive Study Aids:** An interactive eBook, quizzes, flashcards, and more!

Instructors may add CourseMate to the textbook package, or students may purchase CourseMate directly at *www.CengageBrain.com*.

Instructor Resources

The following teaching tools are available for download at our Instructor Companion Site. Simply search for this text at *login.cengage.com*. An instructor login is required.

- **Electronic 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 is also available. Additional exercises in the Instructor's Manual include:
  - **Tough Questions:** Two or more fairly difficult questions that an applicant might encounter in a technical job interview accompany each chapter. These questions are often open-ended; some involve coding and others might involve research.
  - **Up for Discussion:** A few thought-provoking questions concerning programming in general or Java in particular supplement each chapter. The questions can be used to start classroom or online discussions, or to develop and encourage research, writing, and language skills.

  - **Programming exercises and solutions:** Each chapter is accompanied by several programming exercises to supplement those offered in the text. Instructors can use these exercises as additional or alternate assignments, or as the basis for lectures.

- **ExamView:** This textbook is accompanied by ExamView, a powerful testing software package that allows instructors to create and administer printed, computer (LAN-based), and Internet-based exams. ExamView includes hundreds of questions that correspond to the topics covered in this text, enabling students to generate detailed study guides that include page references for further review. The computer-based and Internet testing components allow students to take exams at their computers, and they save the instructor time by grading each exam automatically. These test banks are also available in Blackboard-compatible formats.
- **PowerPoint Presentations:** This text provides PowerPoint slides to accompany each chapter. Slides may be used to guide classroom presentations, to make available to students for chapter review, or to print as classroom handouts. Files are provided for every figure in the text. Instructors may use the files to customize PowerPoint slides, illustrate quizzes, or create handouts.
- **Solutions:** Solutions to "You Do It" exercises and all end-of-chapter exercises are available. Annotated solutions are provided for some of the multiple-choice Review Questions. For example, if students are likely to debate answer choices or not understand the choice deemed to be the correct one, a rationale is provided.

Acknowledgements

I would like to thank all of the people who helped to make this book a reality, including Dan Seiter, Development Editor; Alyssa Pratt, Senior Product Manager; Sreejith Govindan, Content Project Manager; and Chris Scriver and Serge Palladino, Quality Assurance Testers. 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 Lee Cottrell, Bradford School, Pittsburgh; Irene Edge, Kent State University; Susan Peterson, Henry Ford Community College; and Jackie Turner, Central Georgia Technical College.

Thanks, too, to my husband, Geoff, for his constant support and encouragement. Finally, this book is dedicated to Ruth LaFreniere, who brought us Stella, and Bob LaFreniere, who let her.

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 7, 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 7. The book clearly points out the few cases when an example is based on Java 7 and will not work with earlier versions of Java. You also need a text editor, such as Notepad. A few exercises ask you to use a browser, such as Internet Explorer.
- **Hardware:** If you are using Windows 7, the Java Web site suggests at least 128 MB of memory and at least 98 MB of disk space. For other operating system requirements, 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.

Analyzing a Java Application that Produces Console Output

You Do It

Your First Application

Now that you understand the basics of an application written in Java, you are ready to enter your own Java application into a text editor. It is a tradition among programmers that the first program you write in any language produces "Hello, world!" as its output. You will create such a program now. You can use any text editor, such as Notepad or TextPad, or a development environment, such as jGRA

21

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

It is best to use the simplest available text editor when wri processing programs save documents as much larger files as font styles and margin settings, which the Java compile

1. Start any text editor, and then open a new do
2. Type the class header as follows:

```
public class Hello
```

In this example, the class name is `Hello`. Yo want for the class. If you choose *Hello*, you alv *Hello*, and not as *hello*, because Java is case

3. Press **Enter** once, type **{**, press **Enter** again, a `main()` method between these curly braces. A convention used in this book is to place each cu align opening and closing curly brace pairs wit makes your code easier to read.
4. As shown in the shaded portion of Figure 1-9, between the curly braces, and then type a set

`n(String[] args)`

shell for the `Hello` class

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 1 Creating Java Programs

30

name is still stored on your computer. Before the new source code can execute, you must do the following:

1. Save the file with the changes (using the same filename).
2. Compile the class with the `javac` command. (Actually, you are *re*compiling the class.)
3. Interpret the class bytecode and execute the class using the `java` command.

Figure 1-19 shows the new output.

When you recompile a class, the original version of the compiled file with the .class extension is replaced, and the original version no longer exists. When you modify a class, you must decide whether you want to retain the original version. If you do, you must give the new version a new class name and a new filename.

Figure 1-19 Execution of modified `First` class

Once in a while, when you make a change to a Java class and then recompile and execute it, the old version still runs. The simplest solution is to delete the .class file and compile again. Programmers call this creating a **clean build.**

Watch the video *Compiling and Executing a Program*.

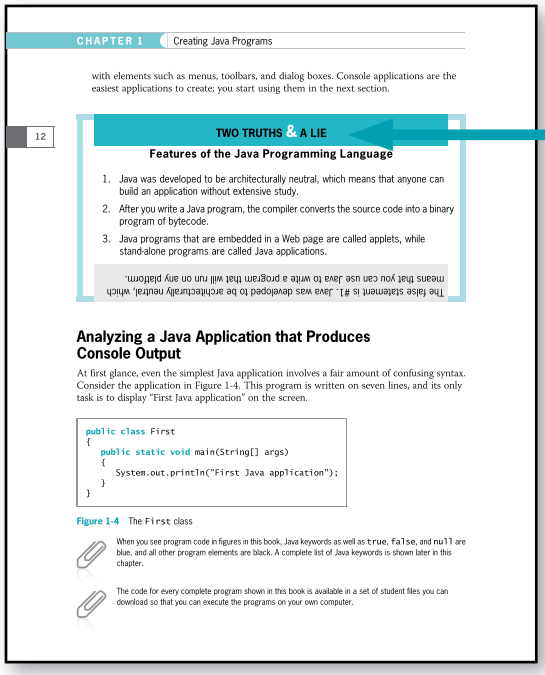
Correcting Logical Errors

Besides syntax errors, a second kind of error occurs when the syntax of the program is correct and the program compiles but produces incorrect results when you execute it. This type of error is a **logic error**, which is often more difficult to find and resolve. For example, Figure 1-20 shows the output of the execution of a successfully compiled program named `FirstBadOutput`. If you glance at the output too quickly, you might not notice that Java is misspelled. The compiler does not find spelling errors within a literal string; it is legitimate to produce any combination of letters as output.
Other examples of logic errors include

VIDEO LESSONS help explain important chapter concepts. Videos are part of the text's enhanced CourseMate site.

Figure 1-20 Output of `FirstBadOutput` program

The author does an excellent job clarifying what my students have historically had trouble with.
—Lee Cottrell, Bradford School, Pittsburgh



CHAPTER 1 Creating Java Programs

with elements such as menus, toolbars, and dialog boxes. Console applications are the easiest applications to create; you start using them in the next section.

12

TWO TRUTHS & A LIE

Features of the Java Programming Language

1. Java was developed to be architecturally neutral, which means that anyone can build an application without extensive study.
2. After you write a Java program, the compiler converts the source code into a binary program of bytecode.
3. Java programs that are embedded in a Web page are called applets, while stand-alone programs are called Java applications.

The false statement is #1. Java was developed to be architecturally neutral, which means that you can use Java to write a program that will run on any platform.

Analyzing a Java Application that Produces Console Output

At first glance, even the simplest Java application involves a fair amount of confusing syntax. Consider the application in Figure 1-4. This program is written on seven lines, and its only task is to display "First Java application" on the screen.

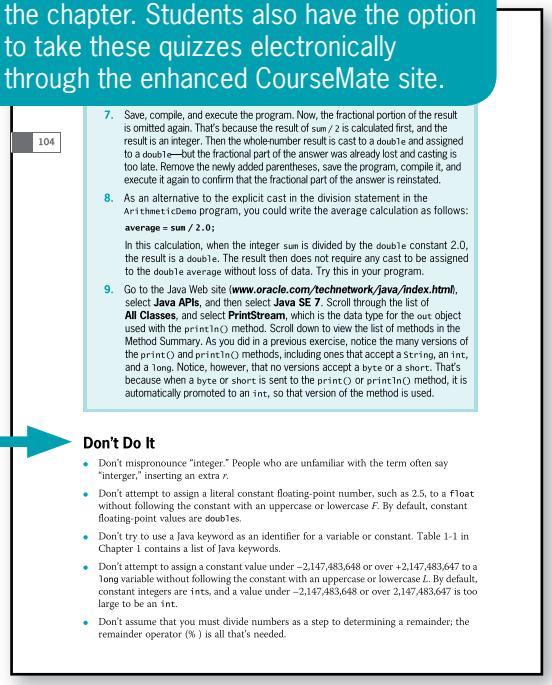
```
public class First
{
   public static void main(String[] args)
   {
      System.out.println("First Java application");
   }
}
```

Figure 1-4 The First class

When you see program code in figures in this book, Java keywords as well as `true`, `false`, and `null` are blue, and all other program elements are black. A complete list of Java keywords is shown later in this chapter.

The code for every complete program shown in this book is available in a set of student files you can download so that you can execute the programs on your own computer.

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 electronically through the enhanced CourseMate site.



104

7. Save, compile, and execute the program. Now, the fractional portion of the result is omitted again. That's because the result of `sum / 2` is calculated first, and the result is an integer. Then the whole-number result is cast to a `double` and assigned to a `double`—but the fractional part of the answer was already lost and casting is too late. Remove the newly added parentheses, save the program, compile it, and execute it again to confirm that the fractional part of the answer is reinstated.
8. As an alternative to the explicit cast in the division statement in the `ArithmeticDemo` program, you could write the average calculation as follows:

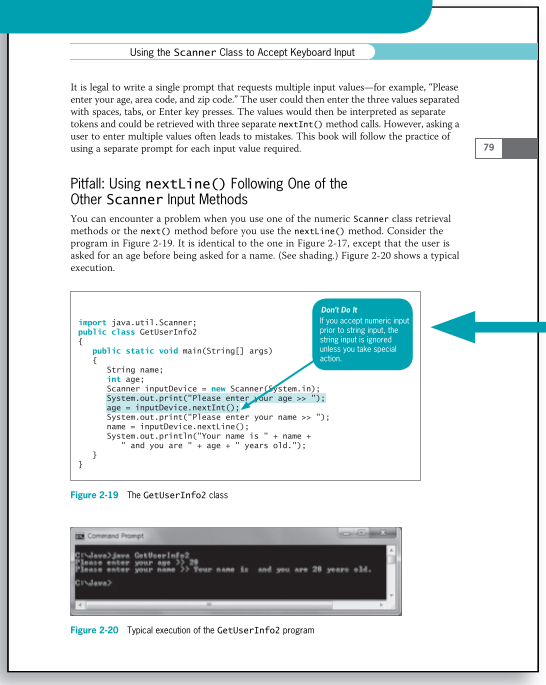
   `average = sum / 2.0;`

   In this calculation, when the integer `sum` is divided by the `double` constant 2.0, the result is a `double`. The result then does not require any cast to be assigned to the `double average` without loss of data. Try this in your program.
9. Go to the Java Web site (***www.oracle.com/technetwork/java/index.html***), select **Java APIs**, and then select **Java SE 7**. Scroll through the list of **All Classes**, and select **PrintStream**, which is the data type for the `out` object used with the `println()` method. Scroll down to view the list of methods in the Method Summary. As you did in a previous exercise, notice the many versions of the `print()` and `println()` methods, including ones that accept a `String`, an `int`, and a `long`. Notice, however, that no versions accept a `byte` or a `short`. That's because when a `byte` or `short` is sent to the `print()` or `println()` method, it is automatically promoted to an `int`, so that version of the method is used.

Don't Do It

- Don't mispronounce "integer." People who are unfamiliar with the term often say "interger," inserting an extra *r*.
- Don't attempt to assign a literal constant floating-point number, such as 2.5, to a `float` without following the constant with an uppercase or lowercase *F*. By default, constant floating-point values are `doubles`.
- Don't try to use a Java keyword as an identifier for a variable or constant. Table 1-1 in Chapter 1 contains a list of Java keywords.
- Don't attempt to assign a constant value under –2,147,483,648 or over +2,147,483,647 to a `long` variable without following the constant with an uppercase or lowercase *L*. By default, constant integers are `ints`, and a value under –2,147,483,648 or over 2,147,483,647 is too large to be an `int`.
- Don't assume that you must divide numbers as a step to determining a remainder; the remainder operator (`%`) is all that's needed.

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



Using the `Scanner` Class to Accept Keyboard Input

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 then be interpreted as separate tokens and could be retrieved with three separate `nextInt()` method calls. However, asking a user to enter multiple values often leads to mistakes. This book will follow the practice of using a separate prompt for each input value required.

79

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. (See shading.) 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

```
Command Prompt
C:\Java>java GetUserInfo2
Please enter your age >> 28
Please enter your name >> Your name is  and you are 28 years old.
C:\Java>
```

Figure 2-20 Typical execution of the GetUserInfo2 program

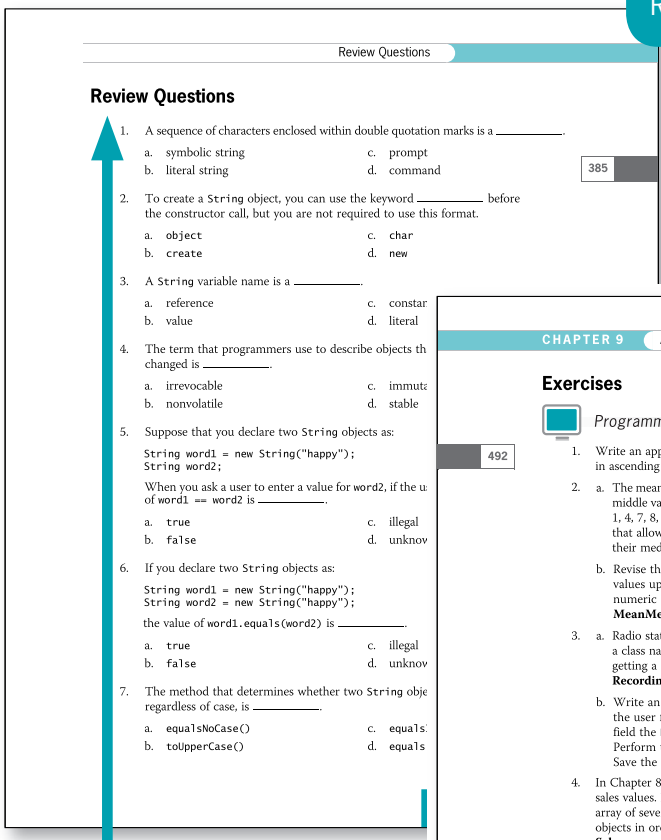
THE DON'T DO IT ICON illustrates how NOT to do something—for example, having a dead code path in a program. This icon provides a visual jolt to the student, emphasizing that particular figures are NOT to be emulated and making students more careful to recognize problems in existing code.

Assessment

I find the flow of information superior to [that of] other texts.

—Susan Peterson,
Henry Ford 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 programming exercises are available in the Instructor's Resource Kit.



Review Questions

Review Questions

1. A sequence of characters enclosed within double quotation marks is a _____.
   a. symbolic string
   b. literal string
   c. prompt
   d. command

385

2. To create a `String` object, you can use the keyword _____ before the constructor call, but you are not required to use this format.
   a. `object`
   b. `create`
   c. `char`
   d. `new`

3. A `String` variable name is a _____.
   a. reference
   b. value
   c. constar
   d. literal

4. The term that programmers use to describe objects th changed is _____.
   a. irrevocable
   b. nonvolatile
   c. immuta
   d. stable

5. Suppose that you declare two `String` objects as:

```
String word1 = new String("happy");
String word2;
```

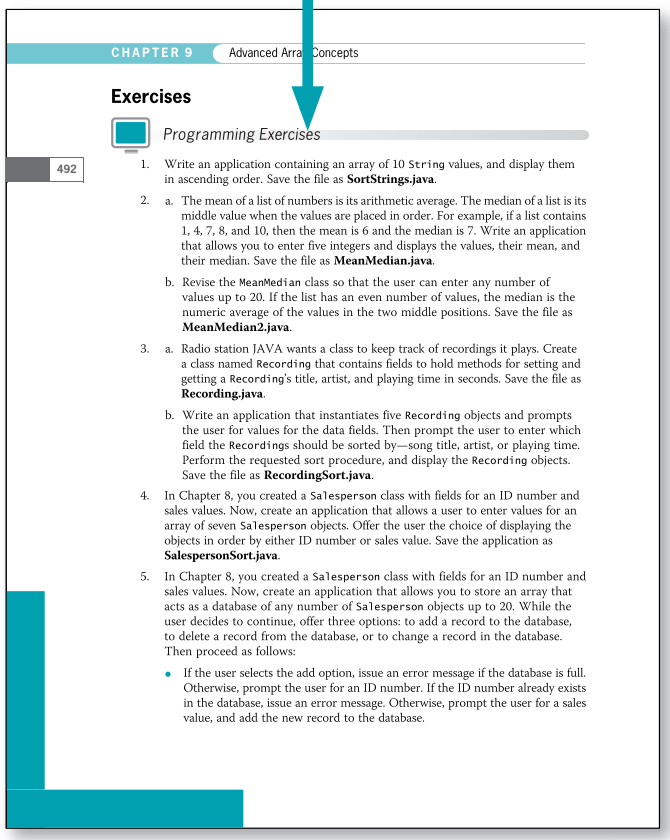
When you ask a user to enter a value for `word2`, if the u of `word1 == word2` is _____.
   a. `true`
   b. `false`
   c. illegal
   d. unknov

6. If you declare two `String` objects as:

```
String word1 = new String("happy");
String word2 = new String("happy");
```

the value of `word1.equals(word2)` is _____.
   a. `true`
   b. `false`
   c. illegal
   d. unknov

7. The method that determines whether two `String` obje regardless of case, is _____.
   a. `equalsNoCase()`
   b. `toUpperCase()`
   c. `equals`
   d. `equals`



CHAPTER 9 Advanced Arra Concepts

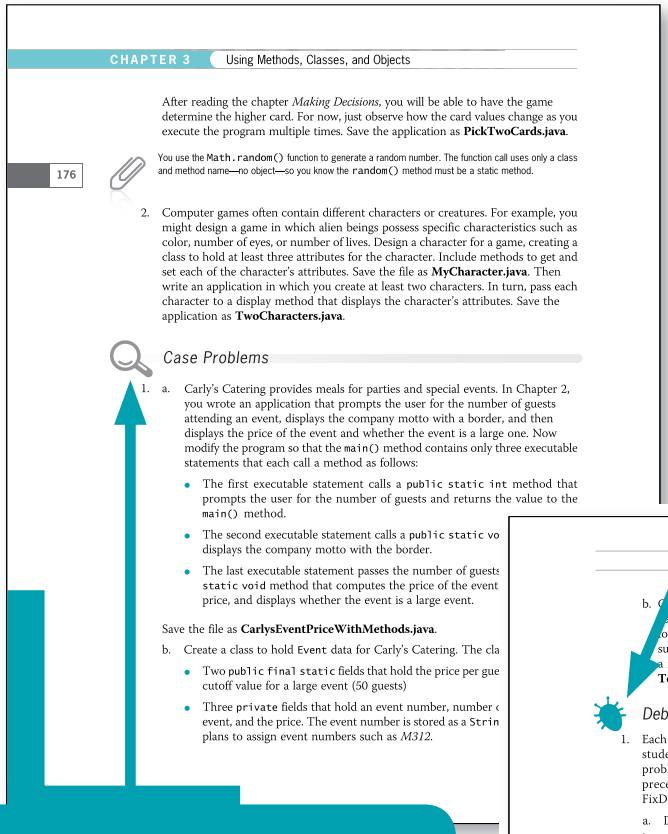
492

Exercises

Programming Exercises

1. Write an application containing an array of 10 `String` values, and display them in ascending order. Save the file as **SortStrings.java**.
2. a. The mean of a list of numbers is its arithmetic average. The median of a list is its middle value when the values are placed in order. For example, if a list contains 1, 4, 7, 8, and 10, then the mean is 6 and the median is 7. Write an application that allows you to enter five integers and displays the values, their mean, and their median. Save the file as **MeanMedian.java**.
   b. Revise the `MeanMedian` class so that the user can enter any number of values up to 20. If the list has an even number of values, the median is the numeric average of the values in the two middle positions. Save the file as **MeanMedian2.java**.
3. a. Radio station JAVA wants a class to keep track of recordings it plays. Create a class named `Recording` that contains fields to hold methods for setting and getting a `Recording`'s title, artist, and playing time in seconds. Save the file as **Recording.java**.
   b. Write an application that instantiates five `Recording` objects and prompts the user for values for the data fields. Then prompt the user to enter which field the `Recordings` should be sorted by—song title, artist, or playing time. Perform the requested sort procedure, and display the `Recording` objects. Save the file as **RecordingSort.java**.
4. In Chapter 8, you created a `Salesperson` class with fields for an ID number and sales values. Now, create an application that allows a user to enter values for an array of seven `Salesperson` objects. Offer the user the choice of displaying the objects in order by either ID number or sales value. Save the application as **SalespersonSort.java**.
5. In Chapter 8, you created a `Salesperson` class with fields for an ID number and sales values. Now, create an application that allows you to store an array that acts as a database of any number of `Salesperson` objects up to 20. While the user decides to continue, offer three options: to add a record to the database, to delete a record from the database, or to change a record in the database. Then proceed as follows:
   - If the user selects the add option, issue an error message if the database is full. Otherwise, prompt the user for an ID number. If the ID number already exists in the database, issue an error message. Otherwise, prompt the user for a sales value, and add the new record to the database.

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



CHAPTER 3 Using Methods, Classes, and Objects

After reading the chapter *Making Decisions*, you will be able to have the game determine the higher card. For now, just observe how the card values change as you execute the program multiple times. Save the application as **PickTwoCards.java**.

176

You use the `Math.random()` function to generate a random number. The function call uses only a class and method name—no object—so you know the `random()` method must be a static method.

2. Computer games often contain different characters or creatures. For example, you might design a game in which alien beings possess specific characteristics such as color, number of eyes, or number of lives. Design a character for a game, creating a class to hold at least three attributes for the character. Include methods to get and set each of the character's attributes. Save the file as **MyCharacter.java**. Then write an application in which you create at least two characters. In turn, pass each character to a display method that displays the character's attributes. Save the application as **TwoCharacters.java**.

Case Problems

1. a. Carly's Catering provides meals for parties and special events. In Chapter 2, you wrote an application that prompts the user for the number of guests attending an event, displays the company motto with a border, and then displays the price of the event and whether the event is a large one. Now modify the program so that the `main()` method contains only three executable statements that each call a method as follows:

- The first executable statement calls a `public static int` method that prompts the user for the number of guests and returns the value to the `main()` method.
- The second executable statement calls a `public static vo` displays the company motto with the border.
- The last executable statement passes the number of guests `static void` method that computes the price of the event price, and displays whether the event is a large event.

Save the file as **CarlysEventPriceWithMethods.java**.

b. Create a class to hold `Event` data for Carly's Catering. The cla

- Two `public final static` fields that hold the price per gue cutoff value for a large event (50 guests)
- Three `private` fields that hold an event number, number event, and the price. The event number is stored as a `Strin` plans to assign event number such as *M312*.

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* and through the CourseMate available for this text. These files are also available to instructors through *login.cengage.com*.

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

Exercises

b. Create a class named `TestPainting` whose `main()` method declares three `Painting` items. Create a method that prompts the user for and accepts values for two of the `Painting` objects, and leave the third with the default values supplied by the constructor. Then display each completed object. Finally, display a message that explains the gallery commission rate. Save the application as **TestPainting.java**.

175

Debugging Exercises

1. Each of the following files saved in the Chapter03 folder in your downloadable student files has syntax and/or logic errors. In each case, determine and fix the problem. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, DebugThree1.java will become FixDebugThree1.java.

a. DebugThree1.java
b. DebugThree2.java
c. DebugThree3.java
d. DebugThree4.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. Playing cards are used in many computer games, including versions of such classics as solitaire, hearts, and poker. Design a `Card` class that contains a character data field to hold a suit (*s* for spades, *h* for hearts, *d* for diamonds, or *c* for clubs) and an integer data field for a value from 1 to 13. (When you learn more about string handling in the chapter *Characters, Strings, and the* `StringBuilder`, you can modify the class to hold words for the suits, such as *spades* or *hearts*, as well as words for some of the values—for example, *ace* or *king*.) Include get and set methods for each field. Save the class as **Card.java**.

Write an application that randomly selects two playing cards and displays their values. Simply assign a suit to each of the cards, but generate a random number for each card's value. Appendix D contains information on generating random numbers. To fully understand the process, you must learn more about Java classes and methods. However, for now, you can copy the following statements to generate a random number between 1 and 13 and assign it to a variable:

```
final int CARDS_IN_SUIT = 13;
myValue = ((int)(Math.random() * 100) % CARDS_IN_SUIT + 1);
```

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

In this chapter, you will:

- 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 logical 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 application software.

All computer programs ultimately are converted to machine language. **Machine language**, or **machine code**, is the most basic set of instructions that a computer can execute. Each type of processor 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.

Machine language is a **low-level programming language**, or one that corresponds closely to a computer processor's circuitry. Low-level languages require you to use memory addresses for specific machines when you create commands. This means that low-level languages are difficult to use and must be customized for every type of machine on which a program runs.

Fortunately, programming has evolved into an easier task because of the development of high-level programming languages. A **high-level programming language** allows you to use a vocabulary of reasonable terms, such as *read*, *write*, or *add*, instead of the sequences of 1s and 0s that perform these tasks. High-level languages also allow you to assign single-word, intuitive names to areas of computer memory, such as `hoursWorked` or `rateOfPay`, rather than having to remember the memory locations. Java is a high-level programming language.

Each high-level language has its own **syntax**, or rules of the language. 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 really are learning the vocabulary and syntax rules for that language.

Using a programming language, programmers write a series of **program statements**, similar to English sentences, to carry out the tasks they want the program to perform. 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, high-level language programmers use a computer program called a **compiler** or **interpreter** to translate their language statements into machine language. A compiler translates an entire program before carrying out the statement, or **executing** it, 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 are 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. Subsequently, the programmer can correct the error and attempt another translation by compiling or interpreting the program again. Locating and repairing all syntax errors is the first part of the process of **debugging** a program—freeing the program of all errors. 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.

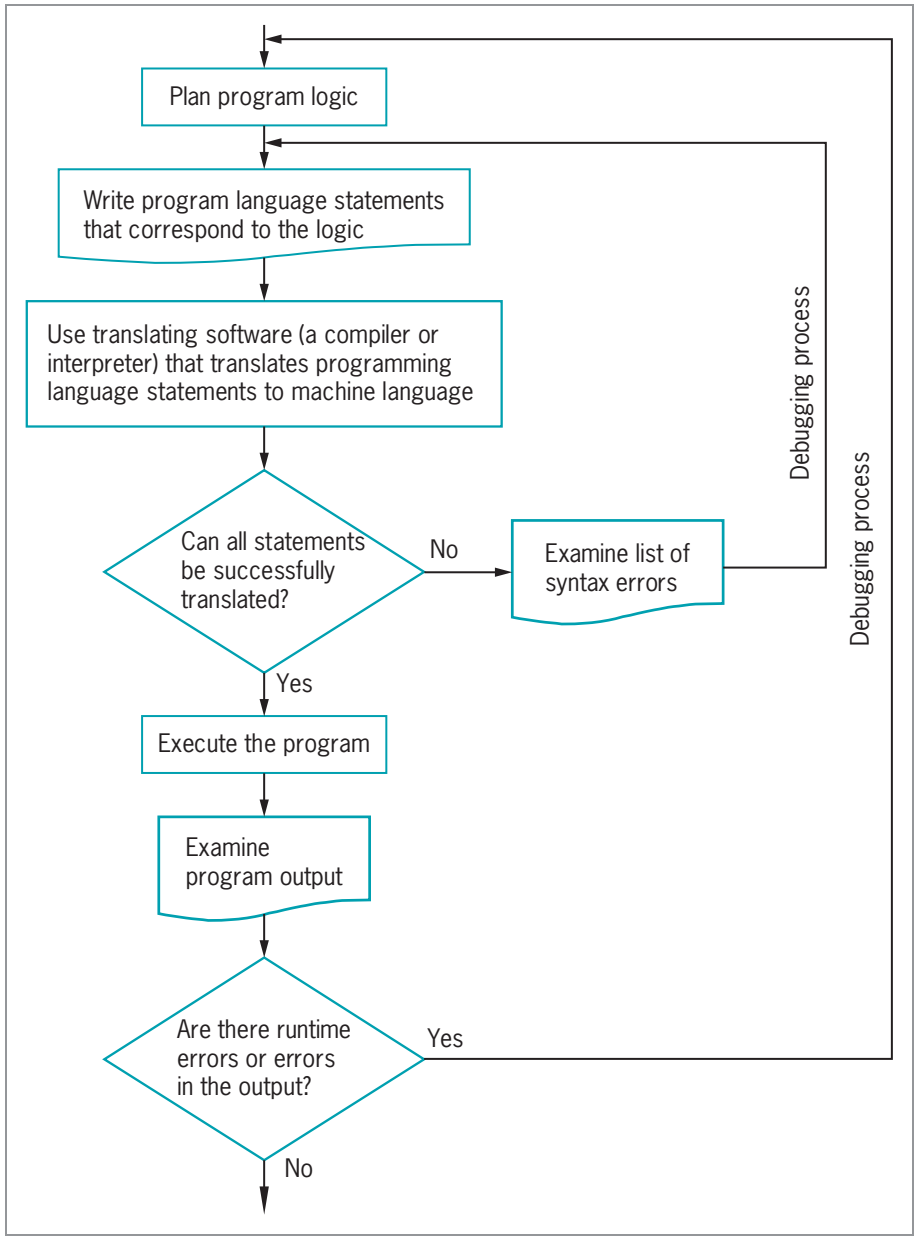


Figure 1-1 The program development process

As Figure 1-1 shows, you might be able to use a computer language's syntax correctly but still have errors to correct. In addition to learning the correct syntax for a particular language, a programmer must also understand computer programming logic. When you develop a program of any significant size, you should plan its logic before you write any program statements. Correct logic requires that all the right commands be issued in the appropriate order. Examples of logical errors include multiplying two values when you meant to divide them or producing output prior to obtaining the appropriate input.

Correcting logical errors is the second part of the debugging process and is much more difficult than correcting syntax errors. Syntax errors are discovered when you compile a program, but often you can identify logical errors only when you examine a program's first output. For example, if you know an employee's paycheck should contain the value $5,000, but you see that it holds $50 or $50,000 after you execute a payroll program, a logical error has occurred. Tools that help you visualize and understand logic are presented in the chapter *Making Decisions*.

Programmers call some logical 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.

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 compiler executes each program statement as soon as it is translated, whereas an interpreter translates all of a program's statements before executing any.
3. A syntax error occurs when you misuse a language; locating and repairing all syntax errors is part of the process of debugging a program.

The false statement is #2. A compiler translates an entire program before carrying out any statements, whereas an interpreter translates one program statement at a time, executing a statement as soon as it is translated.

Comparing Procedural and Object-Oriented Programming Concepts

Two popular approaches to writing computer programs are procedural programming and object-oriented programming.

Procedural Programming

Procedural programming is a style of programming in which operations are executed one after another in sequence. In procedural applications, you create names for computer memory locations that can hold values—for example, numbers and text—in electronic form. The named computer memory locations are called **variables** because they hold values that might vary. For example, a payroll program written for a company might contain a variable named `rateOfPay`. The memory location referenced by the name `rateOfPay` might contain different values (a different value for every employee of the company) at different times. During the execution of the payroll program, each value stored under the name `rateOfPay` might have many operations performed on it—the value might be read from an input device, the value might be multiplied by another variable representing hours worked, and the value might be printed on paper. 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`. A procedural program defines the variable memory locations and then calls a series of procedures to input, manipulate, and output the values stored in those locations. When a program **calls a procedure**, the current logic is temporarily abandoned so that the procedure's commands can execute. A single procedural program often contains hundreds of variables and 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 from those classes; and creating applications that use those objects. After creation, classes can be reused repeatedly to develop new programs. Thinking in an object-oriented manner involves envisioning program components as objects that belong to classes and that are similar to concrete objects in the real world; then, you can manipulate the objects and have them interrelate with each other to achieve a desired result.

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 to help prevent traffic tie-ups. By creating a model with objects such as cars and pedestrians that contain their own data and rules for behavior, the simulation can be set in motion. For example, each car object has a specific current speed and a procedure for changing that speed. By creating a model of city traffic using objects, a computer can create a simulation of a real city at rush hour.

Creating a GUI environment for users also is 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, do not assume that 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.

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 term that describes 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, so does a class definition exist 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. When you create an object, you **instantiate** it. 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, of course, the same values for those attributes. One `Automobile` might be a 2009 white Ford Taurus and another might be a 2014 red Chevrolet Camaro. Similarly, your dog has the properties of all `Dogs`, including a breed, name, age, and whether his shots are current. The values of the properties of an object are also referred to as the object's **state**. In other words, you can think of objects as roughly equivalent to nouns, 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 would probably be named `Automobile`, and the class for dogs would probably be named `Dog`. However, following this convention is not required to produce a workable program.

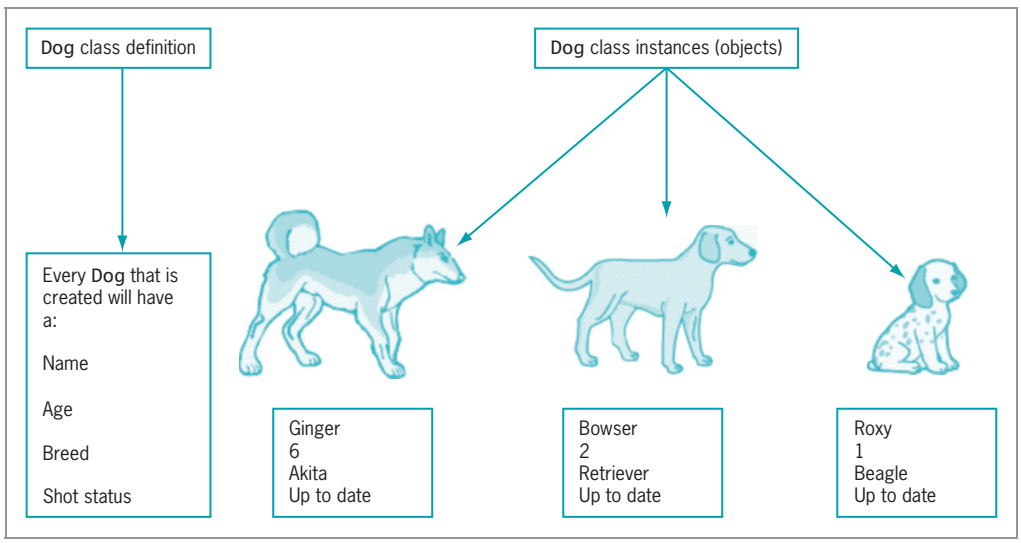


Figure 1-2 A class definition and some objects created from it

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, being filled with gasoline, and being washed. Some methods can ascertain certain attributes, such as the current speed of an `Automobile` and the status of its gas tank. Similarly, a `Dog` can walk or run, eat food, and get a bath, and there are methods to determine how hungry the `Dog` is or what its name is. GUI operating system components can be maximized, minimized, and dragged. 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 that are then used much like real-world 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 is 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 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 mechanism for and 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 `Automobile` systems work. 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 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`, `turn` and `stop` methods 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.

Watch the video *Object-Oriented Programming*.

TWO TRUTHS & A LIE

Comparing Procedural and Object-Oriented Programming Concepts

1. An instance of a class is a created object that possesses the attributes and methods described in the class definition.
2. Encapsulation protects data by hiding it within an object.
3. Polymorphism is the ability to create classes that share the attributes and methods of existing classes, but with more specific features.

The false statement is #3. Inheritance is the ability to create classes that share the attributes and methods of existing classes but with more specific features; polymorphism describes the ability to use one term to cause multiple actions.