

Custom Edition for JAVA 1
PROG10082 — Object Oriented Programming I

Introduction to JAVA PROGRAMMING

Y. Daniel Liang

void printmessage (void)
r Format Exception

<textarea>
type="hidden" />>
dy> my first string

rt java void printmessage (void)

<input type="button" value="function example" />

ng>import java.awt.printmessage (void)

Faculty of Applied Sciences
and Technology



Java Quick Reference

Console Input

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
long longValue = input.nextLong();
double doubleValue = input.nextDouble();
float floatValue = input.nextFloat();
String string = input.next();
```

Console Output

```
System.out.println(anyValue);
```

GUI Input Dialog

```
String string = JOptionPane.showInputDialog(
    "Enter input");
int intValue = Integer.parseInt(string);
double doubleValue =
    Double.parseDouble(string);
```

Message Dialog

```
JOptionPane.showMessageDialog(null,
    "Enter input");
```

Primitive Data Types

| | |
|----------------|------------|
| byte | 8 bits |
| short | 16 bits |
| int | 32 bits |
| long | 64 bits |
| float | 32 bits |
| double | 64 bits |
| char | 16 bits |
| boolean | true/false |

Arithmetic Operators

| | |
|--------------|----------------|
| + | addition |
| - | subtraction |
| * | multiplication |
| / | division |
| % | remainder |
| ++var | preincrement |
| --var | predecrement |
| var++ | postincrement |
| var-- | postdecrement |

Assignment Operators

| | |
|-----------|---------------------------|
| = | assignment |
| += | addition assignment |
| -= | subtraction assignment |
| *= | multiplication assignment |
| /= | division assignment |
| %= | remainder assignment |

Relational Operators

| | |
|--------------|--------------------------|
| < | less than |
| <= | less than or equal to |
| > | greater than |
| >= | greater than or equal to |
| == | equal to |
| != | not equal |

Logical Operators

| | |
|-------------------|-------------------|
| && | short circuit AND |
| | short circuit OR |
| ! | NOT |
| ^ | exclusive OR |

if Statements

```
if (condition) {
    statements;
}

if (condition) {
    statements;
}
else {
    statements;
}

if (condition1) {
    statements;
}
else if (condition2) {
    statements;
}
else {
    statements;
}
```

switch Statements

```
switch (intExpression) {
    case value1:
        statements;
        break;
    ...
    case valuen:
        statements;
        break;
    default:
        statements;
}
```

Loop Statements

```
while (condition) {
    statements;
}

do {
    statements;
} while (condition);

for (init; condition;
    adjustment) {
    statements;
}
```

Companion Web site: www.pearsonhighered.com/Liang

CONTENTS

| | | |
|----------------------------------|--|---------------------------|
| <u>Chapter 1</u> | Introduction to Computers, Programs, and Java | <u>1</u> |
| 1.1 | Introduction | <u>2</u> |
| 1.2 | What Is a Computer? | <u>2</u> |
| 1.3 | Programs | <u>5</u> |
| 1.4 | Operating Systems | <u>7</u> |
| 1.5 | Java, World Wide Web, and Beyond | <u>8</u> |
| 1.6 | The Java Language Specification, API, JDK, and IDE | <u>10</u> |
| 1.7 | A Simple Java Program | <u>11</u> |
| 1.8 | Creating, Compiling, and Executing a Java Program | <u>13</u> |
| 1.9 | (GUI) Displaying Text in a Message Dialog Box | <u>16</u> |
| <u>Chapter 2</u> | Elementary Programming | <u>23</u> |
| 2.1 | Introduction | <u>24</u> |
| 2.2 | Writing Simple Programs | <u>24</u> |
| 2.3 | Reading Input from the Console | <u>26</u> |
| 2.4 | Identifiers | <u>29</u> |
| 2.5 | Variables | <u>29</u> |
| 2.6 | Assignment Statements and Assignment Expressions | <u>30</u> |
| 2.7 | Named Constants | <u>31</u> |
| 2.8 | Numeric Data Types and Operations | <u>32</u> |
| 2.9 | Problem: Displaying the Current Time | <u>37</u> |
| 2.10 | Shorthand Operators | <u>39</u> |
| 2.11 | Numeric Type Conversions | <u>41</u> |
| 2.12 | Problem: Computing Loan Payments | <u>43</u> |
| 2.13 | Character Data Type and Operations | <u>44</u> |
| 2.14 | Problem: Counting Monetary Units | <u>47</u> |
| 2.15 | The String Type | <u>50</u> |
| 2.16 | Programming Style and Documentation | <u>51</u> |
| 2.17 | Programming Errors | <u>53</u> |
| 2.18 | (GUI) Getting Input from Input Dialogs | <u>55</u> |

| | | |
|------------------|---|---------------------|
| <u>Chapter 3</u> | Selections | 71 |
| 3.1 | Introduction | 72 |
| 3.2 | boolean Data Type | 72 |
| 3.3 | Problem: A Simple Math Learning Tool | 73 |
| 3.4 | if Statements | 74 |
| 3.5 | Problem: Guessing Birthdays | 75 |
| 3.6 | Two-Way if Statements | 79 |
| 3.7 | Nested if Statements | 80 |
| 3.8 | Common Errors in Selection Statements | 81 |
| 3.9 | Problem: An Improved Math Learning Tool | 82 |
| 3.10 | Problem: Computing Body Mass Index | 84 |
| 3.11 | Problem: Computing Taxes | 85 |
| 3.12 | Logical Operators | 88 |
| 3.13 | Problem: Determining Leap Year | 90 |
| 3.14 | Problem: Lottery | 91 |
| 3.15 | switch Statements | 93 |
| 3.16 | Conditional Expressions | 95 |
| 3.17 | Formatting Console Output | 95 |
| 3.18 | Operator Precedence and Associativity | 97 |
| 3.19 | (GUI) Confirmation Dialogs | 98 |
| <u>Chapter 4</u> | Loops | 115 |
| 4.1 | Introduction | 116 |
| 4.2 | The while Loop | 116 |
| 4.3 | The do-while Loop | 124 |
| 4.4 | The for Loop | 126 |
| 4.5 | Which Loop to Use? | 128 |
| 4.6 | Nested Loops | 129 |
| 4.7 | Minimizing Numeric Errors | 130 |
| 4.8 | Case Studies | 131 |
| 4.9 | Keywords <i>break</i> and <i>and continue</i> | 135 |
| 4.10 | (GUI) Controlling a Loop with a Confirmation Dialog | 139 |
| <u>Chapter 5</u> | Methods | 155 |

| | | |
|----------------------------------|--|---------------------|
| 5.1 | Introduction | 156 |
| 5.2 | Defining a Method | 156 |
| 5.3 | Calling a Method | 158 |
| 5.4 | void Method Example | 160 |
| 5.5 | Passing Parameters by Values | 162 |
| 5.6 | Modularizing Code | 165 |
| 5.7 | Problem: Converting Decimals to Hexadecimals | 167 |
| 5.8 | Overloading Methods | 168 |
| 5.9 | The Scope of Variables | 171 |
| 5.10 | The Math Class | 172 |
| 5.11 | Case Study: Generating Random Characters | 175 |
| 5.12 | Method Abstraction and Stepwise Refinement | 176 |
| Chapter 8 | Objects and Classes | 263 |
| 8.1 | Introduction | 264 |
| 8.2 | Defining Classes for Objects | 264 |
| 8.3 | Example: Defining Classes and Creating Objects | 266 |
| 8.4 | Constructing Objects Using Constructors | 270 |
| 8.5 | Accessing Objects via Reference Variables | 270 |
| 8.6 | Using Classes from the Java Library | 274 |
| 8.7 | Static Variables, Constants, and Methods | 278 |
| 8.8 | Visibility Modifiers | 282 |
| 8.9 | Data Field Encapsulation | 283 |
| 8.10 | Passing Objects to Methods | 286 |
| 8.11 | Array of Objects | 287 |
| Chapter 9 | Strings and Text I/O | 301 |
| 9.1 | Introduction | 302 |
| 9.2 | The String Class | 302 |
| 9.3 | The Character Class | 313 |
| 9.4 | The StringBuilder/StringBuffer Class | 315 |
| 9.5 | Command-Line Arguments | 320 |
| 9.6 | The File Class | 322 |
| 9.7 | File Input and Output | 325 |

xv

xvi

9.8 (GUI) File Dialogs

[329](#)

APPENDIXES

| | |
|--|----------------------|
| Appendix A Java Keywords | 1309 |
| Appendix B The ASCII Character Set | 1312 |
| Appendix C Operator Precedence Chart | 1314 |
| Appendix D Java Modifiers | 1316 |
| Appendix E Special Floating-Point Values | 1318 |
| Appendix F Number Systems | 1319 |

xvi

CHAPTER 1 INTRODUCTION TO COMPUTERS, PROGRAMS, AND JAVA

Objectives

- To review computer basics, programs, and operating systems (§§[1.2–1.4](#)).
- To explore the relationship between Java and the World Wide Web ([§1.5](#)).
- To distinguish the terms API, IDE, and JDK ([§1.6](#)).
- To write a simple Java program ([§1.7](#)).
- To display output on the console ([§1.7](#)).
- To explain the basic syntax of a Java program ([§1.7](#)).
- To create, compile, and run Java programs ([§1.8](#)).
- (GUI) To display output using the `JOptionPane` output dialog boxes ([§1.9](#)).

1.1 Introduction

You use word processors to write documents, Web browsers to explore the Internet, and email programs to send email. These are all examples of software that runs on computers. Software is developed using programming languages. There are many programming languages—so *why Java?* The answer is that Java enables users to develop and deploy applications on the Internet for servers, desktop computers, and small hand-held devices. The future of computing is being profoundly

influenced by the Internet, and Java promises to remain a big part of that future. Java is *the* Internet programming language.

why Java?

You are about to begin an exciting journey, learning a powerful programming language. At the outset, it is helpful to review computer basics, programs, and operating systems and to become familiar with number systems. If you are already familiar with such terms as CPU, memory, disks, operating systems, and programming languages, you may skip the review in §§[1.2–1.4](#).

1.2 What Is a Computer?

A computer is an electronic device that stores and processes data. It includes both *hardware* and *software*. In general, hardware comprises the visible, physical elements of the computer, and software provides the invisible instructions that control the hardware and make it perform specific tasks. Writing instructions for computers to perform is called computer programming. Knowing computer hardware isn't essential to your learning a programming language, but it does help you understand better the effect of the program instructions. This section introduces computer hardware components and their functions.

hardware

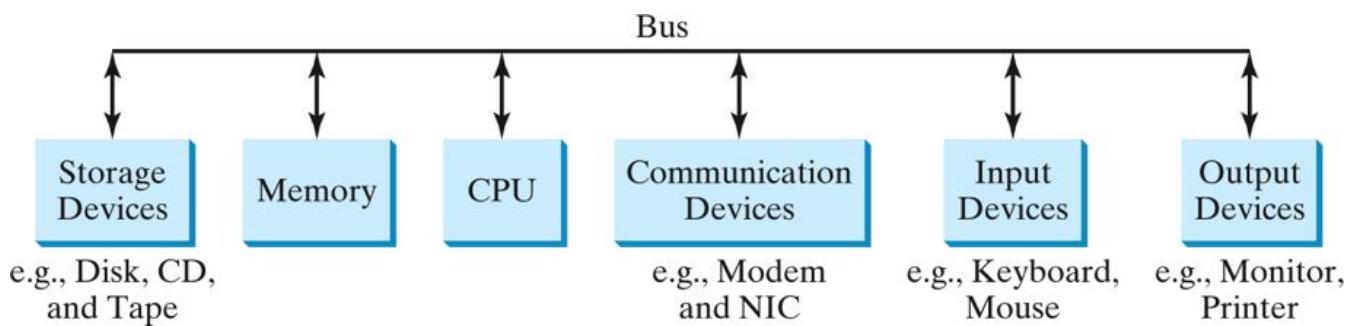
software

A computer consists of the following major hardware components ([Figure 1.1](#)):

- Central processing unit (CPU)

- Memory (main memory)
- Storage devices (e.g., disks, CDs, tapes)
- Input and output devices (e.g., monitors, keyboards, mice, printers)
- Communication devices (e.g., modems and network interface cards (NICs))

FIGURE 1.1 A computer consists of CPU, memory, storage devices, input devices, output devices, and communication devices.



The components are connected through a subsystem called a *bus* that transfers data or power between them.

bus

1.2.1 Central Processing Unit

The *central processing unit* (CPU) is the computer's brain. It retrieves instructions from memory and executes them. The CPU usually has two components: a *control unit* and an *arithmetic/logic unit*. The control unit controls and coordinates the actions of the other components. The arithmetic/logic unit performs numeric operations

components. The arithmetic/logic unit performs numeric operations (addition, subtraction, multiplication, division) and logical operations (comparisons).

CPU

Today's CPU is built on a small silicon semiconductor chip having millions of transistors. Every computer has an internal clock, which emits electronic pulses at a constant rate. These pulses are used to control and synchronize the pace of operations. The higher the clock speed, the more instructions are executed in a given period of time. The unit of measurement of clock speed is the *hertz (Hz)*, with 1 hertz equaling 1 pulse per second. The clock speed of a computer is usually stated in megahertz (MHz) (1 MHz is 1 million Hz). CPU speed has been improved continuously. Intel's Pentium 3 Processor runs at about 500 megahertz and Pentium 4 Processor at about 3 gigahertz (GHz) (1 GHz is 1000 MHz).

speed

hertz

megahertz

gigahertz

1.2.2 Memory

To store and process information, computers use *off* and *on* electrical states, referred to by convention as *0* and *1*. These 0s and 1s are interpreted as digits in the binary number system and called *bits (binary digits)*. Data of various kinds, such as numbers, characters, and

strings, are encoded as series of bits. Data and program instructions for the CPU to execute are stored as groups of bits, or bytes, each byte composed of eight bits, in a computer's *memory*. A memory unit is an ordered sequence of *bytes*, as shown in [Figure 1.2](#).

bit

byte

FIGURE 1.2 Memory stores data and program instructions.

| Memory address | Memory content | |
|----------------|----------------|----------------------------|
| . | . | |
| . | . | |
| . | . | |
| 2000 | 01001010 | Encoding for character 'J' |
| 2001 | 01100001 | Encoding for character 'a' |
| 2002 | 01110110 | Encoding for character 'v' |
| 2003 | 01100001 | Encoding for character 'a' |
| 2004 | 00000011 | Encoding for number 3 |
| . | . | |
| | | |

The programmer need not be concerned about the encoding and decoding of data, which the system performs automatically, based on the encoding scheme. In the popular ASCII encoding scheme, for example, character **&J&** is represented by **01001010** in one byte.

A byte is the minimum storage unit. A small number such as **3** can be stored in a single byte. To store a number that cannot fit into a single byte, the computer uses several adjacent bytes. No two data items can share or split the same byte.

A memory byte is never empty, but its initial content may be meaningless to your program. The current content of a memory byte is lost whenever new information is placed in it.

A program and its data must be brought to memory before they can be executed.

Every byte has a unique address. The address is used to locate the byte for storing and retrieving data. Since bytes can be accessed in any order, the memory is also referred to as *random-access memory (RAM)*. Today's personal computers usually have at least 1 gigabyte of RAM. Computer storage size is measured in bytes, kilobytes (KB), megabytes (MB), gigabytes (GB), and terabytes (TB). A *kilobyte* is $2^{10} = 1024$, about 1000 bytes, a *megabyte* is $2^{20} = 1048576$, about 1 million bytes, a *gigabyte* is about 1 billion bytes, and a terabyte is about 1000 gigabytes. Like the CPU, memory is built on silicon semiconductor chips having thousands of transistors embedded on their surface. Compared to CPU chips, memory chips are less complicated, slower, and less expensive.

RAM

megabyte

1.2.3 Storage Devices

Memory is volatile, because information is lost when the power is turned off. Programs and data are permanently stored on storage devices and are moved, when the computer actually uses them, to memory, which is much faster than storage devices.

There are four main types of storage devices:

- Disk drives
- CD drives (CD-R, CD-RW, and DVD)
- Tape drives
- USB flash drives

Drives are devices for operating a medium, such as disks, CDs, and tapes.

drive

Disks

Each computer has at least one hard drive. *Hard disks* are for permanently storing data and programs. The hard disks of the latest PCs store from 80 to 250 gigabytes. Often disk drives are encased inside the computer. Removable hard disks are also available.

hard disk

CDs and DVDs

CD stands for compact disk. There are two types of CD drives: CD-R and CD-RW. A *CD-R* is for read-only permanent storage; the user cannot modify its contents once they are recorded. A *CD-RW* can be used like a hard disk and can be both read and rewritten. A single CD can hold up to 700 MB. Most software is distributed through CD-ROMs. Most new PCs are equipped with a CD-RW drive that can work with both CD-R and CD-RW.

CD-R

CD-RW

DVD stands for digital versatile disc or digital video disk. DVDs and CDs look alike, and you can use either to store data. A DVD can hold more information than a CD. A standard DVD's storage capacity is 4.7 GB.

Tapes

Tapes are mainly used for backup of data and programs. Unlike disks and CDs, tapes store information sequentially. The computer must retrieve information in the order it was stored. Tapes are very slow. It would take one to two hours to back up a 1-gigabyte hard disk. The new trend is to back up data using flash drives or external hard disks.

USB Flash Drives

USB flash drives are devices for storing and transporting data. A flash drive is small—about the size of a pack of gum. It acts like a

portable hard drive that can be plugged into your computer's USB port. USB flash drives are currently available with up to 32 GB storage capacity.

1.2.4 Input and Output Devices

Input and output devices let the user communicate with the computer. The common input devices are *keyboards* and *mice*. The common output devices are *monitors* and *printers*.

The Keyboard

A computer *keyboard* resembles a typewriter keyboard with extra keys added for certain special functions.

Function keys are located at the top of the keyboard and are numbered with prefix F. Their use depends on the software.

function key

A *modifier key* is a special key (e.g., *Shift*, *Alt*, *Ctr*) that modifies the normal action of another key when the two are pressed in combination.

modifier key

The *numeric keypad*, located on the right-hand corner of the keyboard, is a separate set of keys for quick input of numbers.

numeric keypad

Arrow keys, located between the main keypad and the numeric keypad, are used to move the cursor up, down, left, and right.

The *Insert, Delete, Page Up, and Page Down keys*, located above the arrow keys, are used in word processing for performing insert, delete, page up, and page down.

The Mouse

A *mouse* is a pointing device. It is used to move an electronic pointer called a cursor around the screen or to click on an object on the screen to trigger it to respond.

The Monitor

The *monitor* displays information (text and graphics). The screen resolution and dot pitch determine the quality of the display.

The *screen resolution* specifies the number of pixels per square inch. Pixels (short for “picture elements”) are tiny dots that form an image on the screen. A common resolution for a 17-inch screen, for example, is 1024 pixels wide and 768 pixels high. The resolution can be set manually. The higher the resolution, the sharper and clearer the image is.

screen resolution

The *dot pitch* is the amount of space between pixels in millimeters. The smaller the dot pitch, the better the display.

dot pitch

1.2.5 Communication Devices

Computers can be networked through communication devices, such as the dialup modem (*modulator/demodulator*), DSL, cable modem, network interface card, and wireless. A dialup modem uses a phone line and can transfer data at a speed up to 56,000 bps (bits per second). A *DSL* (digital subscriber line) also uses a phone line and can transfer data twenty times faster. A cable modem uses the TV cable line maintained by the cable company and is as fast as a DSL. A network interface card (NIC) is a device that connects a computer to a local area network (LAN). The *LAN* is commonly used in universities and business and government organizations. A typical *NIC* called *10BaseT* can transfer data at 10 *mbps* (million bits per second). Wireless is becoming popular. Every laptop sold today is equipped with a wireless adapter that enables the computer to connect with the Internet.

modem

DSL

NIC

LAN

mbps

1.3 Programs

Computer *programs*, known as *software*, are instructions to the computer, telling it what to do. Computers do not understand human languages, so you need to use computer languages in computer programs. *Programming* is the creation of a program that is executable by a computer and performs the required tasks.

software

programming

A computer's native language, which differs among different types of computers, is its *machine language*—a set of built-in primitive instructions. These instructions are in the form of binary code, so in telling the machine what to do, you have to enter binary code.

Programming in machine language is a tedious process. Moreover, the programs are highly difficult to read and modify. For example, to add two numbers, you might have to write an instruction in binary like this:

machine language

1101101010011010

Assembly language is a low-level programming language in which a mnemonic is used to represent each of the machine-language instructions. For example, to add two numbers, you might write an instruction in assembly code like this:

ADD F3 R1, R2, R3

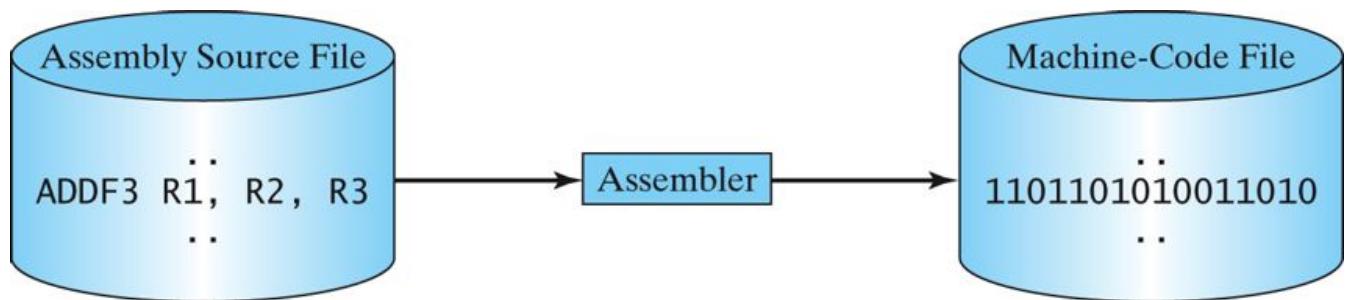
assembly language

5
6

Assembly languages were developed to make programming easy. However, since the computer cannot understand assembly language, a program called an *assembler* is used to convert assembly-language programs into machine code, as shown in [Figure 1.3](#).

assembler

FIGURE 1.3 Assembler translates assembly-language instructions to machine code.



Assembly programs are written in terms of machine instructions with easy-to-remember mnemonic names. Since assembly language is machine dependent, an assembly program can be executed only on a particular kind of machine. The high-level languages were developed in order to transcend platform specificity and make programming easier.

The *high-level languages* are English-like and easy to learn and program. Here, for example, is a high-level language statement that computes the area of a circle with radius 5:

```
area = 5 * 5 * 3.1415;
```

high-level language

Among the more than one hundred high-level languages, the following are well known:

- COBOL (COmmon Business Oriented Language)
- FORTRAN (FORmula TRANslation)
- BASIC (Beginner's All-purpose Symbolic Instruction Code)
- Pascal (named for Blaise Pascal)
- Ada (named for Ada Lovelace)
- C (developed by the designer of B)
- Visual Basic (Basic-like visual language developed by Microsoft)
- Delphi (Pascal-like visual language developed by Borland)
- C++ (an object-oriented language, based on C)
- C# (a Java-like language developed by Microsoft)
- **Java**

Each of these languages was designed for a specific purpose. COBOL was designed for business applications and is used primarily for business data processing. FORTRAN was designed for mathematical computations and is used mainly for numeric computations. BASIC was designed to be learned and used easily. Ada was developed for the Department of Defense and is used mainly in defense projects. C combines the power of an assembly language with the ease of use and portability of a high-level language. Visual Basic and Delphi are used in

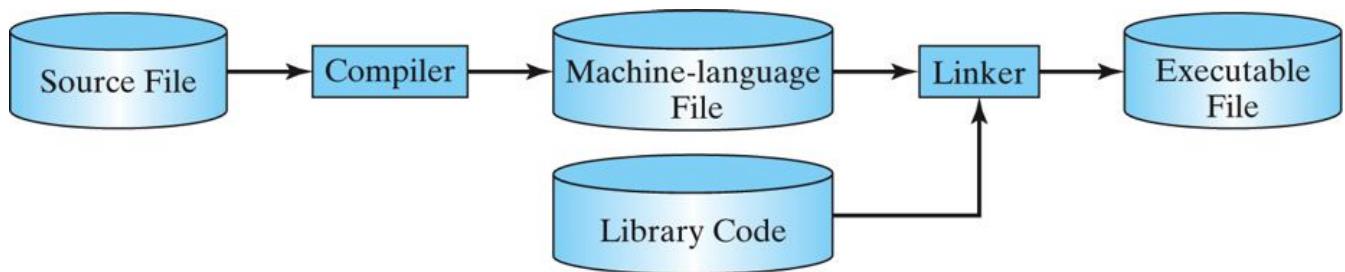
developing graphical user interfaces and in rapid application development. C++ is popular for system software projects such as writing compilers and operating systems. The Microsoft Windows operating system was coded using C++. C# (pronounced C sharp) is a new language developed by Microsoft for developing applications based on the Microsoft .NET platform. Java, developed by Sun Microsystems, is widely used for developing platform-independent Internet applications.

6

A program written in a high-level language is called a *source program* or *source code*. Since a computer cannot understand a source program, a program called a *compiler* is used to translate it into a machine-language program. The machine-language program is then linked with other supporting library code to form an executable file, which can be run on the machine, as shown in [Figure 1.4](#). On Windows, executable files have extension.exe.

source program
compiler

FIGURE 1.4 A source program is compiled into a machine-language file, which is then linked with the system library to form an executable file.



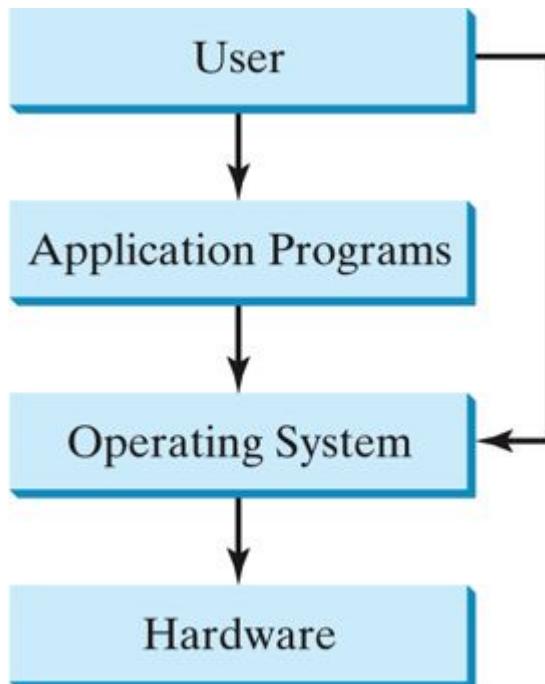
1.4 Operating Systems

The *operating system (OS)* is the most important program that runs on a computer, which manages and controls a computer's activities. The popular operating systems are Microsoft Windows, Mac OS, and Linux. Application programs, such as a Web browser or a word processor, cannot run without an operating system. The interrelationship of

hardware, operating system, application software, and the user is shown in [Figure 1.5](#).

OS

FIGURE 1.5 The operating system is the software that controls and manages the system.



The major tasks of an operating system are:

- Controlling and monitoring system activities
- Allocating and assigning system resources
- Scheduling operations

1.4.1 Controlling and Monitoring System Activities

Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the monitor, keeping track of files and directories on the disk, and controlling peripheral devices, such as disk drives and printers. They also make sure that different programs and users running at the same time do not interfere with each other, and they are responsible for security, ensuring that unauthorized users do not access the system.

7

8

1.4.2 Allocating and Assigning System Resources

The operating system is responsible for determining what computer resources a program needs (e.g., CPU, memory, disks, input and output devices) and for allocating and assigning them to run the program.

1.4.3 Scheduling Operations

The OS is responsible for scheduling programs to make efficient use of system resources. Many of today's operating systems support such techniques as *multiprogramming*, *multithreading*, or *multiprocessing* to increase system performance.

Multiprogramming allows multiple programs to run simultaneously by sharing the CPU. The CPU is much faster than the computer's other components. As a result, it is idle most of the time—for example, while waiting for data to be transferred from the disk or from other sources. A multiprogramming OS takes advantage of this situation by allowing multiple programs to use the CPU when it would otherwise

be idle. For example, you may use a word processor to edit a file at the same time as the Web browser is downloading a file.

multiprogramming

Multithreading allows concurrency within a program, so that its subtasks can run at the same time. For example, a word-processing program allows users to simultaneously edit text and save it to a file. In this example, editing and saving are two tasks within the same application. These two tasks may run on separate threads concurrently.

multithreading

Multiprocessing, or parallel processing, uses two or more processors together to perform a task. It is like a surgical operation where several doctors work together on one patient.

multiprocessing

1.5 Java, World Wide Web, and Beyond

This book introduces Java programming. Java was developed by a team led by James Gosling at Sun Microsystems. Originally called *Oak*, it was designed in 1991 for use in embedded chips in consumer electronic appliances. In 1995, renamed *Java*, it was redesigned for developing Internet applications. For the history of Java, see java.sun.com/features/1998/05/birthday.html.

Java has become enormously popular. Its rapid rise and wide acceptance can be traced to its design characteristics, particularly its promise that you can write a program once and run it anywhere. As stated by Sun,

Java is *simple, object oriented, distributed, interpreted, robust, secure, architecture neutral, portable, high performance, multithreaded, and dynamic*. For the anatomy of Java characteristics, see www.cs.armstrong.edu/liang/JavaCharacteristics.pdf.

Java is a full-featured, general-purpose programming language that can be used to develop robust mission-critical applications. Today, it is employed not only for Web programming, but also for developing standalone applications across platforms on servers, desktops, and mobile devices. It was used to develop the code to communicate with and control the robotic rover on Mars. Many companies that once considered Java to be more hype than substance are now using it to create distributed applications accessed by customers and partners across the Internet. For every new project being developed today, companies are asking how they can use Java to make their work easier.

The World Wide Web is an electronic information repository that can be accessed on the Internet from anywhere in the world. The Internet, the Web's infrastructure, has been around for more than thirty years. The colorful World Wide Web and sophisticated Web browsers are the major reason for the Internet's popularity.

The primary authoring language for the Web is the Hypertext Markup Language (HTML). HTML is a simple language for laying out documents, linking documents on the Internet, and bringing images, sound, and video alive on the Web. However, it cannot interact with the user except through simple forms. Web pages in HTML are essentially static and flat.

applet

Java initially became attractive because Java programs can be run from a Web browser. Such programs are called *applets*. Applets employ a modern graphical interface with buttons, text fields, text areas, radio

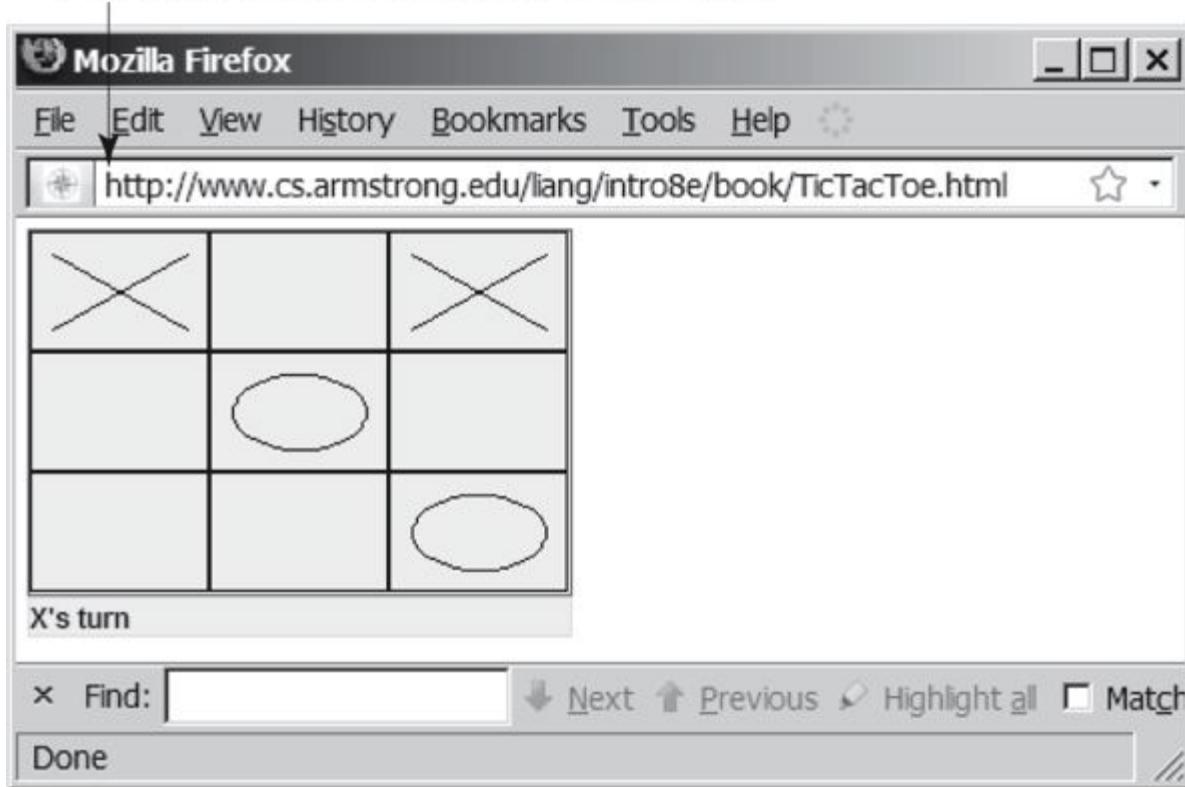
8

9

modern graphical interface with buttons, text fields, text areas, radio buttons, and so on, to interact with users on the Web and process their requests. Applets make the Web responsive, interactive, and fun to use. [Figure 1.6](#) shows an applet running from a Web browser for playing a Tic Tac Toe game.

FIGURE 1.6 A Java applet for playing TicTacToe is embedded in an HTML page.

Enter this URL from a Web browser

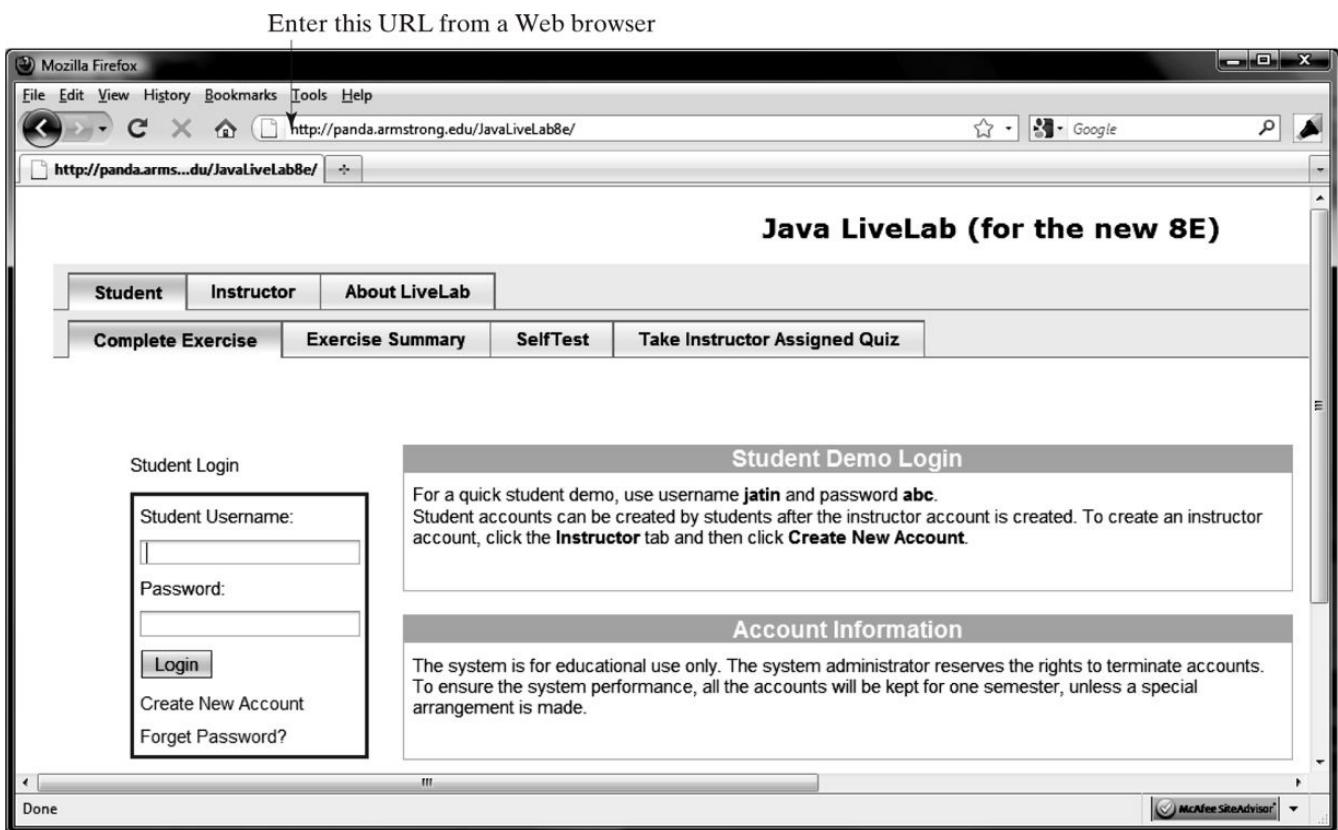


Tip

For a demonstration of Java applets, visit java.sun.com/applets. This site provides a rich Java resource as well as links to other cool applet demo sites. java.sun.com is the official Sun Java Website.

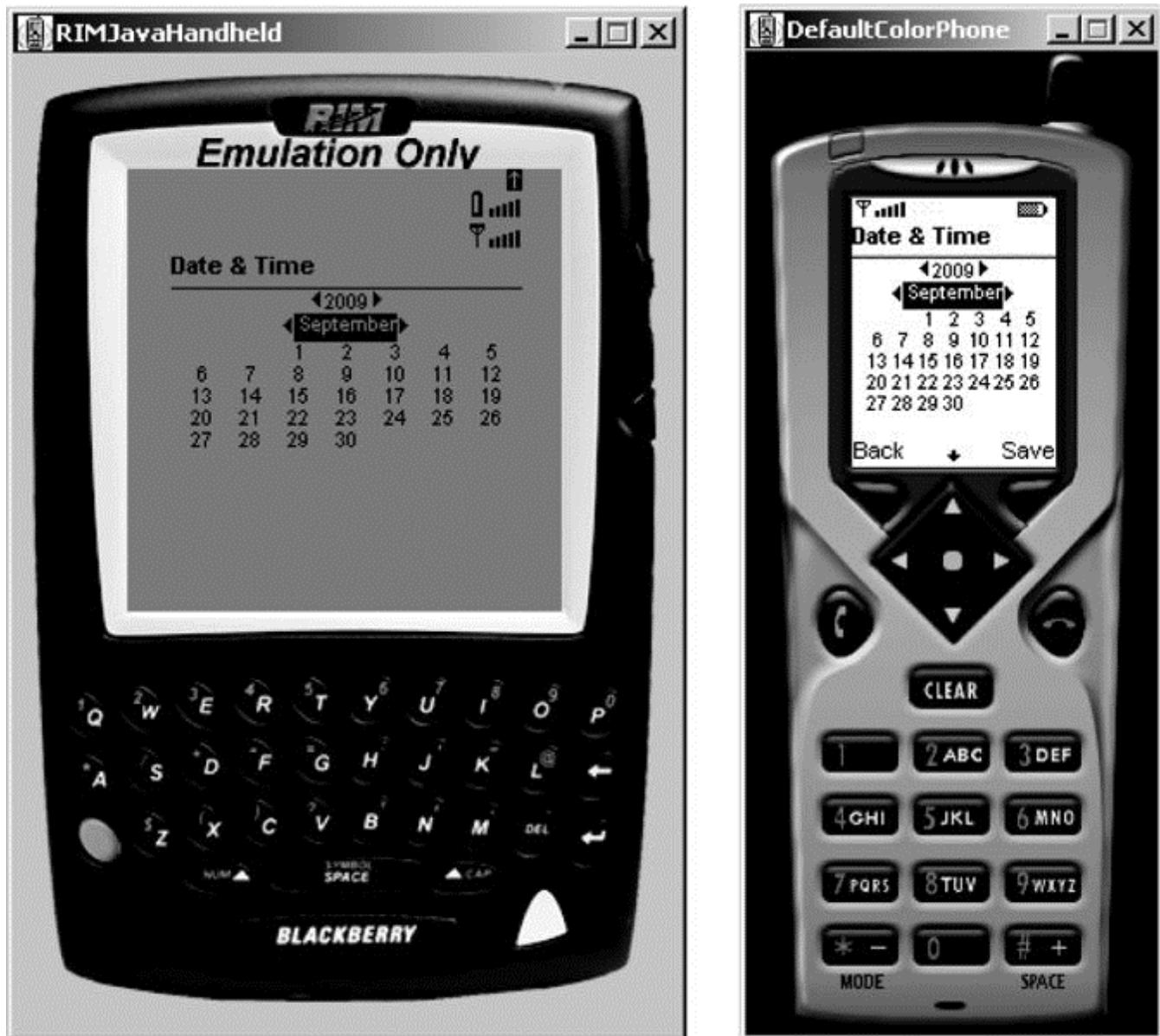
Java can also be used to develop applications on the server side. These applications can be run from a Web server to generate dynamic Web pages. The automatic grading system for this book, as shown in [Figure 1.7](#), was developed using Java.

FIGURE 1.7 Java was used to develop an automatic grading system to accompany this book.



Java is a versatile programming language. You can use it to develop applications on your desktop and on the server. You can also use it to develop applications for small hand-held devices. [Figure 1.8](#) shows a Java-programmed calendar displayed on a BlackBerry® and on a cell phone.

FIGURE 1.8 Java can be used to develop applications for hand-held and wireless devices, such as a BlackBerry® (left) and a cell phone (right).



1.6 The Java Language Specification, API, JDK, and IDE

Computer languages have strict rules of usage. If you do not follow the rules when writing a program, the computer will be unable to understand it. The Java language specification and Java API define the Java standard.

The *Java language specification* is a technical definition of the language that includes the syntax and semantics of the Java programming language. The complete Java language specification can be found at java.sun.com/docs/books/jls.

Java language specification

The *application program interface (API)* contains predefined classes and interfaces for developing Java programs. The Java language specification is stable, but the API is still expanding. At the Sun Java Website (java.sun.com), you can view and download the latest version of the Java API.

API

Java is a full-fledged and powerful language that can be used in many ways. It comes in three editions: *Java Standard Edition (Java SE)*, *Java Enterprise Edition (Java EE)*, and *Java Micro Edition (Java ME)*. Java SE can be used to develop client-side standalone applications or applets. Java EE can be used to develop server-side applications, such as Java servlets and JavaServer Pages. Java ME can be used to develop

applications for mobile devices, such as cell phones. This book uses Java SE to introduce Java programming.

Java SE, EE, and ME

There are many versions of Java SE. The latest, Java SE 6, will be used in this book. Sun releases each version with a *Java Development Toolkit (JDK)*. For Java SE 6, the Java Development Toolkit is called *JDK 1.6* (also known as *Java 6 or JDK 6*).

JDK 1.6 = JDK 6

JDK consists of a set of separate programs, each invoked from a command line, for developing and testing Java programs. Besides JDK, you can use a Java development tool (e.g., Net-Beans, Eclipse, and TextPad)—software that provides an *integrated development environment (IDE)* for rapidly developing Java programs. Editing, compiling, building, debugging, and online help are integrated in one

compiling, building, debugging, and online help are integrated in one graphical user interface. Just enter source code in one window or open an existing file in a window, then click a button, menu item, or function key to compile and run the program.

Java IDE

1.7 A Simple Java Program

Let us begin with a simple Java program that displays the message “Welcome to Java!” on the console. *Console* refers to text entry and display device of a computer. The program is shown in [Listing 1.1](#).

console



Video Note

First Java program

LISTING 1.1 Welcome.java

```
1 public class Welcome {  
2     public static void main(String[] args) {  
3         // Display message Welcome to Java! to the console  
4         System.out.println("Welcome to Java!");  
5     }  
6 }
```

class
main method
display message



Welcome to Java!

The *line numbers* are displayed for reference purposes but are not part of the program. So, don't type line numbers in your program.

line numbers

Line 1 defines a class. Every Java program must have at least one class. Each class has a name. By convention, class names start with an uppercase letter. In this example, the *class name* is **Welcome**.

class name

Line 2 defines the *main method*. In order to run a class, the class must contain a method named **main**. The program is executed from the **main** method.

main method

A method is a construct that contains statements. The **main** method in this program contains the **System.out.println** statement. This statement prints a message "**Welcome to Java!**" to the console (line 4). Every statement in Java ends with a semicolon (;), known as the *statement terminator*.

statement terminator

Reserved words, or *keywords*, have a specific meaning to the compiler and cannot be used for other purposes in the program. For example, when the compiler sees the word **class**, it understands that the word after **class** is the name for the class. Other reserved words in this program are **public**, **static**, and **void**.

reserved word

Line 3 is a *comment* that documents what the program is and how it is constructed. Comments help programmers to communicate and understand the program. They are not programming statements and thus are ignored by the compiler. In Java, comments are preceded by two slashes (//) on a line, called a *line comment*, or enclosed between /* and */ on one or several lines, called a *block comment*. When the compiler sees //, it ignores all text after // on the same line. When it sees /*, it scans for the next */ and ignores any text between /* and */. Here are examples of comments:

```
// This application program prints Welcome to  
Java!  
/* This application program prints Welcome  
to Java! */  
/* This application program  
prints Welcome to Java! */  
prints Welcome to Java! */
```

comment

block

A pair of braces in a program forms a *block* that groups the program's components. In Java, each block begins with an opening brace ({) and ends with a closing brace (}). Every class has a *class block* that groups the data and methods of the class. Every method has a *method block* that groups the statements in the method. Blocks can be *nested*, meaning that one block can be placed within another, as shown in the following code.

```
public class Welcome { ←  
    public static void main(String[] args) { ← Class block  
        System.out.println("Welcome to Java!"); Method block  
    } ←  
}
```

Tip

An opening brace must be matched by a closing brace. Whenever you type an opening brace, immediately type a closing brace to prevent the missing-brace error. Most Java IDEs automatically insert the closing brace for each opening brace.

matching braces

Note

You are probably wondering why the `main` method is declared this way and why `System.out.println(...)` is used to display a message to the console. For the time being, simply accept that this is how things are done. Your questions will be fully answered in subsequent chapters.

Caution

Java source programs are case sensitive. It would be wrong, for example, to replace `main` in the program with `Main`.

case sensitive

Note

Like any programming language, Java has its own syntax, and you need to write code that obeys the *syntax rules*. If your program violates the rules—for example if the semicolon is missing, a brace is missing, a quotation mark is missing, or **String** is misspelled—the Java compiler will report syntax errors. Try to compile the program with these errors and see what the compiler reports.

syntax rules

The program in [Listing 1.1](#) displays one message. Once you understand the program, it is easy to extend it to display more messages. For example, you can rewrite the program to display three messages, as shown in [Listing 1.2](#).

LISTING 1.2 Welcome1.java

| | |
|---|---|
| class
main method
display message | 1 public class Welcome1 {
2 public static void main(String[] args) {
3 System.out.println("Programming is fun!");
4 System.out.println("Fundamentals First");
5 System.out.println("Problem Driven");
6 }
7 } |
|---|---|



Programming is fun!
Fundamentals First
Problem Driven

Further, you can perform mathematical computations and display the result to the console. [Listing 1.3](#) gives an example of evaluating

$$\frac{10.5 + 2 \times 3}{45 - 3.5}$$

LISTING 1.3 ComputeExpression.java

```
1 public class ComputeExpression {  
2     public static void main(String[] args) {  
3         System.out.println((10.5 + 2 * 3) / (45 - 3.5));  
4     }  
5 }
```

class
main method
compute expression



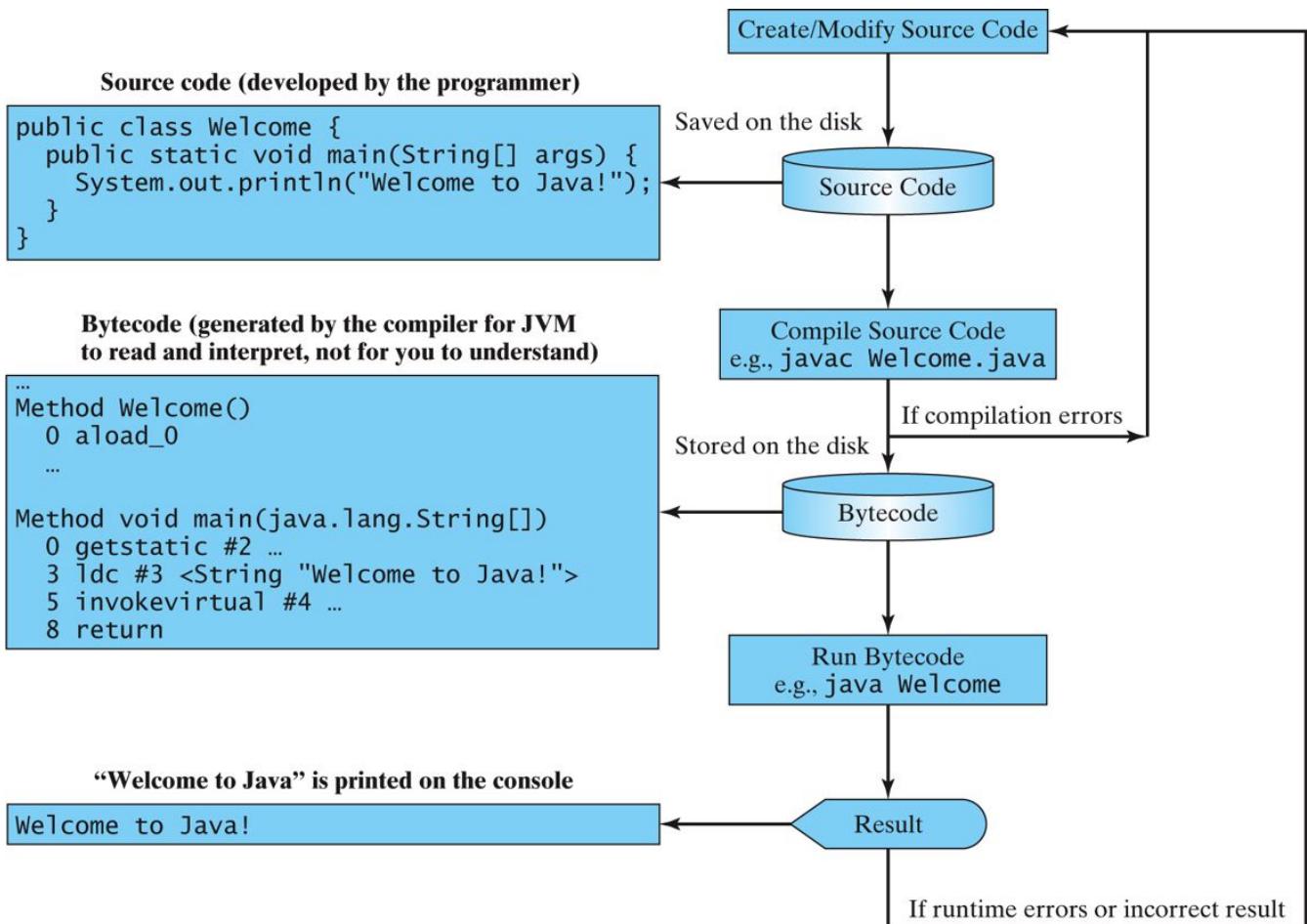
0 . 39759036144578314

The multiplication operator in Java is `*`. As you see, it is a straightforward process to translate an arithmetic expression to a Java expression. We will discuss Java expressions further in [Chapter 2](#).

1.8 Creating, Compiling, and Executing a Java Program

You have to create your program and compile it before it can be executed. This process is repetitive, as shown in [Figure 1.9](#). If your program has compilation errors, you have to modify the program to fix them, then recompile it. If your program has runtime errors or does not produce the correct result, you have to modify the program, recompile it, and execute it again.

FIGURE 1.9 The Java program-development process consists of repeatedly creating/modifying source code, compiling, and executing programs.



You can use any text *editor* or IDE to create and edit a Java source-code file. This section demonstrates how to create, compile, and run Java programs from a command window. If you wish to use an IDE such as Eclipse, NetBeans, or TextPad, please refer to Supplement II for tutorials. From the command window, you can use the NotePad to create the Java source code file, as shown in [Figure 1.10](#).

editor



Video Note

Brief Eclipse Tutorial

FIGURE 1.10 You can create the Java source file using Windows Notepad.



Note

The source file must end with the extension .java and must have exactly the same name as the public class name. For example, the file for the source code in [Listing 1.1](#) should be named Welcome.java, since the public class name is **Welcome**.

file name

A Java compiler translates a Java source file into a Java bytecode file. The following command compiles Welcome.java:

compile

`javac Welcome.java`

Note

You must first install and configure JDK before compiling and running programs. See Supplement I.B, “Installing and Configuring JDK 6,” on how to install JDK and set up the environment to compile and run Java programs. If you have trouble compiling and running programs, please see Supplement I.C, “Compiling and Running Java from the Command Window.” This supplement also explains how to use basic DOS commands and how to use Windows NotePad and WordPad to create and edit files. All the supplements are accessible from the Companion Website.

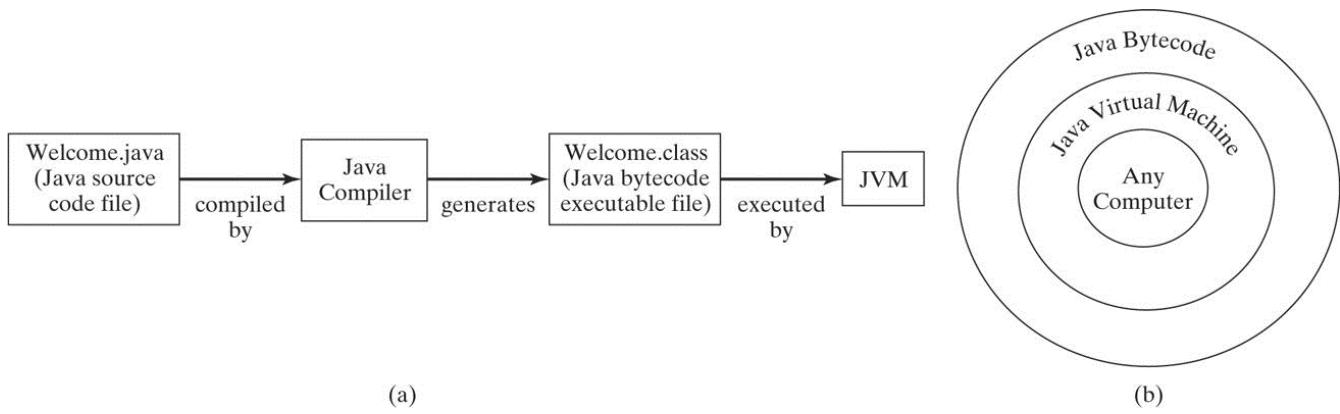
Supplement I.B

Supplement I.C

If there are no syntax errors, the *compiler* generates a bytecode file with a .class extension. So the preceding command generates a file named **Welcome.class**, as shown in [Figure 1.11\(a\)](#). The Java language is a high-level language while Java bytecode is a low-level language. The bytecode is similar to machine instructions but is architecture neutral and can run on any platform that has a Java Virtual Machine (JVM), as shown in [Figure 1.11\(b\)](#). Rather than a physical machine, the virtual machine is a program that interprets Java bytecode. This is one of Java’s primary advantages: *Java bytecode can run on a variety of hardware platforms and operating systems.*

.class bytecode file

FIGURE 1.11 (a) Java source code is translated into bytecode. (b) Java bytecode can be executed on any computer with a Java Virtual Machine.



To execute a Java program is to run the program's bytecode. You can execute the bytecode on any platform with a JVM. Java bytecode is interpreted. Interpreting translates the individual steps in the bytecode into the target machine-language code one at a time rather than translating the whole program as a single unit. Each step is executed immediately after it is translated.

interpreting bytecode

The following command runs the bytecode:

run

java Welcome

[Figure 1.12](#) shows the **javac** command for compiling `Welcome.java`. The compiler generated the `Welcome.class` file. This file is executed using the **java** command.



Video Note

Compile and run a Java program

FIGURE 1.12 The output of Listing 1.1 displays the message “Welcome to Java!”

The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window displays the following text:

```
C:\book>javac Welcome.java
C:\book>java Welcome
Welcome to Java!
```

Annotations with arrows point to specific parts of the window:

- An arrow labeled "Compile" points to the first line of text: "C:\book>javac Welcome".
- An arrow labeled "Show files" points to the second line of text: "C:\book>java Welcome".
- An arrow labeled "Run" points to the third line of text: "Welcome to Java!".

Note

For simplicity and consistency, all source code and class files are placed under **c:\book** unless specified otherwise.

c:\book

Caution

Do not use the extension .class in the command line when executing the program. Use **java ClassName** to run the program. If you use **java ClassName.class** in the command line, the system will attempt to fetch **ClassName.class.class**.

java ClassName

Tip

If you execute a class file that does not exist, a **NoClassDefFoundError** will occur. If you execute a class file that does not have a **main** method or you mistype the **main** method (e.g., by typing **Main** instead of **main**), a **NoSuchMethodError** will occur.

NoClassDefFoundError

NoSuchMethodError

Note

When executing a Java program, the JVM first loads the bytecode of the class to memory using a program called the *class loader*. If your program uses other classes, the class loader dynamically loads them just before they are needed. After a class is loaded, the JVM uses a program called *bytecode verifier* to check the validity of the bytecode and ensure that the bytecode does not violate Java's security restrictions. Java enforces strict security to make sure that Java programs arriving from the network do not harm your computer.

class loader

bytecode verifier

15

16

Pedagogical Note

Instructors may require students to use packages for organizing programs. For example, you may place all programs in this chapter in a package named [chapter 1](#). For instructions on how to use packages, please see Supplement I.F, “Using Packages to Organize the Classes in the Text.”

using package

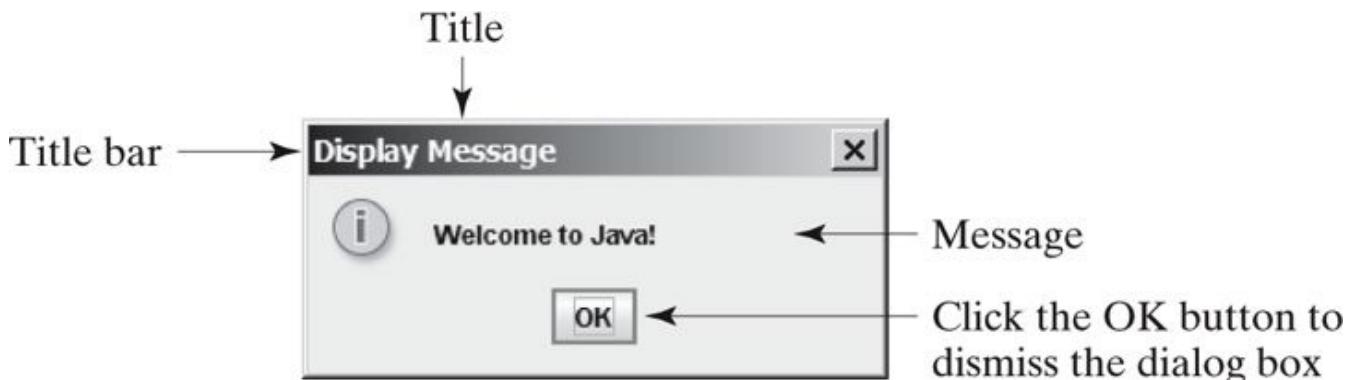
1.9 (GUI) Displaying Text in a Message Dialog Box

The program in [Listing 1.1](#) displays the text on the console, as shown in [Figure 1.12](#). You can rewrite the program to display the text in a

message dialog box. To do so, you need to use the `showMessageDialog` method in the `JOptionPane` class. `JOptionPane` is one of the many predefined classes in the Java system that you can reuse rather than “reinventing the wheel.” You can use the `showMessageDialog` method to display any text in a message dialog box, as shown in [Figure 1.13](#). The new program is given in [Listing 1.4](#).

JOptionPane
showMessageDialog

FIGURE 1.13 “Welcome to Java!” is displayed in a message box.



LISTING 1.4 WelcomeInMessageDialogBox.java

| | |
|-----------------|--|
| block comment | 1 /* This application program displays Welcome to Java! |
| | 2 * in a message dialog box. |
| | 3 */ |
| import | 4 import javax.swing.JOptionPane; |
| | 5 |
| main method | 6 public class WelcomeInMessageDialogBox { |
| | 7 public static void main(String[] args) { |
| display message | 8 // Display Welcome to Java! in a message dialog box |
| | 9 JOptionPane.showMessageDialog(null, "Welcome to Java!"); |
| | 10 } |
| | 11 } |

This program uses a Java class **JOptionPane** (line 9). Java's predefined classes are grouped into packages. **JOptionPane** is in the **javax.swing** package. **JOptionPane** is imported to the program using the **import** statement in line 4 so that the compiler can locate the class without the full name **javax.swing.JOptionPane**.

package

Note

If you replace **JOptionPane** on line 9 with **javax.swing.JOptionPane**, you don't need to import it in line 4. **javax.swing.JOptionPane** is the full name for the **JOptionPane** class.

The **showMessageDialog** method is a *static* method. Such a method should be invoked by using the class name followed by a dot operator (.) and the method name with arguments. Methods will be introduced in [Chapter 5](#), "Methods." The **showMessageDialog** method can be invoked with two arguments, as shown below.



```
JOptionPane.showMessageDialog(null,  
    "Welcome to Java!");
```

The first argument can always be **null**. **null** is a Java keyword that will be fully introduced in [Chapter 8](#), “Objects and Classes.” The second argument is a string for text to be displayed.

There are several ways to use the **showMessageDialog** method. For the time being, you need to know only two ways. One is to use a statement, as shown in the example:

two versions of **showMessageDialog**

```
JOptionPane.showMessageDialog(null, x);
```

where **x** is a string for the text to be displayed.

The other is to use a statement like this one:

```
JOptionPane.showMessageDialog(null, x,  
y, JOptionPane.INFORMATION_MESSAGE);
```

where **x** is a string for the text to be displayed, and **y** is a string for the title of the message box. The fourth argument can be

JOptionPane.INFORMATION_MESSAGE, which causes the icon (ⓘ) to be displayed in the message box, as shown in the following example.



```
JOptionPane.showMessageDialog(null,  
"Welcome to Java!",  
"Display Message",  
JOptionPane.INFORMATION_MESSAGE);
```

Note

There are two types of **import** statements: *specific import* and *wildcard import*. The *specific import* specifies a single class in the import statement. For example, the following statement imports **JOptionPane** from package **javax.swing**.

```
import javax.swing.JOptionPane;
```

specific import

The *wildcard import* imports all the classes in a package. For example, the following statement imports all classes from package **javax.swing**.

```
import javax.swing.*;
```

wildcard import

The information for the classes in an imported package is not read in at compile time or runtime unless the class is used in the program. The import statement simply tells the compiler where to locate the classes. There is no performance difference between a specific import and a wildcard import declaration.

no performance difference

Note

Recall that you have used the **System** class in the statement **System.out.println("Welcome to Java");** in [Listing](#)

1.1. The **System** class is not imported because it is in the **java.lang** package. All the classes in the **java.lang** package are *implicitly* imported in every Java program.

java.lang

implicitly imported

KEY TERMS

.class file [14](#)

.java file [14](#)

assembly language [5](#)

bit [3](#)

block [12](#)

block comment [11](#)

bus [2](#)

byte [3](#)

bytecode [14](#)

bytecode verifier [15](#)

cable modem [5](#)

central processing unit (CPU) [2](#)

class loader [15](#) comment [11](#)

compiler [7](#)

console [11](#)

dot pitch [5](#)

DSL (digital subscriber line) [5](#)

hardware [2](#)

high-level language [6](#)

Integrated Development Environment (IDE) [10](#)

java command [15](#)

javac command [15](#)

Java Development Toolkit (JDK) [10](#)

Java Virtual Machine (JVM) [14](#)

keyword (or reserved word) [11](#)

line comment [11](#)

machine language [5](#)

main method [11](#)

memory [3](#)

modem [5](#)

network interface card (NIC) [5](#)

operating system (OS) [7](#)

pixel [5](#)

program [5](#)

programming [5](#)

resolution [5](#)

software [5](#)

source code [7](#)

source file [14](#)

specific import [17](#)

storage devices [4](#)

statement [11](#)

wildcard import [17](#)



Note

The above terms are defined in the present chapter. Supplement I.A, "Glossary," lists all the key terms and descriptions in the book, organized by chapters.

Supplement I.A

CHAPTER SUMMARY

1. A computer is an electronic device that stores and processes data.
2. A computer includes both *hardware* and *software*.
3. Hardware is the physical aspect of the computer that can be seen.
4. Computer *programs*, known as *software*, are the invisible instructions that control the hardware and make it perform tasks.

5. Computer programming is the writing of instructions (i.e., code) for computers to perform.
6. The central processing unit (CPU) is a computer's brain. It retrieves instructions from memory and executes them.
7. Computers use zeros and ones because digital devices have two stable states, referred to by convention as zero and one.
8. A bit is a binary digit 0 or 1.
9. A byte is a sequence of 8 bits.
10. A kilobyte is about 1000 bytes, a megabyte about 1 million bytes, a gigabyte about 1 billion bytes, and a terabyte about 1000 gigabytes.
11. Memory stores data and program instructions for the CPU to execute.
12. A memory unit is an ordered sequence of bytes.
13. Memory is volatile, because information is lost when the power is turned off.

14. Programs and data are permanently stored on storage devices and are moved to memory when the computer actually uses them.
15. The machine language is a set of primitive instructions built into every computer.
16. Assembly language is a low-level programming language in which a mnemonic is used to represent each machine-language instruction.
17. High-level languages are English-like and easy to learn and program.
18. A program written in a high-level language is called a source program.
19. A compiler is a software program that translates the source program into a machine-language program.
20. The operating system (OS) is a program that manages and controls a computer's activities.
21. Java is platform independent, meaning that you can write a program once and run it anywhere.
22. Java programs can be embedded in HTML pages and downloaded by Web browsers to bring live animation and interaction to Web clients.
23. Java source files end with the .java extension.
24. Every class is compiled into a separate bytecode file that has the same name as the class and ends with the .class extension.

25. To compile a Java source-code file from the command line, use the **javac** command.
26. To run a Java class from the command line, use the **java** command.
27. Every Java program is a set of class definitions. The keyword **class** introduces a class definition. The contents of the class are included in a block.
28. A block begins with an opening brace {} and ends with a closing brace {}. Methods are contained in a class.
29. A Java program must have a **main** method. The **main** method is the entry point where the program starts when it is executed.
30. Every statement in Java ends with a semicolon ;, known as the *statement terminator*.
31. *Reserved words*, or *keywords*, have a specific meaning to the compiler and cannot be used for other purposes in the program.
32. In Java, comments are preceded by two slashes // on a line, called a *line comment*, or enclosed between /* and */ on one or several lines, called a *block comment*.
33. Java source programs are case sensitive.
34. There are two types of **import** statements: *specific import* and *wildcard import*. The *specific import* specifies a single class in the import statement. The *wildcard import* imports all the classes in a package.

REVIEW QUESTIONS



Note

Answers to review questions are on the Companion Website.

Sections 1.2–1.4

- 1.1** Define hardware and software.
- 1.2** List the main components of the computer.
- 1.3** Define machine language, assembly language, and high-level programming language.
- 1.4** What is a source program? What is a compiler?
- 1.5** What is the JVM?
- 1.6** What is an operating system?

Sections 1.5–1.6

- 1.7** Describe the history of Java. Can Java run on any machine? What is needed to run Java on a computer?
- 1.8** What are the input and output of a Java compiler?
- 1.9** List some Java development tools. Are tools like NetBeans and Eclipse different languages from Java, or are they dialects or extensions of Java?
- 1.10** What is the relationship between Java and HTML?

Sections 1.7–1.9

- 1.11** Explain the Java keywords. List some Java keywords you learned in this chapter.
- 1.12** Is Java case sensitive? What is the case for Java keywords?
- 1.13** What is the Java source filename extension, and what is the Java bytecode filename extension?
- 1.14** What is a comment? Is the comment ignored by the compiler? How do you denote a comment line and a comment paragraph?
- 1.15** What is the statement to display a string on the console? What is the statement to display the message “Hello world” in a message dialog box?
- 1.16** The following program is wrong. Reorder the lines so that the program displays **morning** followed by **afternoon**.

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

public class Welcome {
    System.out.println("afternoon");
    System.out.println("morning");
}
```

- 1.17** Identify and fix the errors in the following code:

```
1 public class Welcome {
2     public void Main(String[ ] args) {
3         System.out.println(&Welcome to
Java!);
```

4 }
5)

20

1.18 What is the command to compile a Java program? What is the command to run a Java program?

1.19 If a **NoClassDefFoundError** occurs when you run a program, what is the cause of the error?

1.20 If a **NoSuchMethodError** occurs when you run a program, what is the cause of the error?

1.21 Why does the **System** class not need to be imported?

1.22 Are there any performance differences between the following two **import** statements?

```
import javax.swing.JOptionPane;  
import javax.swing.*;
```

1.23 Show the output of the following code:

```
public class Test {  
    public static void main(String[]  
args) {  
        System.out.println("3.5 * 4 / 2 -  
2.5 is ");  
        System.out.println(3.5 * 4 / 2 -  
2.5);  
    }  
}
```

PROGRAMMING EXERCISES



Note

Solutions to even-numbered exercises are on the Companion Website. Solutions to all exercises are on the Instructor Resource

Website. The level of difficulty is rated easy (no star), moderate (*), hard (**), or challenging (***)�

level of difficulty

1.1 (*Displaying three messages*) Write a program that displays

**Welcome to Java, Welcome to Computer
Science, and Programming is fun.**

1.2 (*Displaying five messages*) Write a program that displays

Welcome to Java five times.

1.3* (*Displaying a pattern*) Write a program that displays the following pattern:

| | | | | |
|---|-----|-------|-----|-------|
| J | A | V | V | A |
| J | A A | V | V | A A |
| J | J | AAAAA | V V | AAAAA |
| J | J | A | V | A |
| | | A | | A |

1.4 (*Printing a table*) Write a program that displays the following table:

| | | |
|---|-------|-------|
| a | a^2 | a^3 |
| 1 | 1 | 1 |
| 2 | 4 | 8 |
| 3 | 9 | 27 |
| 4 | 16 | 64 |

1.5 (*Computing expressions*) Write a program that displays the result of

$$\frac{9.5 \times 4.5 - 2.5 \times 3}{45.5 - 3.5}.$$

1.6 (*Summation of a series*) Write a program that displays the result of $1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9$.

1.7 (*Approximating π*) π can be computed using the following formula:

$$\pi = 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} + \dots \right)$$

Write a program that displays the result of

$4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} \right)$. Use **1.0** instead of **1** in your program.

LISTING 2.1 ComputeArea.java

```
1 public class ComputeArea {  
2     public static void main(String[] args) {  
3         double radius; // Declare radius  
4         double area; // Declare area  
5  
6         // Assign a radius  
7         radius = 20; // New value is radius  
8  
9         // Compute area  
10        area = radius * radius * 3.14159;  
11  
12        // Display results  
13        System.out.println("The area for the circle of radius " +  
14            radius + " is " + area);  
15    }  
16 }
```

Variables such as `radius` and `area` correspond to memory locations. Every variable has a name, a type, a size, and a value. Line 3 declares that `radius` can store a `double` value. The value is not defined until you assign a value. Line 7 assigns `20` into `radius`. Similarly, line 4 declares variable `area`, and line 10 assigns a value into `area`. The following table shows the value in the memory for `area` and `radius` as the program is executed. Each row in the table shows the values of variables after the statement in the corresponding line in the program is executed. Hand trace is helpful to understand how a program works, and it is also a useful tool for finding errors in the program.

declaring variable

assign value



line #

radius

area

no value

4

no value

7

20

10

1256.636

The plus sign (+) has two meanings: one for addition and the other for concatenating strings. The plus sign (+) in lines 13–14 is called a *string concatenation operator*. It combines two strings if two operands are strings. If one of the operands is a nonstring (e.g., a number), the nonstring value is converted into a string and concatenated with the other string. So the plus signs (+) in lines 13–14 concatenate strings into a longer string, which is then displayed in the output. Strings and string concatenation will be discussed further in §2.15, “The **String** Type.”

concatenating strings

concatenating strings with numbers



Caution

A string constant cannot cross lines in the source code. Thus the following statement would result in a compile error:

```
System.out.println("Introduction to Java Programming,  
by Y. Daniel Liang");
```

To fix the error, break the string into separate substrings, and use the concatenation operator (+) to combine them:

```
System.out.println("Introduction to Java Programming, " +  
    "by Y. Daniel Liang");
```

breaking a long string



Tip

This example consists of three steps. It is a good approach to develop and test these steps incrementally by adding them one at a time.

incremental development and testing

2.3 Reading Input from the Console

In [Listing 2.1](#), the radius is fixed in the source code. To use a different radius, you have to modify the source code and recompile it. Obviously, this is not convenient. You can use the `Scanner` class for console input.



Video Note

Obtain input

Java uses `System.out` to refer to the standard output device and `System.in` to the standard input device. By default the output device is the display monitor, and the input device is the keyboard. To perform console output, you simply use the `println` method to display a primitive value or a string to the console. Console input is not directly supported in Java, but you can use the `Scanner` class to create an object to read input from `System.in`, as follows:

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

The syntax `new Scanner(System.in)` creates an object of the `Scanner` type. The syntax `Scanner input` declares that `input` is a variable whose type is `Scanner`. The whole line `Scanner input = new Scanner(System.in)` creates a `Scanner` object and assigns its reference to the variable `input`. An object may invoke its methods. To

invoke a method on an object is to ask the object to perform a task. You can invoke the methods in [Table 2.1](#) to read various types of input.

TABLE 2.1 Methods for Scanner Objects

| <i>Method</i> | <i>Description</i> |
|---------------------------|---|
| <code>nextByte()</code> | reads an integer of the <code>byte</code> type. |
| <code>nextShort()</code> | reads an integer of the <code>short</code> type. |
| <code>nextInt()</code> | reads an integer of the <code>int</code> type. |
| <code>nextLong()</code> | reads an integer of the <code>long</code> type. |
| <code>nextFloat()</code> | reads a number of the <code>float</code> type. |
| <code>nextDouble()</code> | reads a number of the <code>double</code> type. |
| <code>next()</code> | reads a string that ends before a whitespace character. |
| <code>nextLine()</code> | reads a line of text (i.e., a string ending with the <i>Enter</i> key pressed). |

For now, we will see how to read a number that includes a decimal point by invoking the `nextDouble()` method. Other methods will be covered when they are used. [Listing 2.2](#) rewrites [Listing 2.1](#) to prompt the user to enter a radius.

LISTING 2.2 ComputeAreaWithConsoleInput.java

```
1 import java.util.Scanner; // Scanner is in the java.util package      import class
2
3 public class ComputeAreaWithConsoleInput {
4     public static void main(String[] args) {
5         // Create a Scanner object
6         Scanner input = new Scanner(System.in);                          create a Scanner
7
8         // Prompt the user to enter a radius
9         System.out.print("Enter a number for radius: ");
10        double radius = input.nextDouble();                                read a double
11
12        // Compute area
13        double area = radius * radius * 3.14159;
14
15        // Display result
16        System.out.println("The area for the circle of radius " +
17            radius + " is " + area);
18    }
19 }
```



2.5 ↵ Enter

A screenshot of a terminal window. It shows the text "2.5" in a blue input field and a grey "Enter" button with a left arrow symbol to its left.

Enter a number for radius:

The area for the circle of radius 2.5 is 19.6349375



23 ↵ Enter

A screenshot of a terminal window. It shows the text "23" in a blue input field and a grey "Enter" button with a left arrow symbol to its left.

Enter a number for radius:

The area for the circle of radius 23.0 is 1661.90111

The **Scanner** class is in the **java.util** package. It is imported in line 1. Line 6 creates a **Scanner** object.

The statement in line 9 displays a message to prompt the user for input.

System.out.print ("Enter a number for radius: ");

The `print` method is identical to the `println` method except that `println` moves the cursor to the next line after displaying the string, but `print` does not advance the cursor to the next line when completed.

print vs. println

The statement in line 10 reads an input from the keyboard.

```
double radius = input.nextDouble();
```

After the user enters a number and presses the *Enter* key, the number is read and assigned to `radius`.

More details on objects will be introduced in [Chapter 8](#), “Objects and Classes.” For the time being, simply accept that this is how to obtain input from the console.

[Listing 2.3](#) gives another example of reading input from the keyboard. The example reads three numbers and displays their average.

LISTING 2.3 ComputeAverage.java

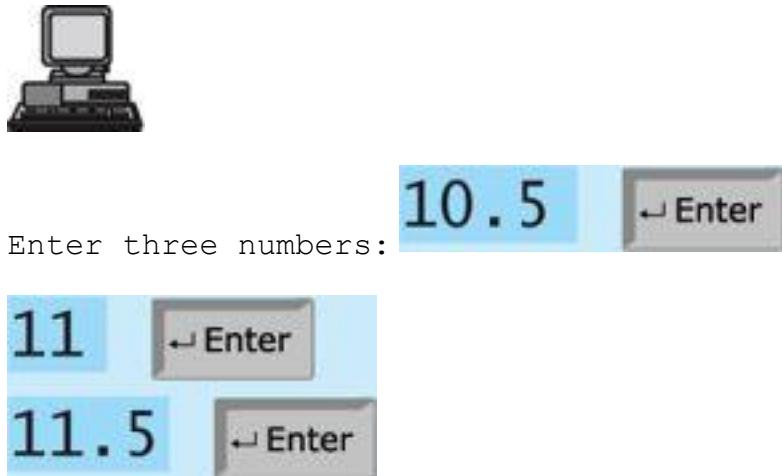
```
import class
1 import java.util.Scanner; // Scanner is in the java.util package
2
3 public class ComputeAverage {
4     public static void main(String[] args) {
5         // Create a Scanner object
6         Scanner input = new Scanner(System.in);
7
8         // Prompt the user to enter three numbers
9         System.out.print("Enter three numbers: ");
10        double number1 = input.nextDouble();
11        double number2 = input.nextDouble();
12        double number3 = input.nextDouble();
13
14        // Compute average
15        double average = (number1 + number2 + number3) / 3;
16
17        // Display result
18        System.out.println("The average of " + number1 + " " + number2
19                      + " " + number3 + " is " + average);
20    }
21 }
```



Enter three numbers:

The average of 1.0 2.0 3.0 is 2.0

enter input in one line



The average of 10.5 11.0 11.5 is 11.0

enter input in multiple lines

The code for importing the `Scanner` class (line 1) and creating a `Scanner` object (line 6) are the same as in the preceding example as well as in all new programs you will write.

Line 9 prompts the user to enter three numbers. The numbers are read in lines 10–12. You may enter three numbers separated by spaces, then press the *Enter* key, or enter each number followed by a press of the *Enter* key, as shown in the sample runs of this program.

2.4 Identifiers

As you see in [Listing 2.3](#), `ComputeAverage`, `main`, `input`, `number1`, `number2`, `number3`, and so on are the names of things that appear in the program. Such names are called *identifiers*. All identifiers must obey the following rules:

- An identifier is a sequence of characters that consists of letters, digits, underscores(`_`), and dollar signs (`$`).
- An identifier must start with a letter, an underscore (`_`), or a dollar sign (`$`). It cannot start with a digit.
- An identifier cannot be a reserved word. (See [Appendix A](#), “Java Keywords,” for a list of reserved words.)
- An identifier cannot be `true`, `false`, or `null`.

- An identifier can be of any length.

identifier naming rules

For example, `$2`, `ComputeArea`, `area`, `radius`, and `showMessageDialog` are legal identifiers, whereas `2A` and `d+4` are not because they do not follow the rules. The Java compiler detects illegal identifiers and reports syntax errors.



Note

Since Java is case sensitive, `area`, `Area`, and `AREA` are all different identifiers.

case sensitive



Tip

Identifiers are for naming variables, constants, methods, classes, and packages. Descriptive identifiers make programs easy to read.

descriptive names



Tip

Do not name identifiers with the `$` character. By convention, the `$` character should be used only in mechanically generated source code.

the `$` character

2.5 Variables

As you see from the programs in the preceding sections, variables are used to store values to be used later in a program. They are called variables because their values can be changed. In the program in [Listing 2.2](#), `radius` and `area` are variables of double-precision, floating-point type. You can assign any numerical value to `radius` and `area`, and the values of `radius` and `area` can be reassigned. For example, you can write the code shown below to compute the area for different radii:

```

// Compute the first area
radius = 1.0;
area = radius * radius * 3.14159;
System.out.println("The area is " + area + " for radius "
+ radius);
// Compute the second area
radius = 2.0;
area = radius * radius * 3.14159;
System.out.println("The area is " + area + " for radius "
+ radius);

```

why called variables?

Variables are for representing data of a certain type. To use a variable, you declare it by telling the compiler its name as well as what type of data it can store. The *variable declaration* tells the compiler to allocate appropriate memory space for the variable based on its data type. The syntax for declaring a variable is

```
datatype variableName;
```

Here are some examples of variable declarations:

```

int count;           // Declare count to be an integer
variable;
double radius;      // Declare radius to be a double
variable;
double interestRate; // Declare interestRate to be a
double variable;

```

declaring variables

The examples use the data types **int**, **double**, and **char**. Later you will be introduced to additional data types, such as **byte**, **short**, **long**, **float**, **char**, and **boolean**.

If variables are of the same type, they can be declared together, as follows:

```
datatype variable1, variable2,..., variablen;
```

The variables are separated by commas. For example,

```
int i, j, k; // Declare i, j, and k as int variables
```



Note

By convention, variable names are in lowercase. If a name consists of several words, concatenate all of them and capitalize the first letter of each word except the first.

Examples of variables are **radius** and **interestRate**.

naming variables

Variables often have initial values. You can declare a variable and initialize it in one step. Consider, for instance, the following code:

```
int count = 1;
```

initializing variables

This is equivalent to the next two statements:

```
int count;  
x = 1;
```

You can also use a shorthand form to declare and initialize variables of the same type together. For example,

```
int i = 1, j = 2;
```



Tip

A variable must be declared before it can be assigned a value. A variable declared in a method must be assigned a value before it can be used.

Whenever possible, declare a variable and assign its initial value in one step. This will make the program easy to read and avoid programming errors.

2.6 Assignment Statements and Assignment Expressions

After a variable is declared, you can assign a value to it by using an *assignment statement*. In Java, the equal sign (`=`) is used as the *assignment operator*. The syntax for assignment statements is as follows:

```
variable = expression;
```

assignment statement

assignment operator

An *expression* represents a computation involving values, variables, and operators that, taking them together, evaluates to a value. For example, consider the following code:

expression

```
int x = 1; // Assign 1 to variable x  
double radius = 1.0; // Assign 1.0 to variable radius
```

```
x = 5 * (3 / 2) + 3 * 2; // Assign the value of the  
expression to x  
x = y + 1; // Assign the addition of y and  
1 to x  
area = radius * radius * 3.14159; // Compute area
```

A variable can also be used in an expression. For example,

```
x = x + 1;
```

In this assignment statement, the result of `x + 1` is assigned to `x`. If `x` is `1` before the statement is executed, then it becomes `2` after the statement is executed.

To assign a value to a variable, the variable name must be on the left of the assignment operator. Thus, `1 = x` would be wrong.



Note

In mathematics, `x = 2 * x + 1` denotes an equation. However, in Java, `x = 2 * x + 1` is an assignment statement that evaluates the expression `2 * x + 1` and assigns the result to `x`.

In Java, an assignment statement is essentially an expression that evaluates to the value to be assigned to the variable on the left-hand side of the assignment operator. For this reason, an assignment statement is also known as an *assignment expression*. For example, the following statement is correct:

assignment expression

```
System.out.println(x = 1);
```

which is equivalent to

```
x = 1;  
System.out.println(x);
```

The following statement is also correct:

```
i = j = k = 1;
```

which is equivalent to

```
k = 1;  
j = k;  
i = j;
```



Note

In an assignment statement, the data type of the variable on the left must be compatible with the data type of the value on the right. For example, `int x = 1.0` would be illegal, because the data type of `x` is `int`. You cannot assign a `double` value (`1.0`) to an `int` variable without using type casting. Type casting is introduced in §2.11 “Numeric Type Conversions.”

2.7 Named Constants

The value of a variable may change during the execution of a program, but a *named constant* or simply *constant* represents permanent data that never changes. In our `ComputeArea` program, `PI` is a constant. If you use it frequently, you don’t want to keep typing `3.14159`; instead, you can declare a constant for π . Here is the syntax for declaring a constant:

```
constant
final datatype CONSTANTNAME = VALUE;
```

A constant must be declared and initialized in the same statement. The word `final` is a Java keyword for declaring a constant. For example, you can declare π as a constant and rewrite [Listing 2.1](#) as follows:

```
// ComputeArea.java: Compute the area of a circle
public class ComputeArea {
    public static void main(String[] args) {
        final double PI = 3.14159; // Declare a constant

        // Assign a radius
        double radius = 20;

        // Compute area
        double area = radius * radius * PI;

        // Display results
        System.out.println("The area for the circle of radius " +
            radius + " is " + area);
    }
}
```



Caution

By convention, constants are named in uppercase: `PI`, not `pi` or `Pi`.

naming constants



Note

There are three benefits of using constants: (1) you don't have to repeatedly type the same value; (2) if you have to change the constant value (e.g., from `3.14` to `3.14159` for `PI`), you need to change it only in a single location in the source code; (3) a descriptive name for a constant makes the program easy to read.

benefits of constants

2.8 Numeric Data Types and Operations

Every data type has a range of values. The compiler allocates memory space for each variable or constant according to its data type. Java provides eight primitive data types for numeric values, characters, and Boolean values. This section introduces numeric data types.

[Table 2.2](#) lists the six numeric data types, their ranges, and their storage sizes.

TABLE 2.2 Numeric Data Types

Name

Range

Storage Size

`byte`

-2^7 (-128) to $2^7 - 1$ (127)

8-bit signed

`short`

-2^{15} (-32768) to $2^{15}-1(32767)$

16-bit signed

int

-2^{31} (-2147483648) to $2^{31}-1(2147483647)$

32-bit signed

long

-2^{63} to $2^{63}-1$ (i.e., -9223372036854775808 to 9223372036854775807)

64-bit signed

float

Negative range: to $-3.4028235E + 38$ $-1.4E - 45$

Positive range: $1.4E - 45$ to $3.4028235E + 38$

32-bit IEEE 754

double

Negative range: $-1.7976931348623157E + 308$ to $-4.9E - 324$

Positive range: $4.9E - 324$ to $1.7976931348623157E + 308$

64-bit IEEE 754



Note

IEEE 754 is a standard approved by the Institute of Electrical and Electronics Engineers for representing floating-point numbers on computers. The standard has been widely adopted. Java has adopted the 32-bit **IEEE 754** for the **float** type and the 64-bit **IEEE 754** for the **double** type. The **IEEE 754** standard also defines special values as given in [Appendix E](#), “Special Floating-Point Values.”

Java uses four types for integers: **byte**, **short**, **int**, and **long**. Choose the type that is most appropriate for your variable. For example, if you know an integer stored in a variable

is within a range of byte, declare the variable as a **byte**. For simplicity and consistency, we will use **int** for integers most of the time in this book.

integer types

Java uses two types for floating-point numbers: **float** and **double**. The **double** type is twice as big as **float**. So, the **double** is known as *double precision*, **float** as *single precision*. Normally you should use the **double** type, because it is more accurate than the **float** type.

floating point



Caution

When a variable is assigned a value that is too large (*in size*) to be stored, it causes *overflow*. For example, executing the following statement causes *overflow*, because the largest value that can be stored in a variable of the **int** type is **2147483647**. **2147483648** is too large.

what is overflow?

```
int value = 2147483647 + 1; // value will actually be -  
2147483648
```

Likewise, executing the following statement causes *overflow*, because the smallest value that can be stored in a variable of the **int** type is **-2147483648**. **-2147483649** is too large in size to be stored in an **int** variable.

```
int value = -2147483648 - 1; // value will actually be  
2147483647
```

Java does not report warnings or errors on overflow. So be careful when working with numbers close to the maximum or minimum range of a given type.

When a floating-point number is too small (i.e., too close to zero) to be stored, it causes *underflow*. Java approximates it to zero. So normally you should not be concerned with underflow.

what is underflow?

2.8.1 Numeric Operators

The operators for numeric data types include the standard arithmetic operators: addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and remainder (`%`), as shown in [Table 2.3](#).

operators `+`, `-`, `*`, `/`, `%`

When both operands of a division are integers, the result of the division is an integer. The fractional part is truncated. For example, `5 / 2` yields `2`, not `2.5`, and `-5 / 2` yields `-2`, not `-2.5`. To perform regular mathematical division, one of the operands must be a floating-point number. For example, `5.0 / 2` yields `2.5`.

integer division

The `%` operator yields the remainder after division. The left-hand operand is the dividend and the right-hand operand the divisor. Therefore, `7 % 3` yields `1`, `12 % 4` yields `0`, `26 % 8` yields `2`, and `20 % 13` yields `7`.

TABLE 2.3 Numeric Operators

Name

Meaning

Example

Result

`+`

Addition

`34 + 1`

`35`

`-`

Subtraction

`34.0 - 0.1`

`33.9`

`*`

Multiplication

$300 * 30$

9000

/

Division

$1.0 / 2.0$

0.5

%

Remainder

$20 \% 3$

2

$$\begin{array}{r} 2 \\ 3 \sqrt{7} \\ \underline{-6} \\ 1 \end{array} \quad \begin{array}{r} 3 \\ 4 \sqrt{12} \\ \underline{-12} \\ 0 \end{array} \quad \begin{array}{r} 3 \\ 8 \sqrt{26} \\ \underline{-24} \\ 2 \end{array}$$

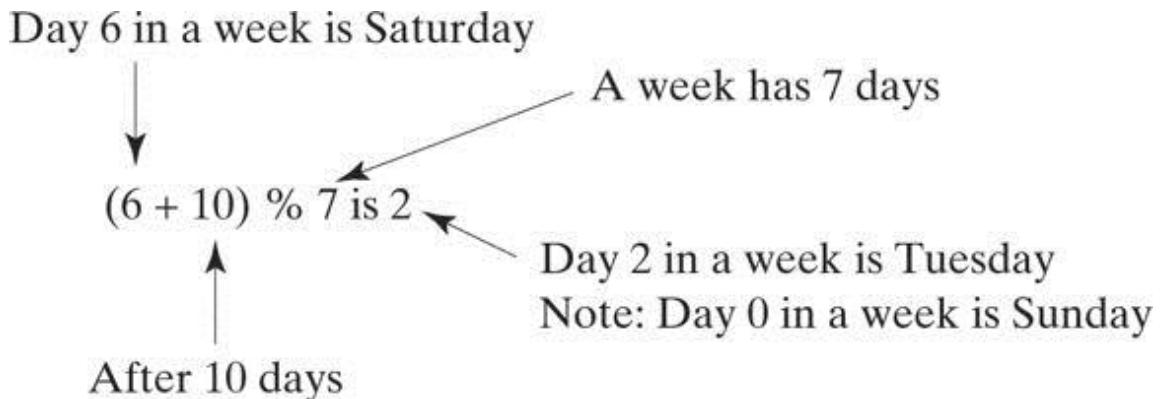
Divisor \longrightarrow 13 $\overline{)20}$ ← Dividend

1 ← Quotient

13
7 ← Remainder

The % operator is often used for positive integers but can be used also with negative integers and floating-point values. The remainder is negative only if the dividend is negative. For example, $-7 \% 3$ yields -1 , $-12 \% 4$ yields 0 , $-26 \% -8$ yields -2 , and $20 \% -13$ yields 7 .

Remainder is very useful in programming. For example, an even number % 2 is always 0 and an odd number % 2 is always 1. So you can use this property to determine whether a number is even or odd. If today is Saturday, it will be Saturday again in 7 days. Suppose you and your friends are going to meet in 10 days. What day is in 10 days? You can find that the day is Tuesday using the following expression:



[Listing 2.4](#) gives a program that obtains minutes and remaining seconds from an amount of time in seconds. For example, 500 seconds contains 8 minutes and 20 seconds.

LISTING 2.4 `DisplayTime.java`

```

1 import java.util.Scanner;
2
3 public class DisplayTime {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6         // Prompt the user for input
7         System.out.print("Enter an integer for seconds: ");
8         int seconds = input.nextInt();
9
10        int minutes = seconds / 60; // Find minutes in seconds
11        int remainingSeconds = seconds % 60; // Seconds remaining
12        System.out.println(seconds + " seconds is " + minutes +
13                            " minutes and " + remainingSeconds + " seconds");
14    }
15 }
```



500 ↵ Enter

Enter an integer for seconds:

500 seconds is 8 minutes and 20 seconds



line #

seconds

minutes

remainingSeconds

8

500

10

8

11

20

The `nextInt()` method (line 8) reads an integer for `seconds`. Line 4 obtains the minutes using `seconds / 60`. Line 5 (`seconds % 60`) obtains the remaining seconds after taking away the minutes.

The `+` and `-` operators can be both unary and binary. A *unary* operator has only one operand; a *binary* operator has two. For example, the `-` operator in `-5` is a unary operator to negate number `5`, whereas the `-` operator in `4-5` is a binary operator for subtracting `5` from `4`.

unary operator

binary operator



Note

Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy. For example,

```
System.out.println(1.0 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);  
displays 0.5000000000000001, not 0.5, and  
System.out.println(1.0 - 0.9);  
displays 0.0999999999999998, not 0.1. Integers are  
stored precisely. Therefore, calculations with integers  
yield a precise integer result.
```

floating-point approximation

2.8.2 Numeric Literals

A *literal* is a constant value that appears directly in a program. For example, `34` and `0.305` are literals in the following statements:

literal

```
int numberOfYears = 34;  
double weight = 0.305;
```

Integer Literals

An integer literal can be assigned to an integer variable as long as it can fit into the variable. A compile error will occur if the literal is too large for the variable to hold. The statement `byte b = 128`, for example, will cause a compile error, because 128 cannot be stored in a variable of the `byte` type. (Note that the range for a byte value is from –128 to 127.)

An integer literal is assumed to be of the `int` type, whose value is between -2^{31} (-2147483648) and $2^{31}-1$ (2147483647). To denote an integer literal of the `long` type, append the letter `L` or `l` to it (e.g., `2147483648L`). `L` is preferred because `l` (lowercase L) can easily be confused with 1 (the digit one). To write integer `2147483648` in a Java program, you have to write it as `2147483648L`, because `2147483648` exceeds the range for the `int` value.

`long` literal



Note

By default, an integer literal is a decimal integer number. To denote an octal integer literal, use a leading `0` (zero), and to denote a hexadecimal integer literal, use a leading `0x` or `0X` (zero x). For example, the following code displays the decimal value `65535` for hexadecimal number `FFFF`.

```
System.out.println(0xFFFF);
```

Hexadecimal numbers, binary numbers, and octal numbers are introduced in [Appendix E](#), “Number Systems.”

octal and hex literals

Floating-Point Literals

Floating-point literals are written with a decimal point. By default, a floating-point literal is treated as a `double` type value. For example, `5.0` is considered a `double` value, not a `float` value. You can make a number a `float` by appending the letter `f` or `F`, and you can make a number a `double` by appending the letter `d` or `D`. For example, you can use `100.2f` or `100.2F` for a `float` number, and `100.2d` or `100.2D` for a `double` number.

suffix d or D

suffix f or F



Note

The `double` type values are more accurate than the `float` type values. For example,

`double` vs. `float`

```
System.out.println("1.0 / 3.0 is " + 1.0 / 3.0);
displays 1.0 / 3.0 is 0.3333333333333333.
System.out.println("1.0F / 3.0F is " + 1.0F / 3.0F);
displays 1.0F / 3.0F is 0.33333334.
```

Scientific Notation

Floating-point literals can also be specified in scientific notation; for example, `1.23456e+2`, the same as `1.23456e2`, is equivalent to $1.23456 \times 10^2 = 123.456$, and

1.23456e-2 is equivalent to $1.23456 \times 10^{-2} = 0.0123456$. **E** (or **e**) represents an exponent and can be in either lowercase or uppercase.



Note

The **float** and **double** types are used to represent numbers with a decimal point. Why are they called *floating-point numbers*? These numbers are stored in scientific notation. When a number such as **50.534** is converted into scientific notation, such as **5.0534e+1**, its decimal point is moved (i.e., floated) to a new position.

why called floating-point?

2.8.3 Evaluating Java Expressions

Writing a numeric expression in Java involves a straightforward translation of an arithmetic expression using Java operators. For example, the arithm

$$\frac{3 + 4x}{5} - \frac{10(y - 5)(a + b + c)}{x} + 9\left(\frac{4}{x} + \frac{9+x}{y}\right)$$

can be translated into a Java expression as:

```
(3 + 4 * x) / 5 - 10 * (y - 5) * (a + b + c) / x +
9 * (4 / x + (9 + x) / y)
```

Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression are the same. Therefore, you can safely apply the arithmetic rule for evaluating a Java expression. Operators contained within pairs of parentheses are evaluated first. Parentheses can be nested, in which case the expression in the inner parentheses is evaluated first. Multiplication, division, and remainder operators are applied next. If an expression contains several multiplication, division, and remainder operators, they are applied from left to right. Addition and subtraction operators are applied last. If an expression contains several addition and subtraction operators, they are applied from left to right. Here is an example of how an expression is evaluated:

evaluating an expression

$$\begin{aligned}
 & 3 + 4 * 4 + 5 * (4 + 3) - 1 \\
 & 3 + 4 * 4 + 5 * 7 - 1 \quad (1) \text{ inside parentheses first} \\
 & 3 + 16 + 5 * 7 - 1 \quad (2) \text{ multiplication} \\
 & 3 + 16 + 35 - 1 \quad (3) \text{ multiplication} \\
 & 19 + 35 - 1 \quad (4) \text{ addition} \\
 & 54 - 1 \quad (5) \text{ addition} \\
 & 53 \quad (6) \text{ subtraction}
 \end{aligned}$$

[Listing 2.5](#) gives a program that converts a Fahrenheit degree to Celsius using the formula

$$celsius = \left(\frac{5}{9}\right)(fahrenheit - 32)$$

celsius

LISTING 2.5 `FahrenheitToCelsius.java`

```

1 import java.util.Scanner;
2
3 public class FahrenheitToCelsius {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6
7         System.out.print("Enter a degree in Fahrenheit: ");
8         double fahrenheit = input.nextDouble();
9
10        // Convert Fahrenheit to Celsius
11        double celsius = (5.0 / 9) * (fahrenheit - 32);      divide
12        System.out.println("Fahrenheit " + fahrenheit + " is " +
13                           celsius + " in Celsius");
14    }
15 }
```



Enter a degree in Fahrenheit:

100

```
Fahrenheit 100.0 is 37.7777777777778 in Celsius
```



line#

fahrenheit

celsius

8

100

11

37.7777777777778

Be careful when applying division. Division of two integers yields an integer in Java. $\frac{5}{9}$ is translated to `5.0 / 9` instead of `5/9` in line 11, because `5/9` yields `0` in Java.

integer vs. decimal division

2.9 Problem: Displaying the Current Time

The problem is to develop a program that displays the current time in GMT (Greenwich Mean Time) in the format hour:minute:second, such as 13:19:8.



Video Note

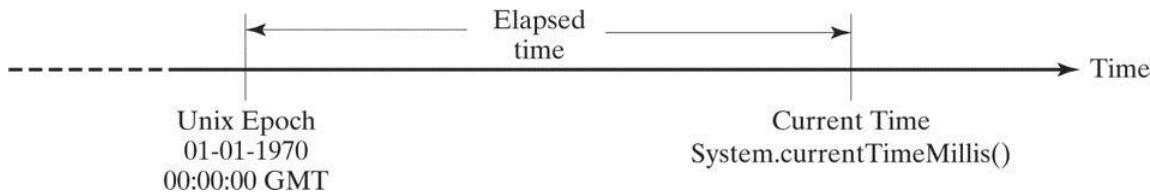
Use operators / and %

The `currentTimeMillis` method in the `System` class returns the current time in milliseconds elapsed since the time `00:00:00` on January 1, 1970 GMT, as shown in [Figure 2.2](#). This time is known as the *Unix epoch*, because `1970` was the year when the Unix operating system was formally introduced.

currentTimeMillis

Unix epoch

FIGURE 2.2 The `System.currentTimeMillis()` returns the number of milliseconds since the Unix epoch.



You can use this method to obtain the current time, and then compute the current second, minute, and hour as follows.

1. Obtain the total milliseconds since midnight, Jan 1, 1970, in `totalMilliseconds` by invoking `System.currentTimeMillis()` (e.g., `1203183086328` milliseconds).
2. Obtain the total seconds `totalSeconds` by dividing `totalMilliseconds` by `1000` (e.g., `1203183086328` milliseconds / `1000` = `1203183086` seconds).
3. Compute the current second from `totalSeconds % 60` (e.g., `1203183086` seconds % `60` = `26`, which is the current second).
4. Obtain the total minutes `totalMinutes` by dividing `totalSeconds` by `60` (e.g., `1203183086` seconds / `60` = `20053051` minutes).
5. Compute the current minute from `totalMinutes % 60` (e.g., `20053051` minutes % `60` = `31`, which is the current minute).
6. Obtain the total hours `totalHours` by dividing `totalMinutes` by `60` (e.g., `20053051` minutes / `60` = `334217` hours).
7. Compute the current hour from `totalHours % 24` (e.g., `334217` hours % `24` = `17`, which is the current hour).

[Listing 2.6](#) gives the complete program.

LISTING 2.6 ShowCurrentTime.java

```
1 public class ShowcurrentTime {
2     public static void main(String[] args) {
3         // Obtain the total milliseconds since midnight, Jan 1, 1970
4         long totalMilliseconds = System.currentTimeMillis();
5
6         // Obtain the total seconds since midnight, Jan 1, 1970
7         long totalSeconds = totalMilliseconds / 1000;
8
9         // Compute the current second in the minute in the hour
10        long currentSecond = (int)(totalSeconds % 60);
11
12        // Obtain the total minutes
13        long totalMinutes = totalSeconds / 60;
14
15        // Compute the current minute in the hour
16        long currentMinute = totalMinutes % 60;
17
18        // Obtain the total hours
19        long totalHours = totalMinutes / 60;
20
21        // Compute the current hour
22        long currentHour = totalHours % 24;
23
24        // Display results
25        System.out.println("Current time is " + currentHour + ":"
26                           + currentMinute + ":" + currentSecond + " GMT");
27    }
28 }
```



Current time is 17:31:26 GMT

| variables | line# 4 | 7 | 10 | 13 | 16 | 19 | 22 |
|-------------------|---------|---------------|------------|----|----------|----|--------|
| totalMilliseconds | | 1203183086328 | | | | | |
| totalSeconds | | | 1203183086 | | | | |
| currentSecond | | | | 26 | | | |
| totalMinutes | | | | | 20053051 | | |
| currentMinute | | | | | | 31 | |
| totalHours | | | | | | | 334217 |
| currentHour | | | | | | | 17 |

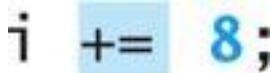
When `System.currentTimeMillis()` (line 4) is invoked, it returns the difference, measured in milliseconds, between the current GMT and midnight, January 1, 1970 GMT. This method returns the milliseconds as a `long` value. So, all the variables are declared as the `long` type in this program.

2.10 Shorthand Operators

Very often the current value of a variable is used, modified, and then reassigned back to the same variable. For example, the following statement adds the current value of `i` with `8` and assigns the result back to `i`:

```
i = i + 8;
```

Java allows you to combine assignment and addition operators using a shorthand operator. For example, the preceding statement can be written as:



The `+=` is called the *addition assignment operator*. Other shorthand operators are shown in [Table 2.4](#).

TABLE 2.4 Shorthand Operators

Operator

Name

Example

Equivalent

`+=`

Addition assignment

`i += 8`

`i = i + 8`

`-=`

Subtraction assignment

`i -= 8`

`i = i - 8`

`*=`

Multiplication assignment

`i *= 8`

`i = i * 8`

`/=`

Division assignment

`i /= 8`

`i = i / 8`

`%=`

Remainder assignment

`i %= 8`

`i = i % 8`



Caution

There are no spaces in the shorthand operators. For example, `+ =` should be `+=`.



Note

Like the assignment operator (`=`), the operators (`+=`, `-=`, `*=`, `/=`, `%=`) can be used to form an assignment statement as well as an expression. For example, in the following code, `x += 2` is a statement in the first line and an expression in the second line.

```
x += 2; // Statement  
System.out.println(x += 2); // Expression
```

There are two more shorthand operators for incrementing and decrementing a variable by 1. These are handy, because that's often how much the value needs to be changed. The two operators are `++` and `--`. For example, the following code increments `i` by 1 and decrements `j` by 1.

```
int i = 3, j = 3;  
i++; // i becomes 4  
j--; // j becomes 2
```

The `++` and `--` operators can be used in prefix or suffix mode, as shown in [Table 2.5](#).

TABLE 2.5 Increment and Decrement Operators

Operator

Name

Description

Example (assume i = 1)

`++var`

preincrement

Increment `var` by 1 and use the new `var` value

```
int j = ++i; // j is 2, // i is 2
```

`var++`

postincrement

Increment `var` by 1, but use the original `var` value

```
int j = i++; // j is 1, // i is 2
```

`--var`

predecrement

Decrement `var` by 1 and use the new `var` value

```
int j = --i; // j is 0, // i is 0
```

var--

postdecrement

Decrement var by 1 and use the original var value

```
int j = ++i; // j is 1, // i is 0
```

If the operator is *before* (prefixed to) the variable, the variable is incremented or decremented by 1, then the *new* value of the variable is returned. If the operator is *after* (suffixed to) the variable, then the variable is incremented or decremented by 1, but the original *old* value of the variable is returned. Therefore, the prefixes `++x` and `--x` are referred to, respectively, as the *preincrement operator* and the *predecrement operator*; and the suffixes `x++` and `x--` are referred to, respectively, as the *postincrement operator* and the *postdecrement operator*. The prefix form of `++` (or `--`) and the suffix form of `++` (or `--`) are the same if they are used in isolation, but they cause different effects when used in an expression. The following code illustrates this:

preincrement, predecrement

postincrement, postdecrement

```
int i = 10;  
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;  
i = i + 1;
```

In this case, `i` is incremented by 1, then the *old* value of `i` is returned and used in the multiplication. So `newNum` becomes 100. If `i++` is replaced by `++i` as follows,

```
int i = 10;  
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;  
int newNum = 10 * i;
```

`i` is incremented by 1, and the *new* value of `i` is returned and used in the multiplication. Thus `newNum` becomes 110.

Here is another example:

```
double x = 1.0;  
double y = 5.0;  
double z = x-- + (++y);
```

After all three lines are executed, `y` becomes 6.0, `z` becomes 7.0, and `x` becomes 0.0.

The increment operator `++` and the decrement operator `--` can be applied to all integer and floating-point types. These operators are often used in loop statements. A *loop statement* is a construct that controls how many times an operation or a sequence of operations is performed in succession. This construct, and the topic of loop statements, are introduced in [Chapter 4](#), “Loops.”



Tip

Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables or the same variable multiple times, such as this one: `int k = ++i + i.`

2.11 Numeric Type Conversions

Can you perform binary operations with two operands of different types? Yes. If an integer and a floating-point number are involved in a binary operation, Java automatically converts the integer to a floating-point value. So, `3 * 4.5` is same as `3.0 * 4.5`.

You can always assign a value to a numeric variable whose type supports a larger range of values; thus, for instance, you can assign a `long` value to a `float` variable. You cannot, however, assign a value to a variable of a type with smaller range unless you use *type casting*. Casting is an operation that converts a value of one data type into a value of another data type. Casting a variable of a type with a small range to a variable of a type with a larger range is known as *widening a type*. Casting a variable of a type with a large range to a variable of a type with a smaller range is known as *narrowing a type*. Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly.

widening a type

narrowing a type

The syntax is the target type in parentheses, followed by the variable’s name or the value to be cast. For example, the following statement

```
type casting  
System.out.println((int)1.7);
```

displays `1`. When a `double` value is cast into an `int` value, the fractional part is truncated.

The following statement

```
System.out.println((double)1 / 2);
```

displays **0.5**, because **1** is cast to **1.0** first, then **1.0** is divided by **2**. However, the statement

```
System.out.println(1 / 2);
```

displays **0**, because **1** and **2** are both integers and the resulting value should also be an integer.



Caution

Casting is necessary if you are assigning a value to a variable of a smaller type range, such as assigning a **double** value to an **int** variable. A compile error will occur if casting is not used in situations of this kind. Be careful when using casting. Loss of information might lead to inaccurate results.

possible loss of precision



Note

Casting does not change the variable being cast. For example, **d** is not changed after casting in the following code:

```
double d = 4.5;
int i = (int)d; // i becomes 4, but d is not changed,
still 4.5
```



Note

To assign a variable of the **int** type to a variable of the **short** or **byte** type, explicit casting must be used. For example, the following statements have a compile error:

```
int i = 1;
byte b = i; // Error because explicit casting is required
```

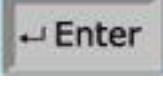
However, so long as the integer literal is within the permissible range of the target variable, explicit casting is not needed to assign an integer literal to a variable of the **short** or **byte** type. Please refer to §[2.8.2](#), “Numeric Literals.”

[Listing 2.7](#) gives a program that displays the sales tax with two digits after the decimal point.

LISTING 2.7 SalesTax.java

```
1 import java.util.Scanner;  
2  
3 public class SalesTax {  
4     public static void main(String[] args) {  
5         Scanner input = new Scanner(System.in);  
6  
7         System.out.print("Enter purchase amount: ");  
8         double purchaseAmount = input.nextDouble();  
9  
10        double tax = purchaseAmount * 0.06;  
11        System.out.println("Sales tax is " + (int)(tax * 100) / 100.0);  
12    }  
13 }
```



Enter purchase amount: **197.55**  ↵ Enter

Sales tax is 11.85



line

purchaseAmount

tax

output

8

197.55

10

11.853

11

11.85

Variable `purchaseAmount` is **197.55** (line 8). The sales tax is **6%** of the purchase, so the `tax` is evaluated as **11.853** (line 10). Note that

formatting numbers

```
tax * 100 is 1185.3  
(int) (tax * 100) is 1185  
(int) (tax * 100) / 100.0 is 11.85
```

So, the statement in line 11 displays the tax **11.85** with two digits after the decimal point.

2.12 Problem: Computing Loan Payments

The problem is to write a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. The program lets the user enter the interest rate, number of years, and loan amount, and displays the monthly and total payments.



Video Note

Program computations

The formula to compute the monthly payment is as follows:

$$\text{monthlyPayment} = \frac{\text{loanAmount} \times \text{monthlyInterestRate}}{1 - \frac{1}{(1 + \text{monthlyInterestRate})^{\text{numberOfYears} \times 12}}}$$

You don't have to know how this formula is derived. Nonetheless, given the monthly interest rate, number of years, and loan amount, you can use it to compute the monthly payment.

In the formula, you have to compute $(1 + monthlyInterestRate)^{numberYears \times 12}$. The **pow(a, b)** method in the **Math** class can be used to compute a^b . The **Math** class, which comes with the Java API, is available to all Java programs. For example,

pow(a, b) method

```
System.out.println(Math.pow(2, 3)); // Display 8  
System.out.println(Math.pow(4, 0.5)); // Display 4
```

$(1 + monthlyInterestRate)^{numberYears \times 12}$ can be computed using
Math.pow(1+monthlyInterestRate,numberOfyears * 12.)

Here are the steps in developing the program:

1. Prompt the user to enter the annual interest rate, number of years, and loan amount.
2. Obtain the monthly interest rate from the annual interest rate.
3. Compute the monthly payment using the preceding formula.
4. Compute the total payment, which is the monthly payment multiplied by **12** and multiplied by the number of years.
5. Display the monthly payment and total payment.

[Listing 2.8](#) gives the complete program.

LISTING 2.8 ComputeLoan.java

```

1 import java.util.Scanner; import class
2
3 public class ComputeLoan {
4     public static void main(String[] args) {
5         // Create a Scanner
6         Scanner input = new Scanner(System.in); create a Scanner
7
8         // Enter yearly interest rate
9         System.out.print("Enter yearly interest rate, for example 8.25: ");
10        double annualInterestRate = input.nextDouble(); enter interest rate
11
12        // Obtain monthly interest rate
13        double monthlyInterestRate = annualInterestRate / 1200;
14
15        // Enter number of years
16        System.out.print(
17            "Enter number of years as an integer, for example 5: ");
18        int numberOfYears = input.nextInt(); enter years
19
20        // Enter loan amount
21        System.out.print("Enter loan amount, for example 120000.95: ");
22
23        double loanAmount = input.nextDouble();
24
25        // Calculate payment
26        double monthlyPayment = loanAmount * monthlyInterestRate / (1
27            - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
28        double totalPayment = monthlyPayment * numberOfYears * 12;
29
30        // Display results
31        System.out.println("The monthly payment is " +
32            (int)(monthlyPayment * 100) / 100.0);
33        System.out.println("The total payment is " +
34            (int)(totalPayment * 100) / 100.0);
35    }

```



Enter yearly interest rate, for example 8.25:

5.75

Enter number of years as an integer, for example

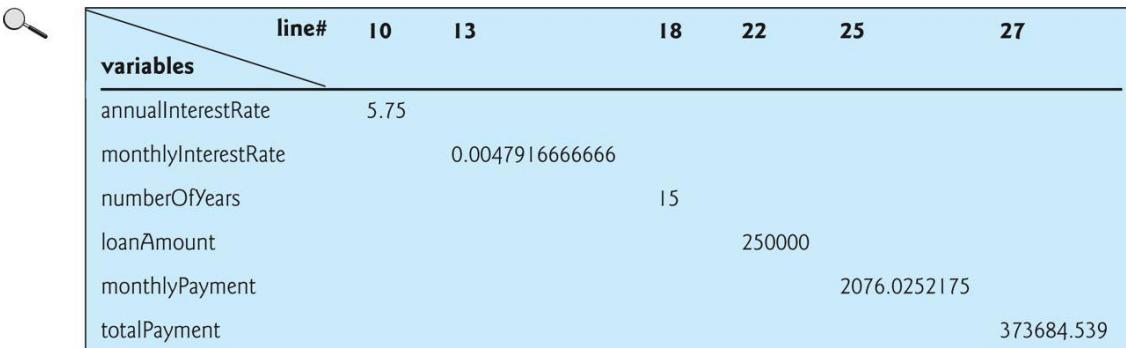
5: 15

Enter loan amount, for example 120000.95:

250000

The monthly payment is 2076.02

The total payment is 373684.53



| variables | line# | 10 | 13 | 18 | 22 | 25 | 27 |
|---------------------|-------|----|------|-----------------|----|--------|--------------|
| annualInterestRate | | | 5.75 | | | | |
| monthlyInterestRate | | | | 0.0047916666666 | | | |
| numberOfYears | | | | | 15 | | |
| loanAmount | | | | | | 250000 | |
| monthlyPayment | | | | | | | 2076.0252175 |
| totalPayment | | | | | | | 373684.539 |

Line 10 reads the yearly interest rate, which is converted into monthly interest rate in line 13. If you entered an input other than a numeric value, a runtime error would occur.

Choose the most appropriate data type for the variable. For example, `numberOfYears` is best declared as an `int` (line 18), although it could be declared as a `long`, `float`, or `double`. Note that `byte` might be the most appropriate for `numberOfYears`. For simplicity, however, the examples in this book will use `int` for integer and `double` for floating-point values.

The formula for computing the monthly payment is translated into Java code in lines 25–27.

Casting is used in lines 31 and 33 to obtain a new `monthlyPayment` and `totalPayment` with two digits after the decimal point.

The program uses the `Scanner` class, imported in line 1. The program also uses the `Math` class; why isn't it imported? The `Math` class is in the `java.lang` package. All classes in the `java.lang` package are implicitly imported. So, there is no need to explicitly import the `Math` class.

`java.lang` package

2.13 Character Data Type and Operations

The character data type, `char`, is used to represent a single character. A character literal is enclosed in single quotation marks. Consider the following code:

```
char type
char letter = &A&;
char numChar = &4&;
```

The first statement assigns character `A` to the `char` variable `letter`. The second statement assigns digit character `4` to the `char` variable `numChar`.



Caution

A string literal must be enclosed in quotation marks. A character literal is a single character enclosed in single quotation marks. So `"A"` is a string, and `'A'` is a character.

`char` literal

2.13.1 Unicode and ASCII code

Computers use binary numbers internally. A character is stored in a computer as a sequence of 0s and 1s. Mapping a character to its binary representation is called *encoding*. There are different ways to encode a character. How characters are encoded is defined by an *encoding scheme*.

character encoding

Java supports *Unicode*, an encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world's diverse languages. Unicode was originally designed as a 16-bit character encoding. The primitive data type `char` was intended to take advantage of this design by providing a simple data type that could hold any character. However, it turned out that the `65,536` characters possible in a 16-bit encoding are not sufficient to represent all the characters in the world. The Unicode standard therefore has been extended to allow up to `1,112,064` characters. Those characters that go beyond the original 16-bit limit are called *supplementary characters*. Java supports supplementary characters. The processing and representing of supplementary characters are beyond the scope of this book. For simplicity, this book considers only the original 16-bit Unicode characters. These characters can be stored in a `char` type variable.

Unicode

original Unicode

supplementary Unicode

A 16-bit Unicode takes two bytes, preceded by `\u`, expressed in four hexadecimal digits that run from `&\u0000&` to `&\uFFFF&`. For example, the word “welcome” is translated

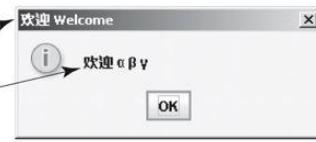
欢迎

into Chinese using two characters, 欢迎. The Unicodes of these two characters are “\u6B22\u8FCE”.

[Listing 2.9](#) gives a program that displays two Chinese characters and three Greek letters.

LISTING 2.9 DisplayUnicode.java

```
1 import javax.swing.JOptionPane;
2
3 public class DisplayUnicode {
4     public static void main(String[] args) {
5         JOptionPane.showMessageDialog(null,
6             "\u6B22\u8FCE \u03b1 \u03b2 \u03b3",
7             "\u6B22\u8FCE Welcome",
8             JOptionPane.INFORMATION_MESSAGE);
9     }
10 }
```



If no Chinese font is installed on your system, you will not be able to see the Chinese characters. The Unicodes for the Greek letters are \u03b1 \u03b2 \u03b3.

Most computers use *ASCII* (*American Standard Code for Information Interchange*), a 7-bit encoding scheme for representing all uppercase and lowercase letters, digits, punctuation marks, and control characters. Unicode includes ASCII code, with &\u0000& to &\u007F& corresponding to the 128 ASCII characters. (See [Appendix B](#), “The ASCII Character Set,” for a list of ASCII characters and their decimal and hexadecimal codes.) You can use ASCII characters such as &X&, &1&, and &\$& in a Java program as well as Unicodes. Thus, for example, the following statements are equivalent:

ASCII

```
char letter = &A&;
char letter = &\u0041&; // Character A's Unicode is 0041
```

Both statements assign character **A** to **char** variable **letter**.



Note

The increment and decrement operators can also be used on **char** variables to get the next or preceding Unicode character. For example, the following statements display character **b**.

char increment and decrement

```
char ch = &a&;  
System.out.println(++ch);
```

2.13.2 Escape Sequences for Special Characters

Suppose you want to print a message with quotation marks in the output. Can you write a statement like this?

```
System.out.println("He said "Java is fun"");
```

No, this statement has a syntax error. The compiler thinks the second quotation character is the end of the string and does not know what to do with the rest of characters.

backslash

To overcome this problem, Java defines escape sequences to represent special characters, as shown in [Table 2.6](#). An escape sequence begins with the backslash character (\) followed by a character that has a special meaning to the compiler.

TABLE 2.6 Java Escape Sequences

Character Escape Sequence

Name

Unicode Code

\b

Backspace

\u0008

\t

Tab

\u0009

\n

Linefeed

\u000A

\f

Formfeed

\u000C

\r

Carriage Return

\u000D

\\"

Backslash

\u005C

\&

Single Quote

\u0027

\"

Double Quote

\u0022

So, now you can print the quoted message using the following statement:

```
System.out.println("He said \"Java is fun\"");
```

The output is

```
He said "Java is fun"
```

2.13.3 Casting between char and Numeric Types

A **char** can be cast into any numeric type, and vice versa. When an integer is cast into a **char**, only its lower 16 bits of data are used; the other part is ignored. For example:

```
char ch = (char) 0xAB0041; // the lower 16 bits hex code  
0041 is  
                                // assigned to ch  
System.out.println(ch);      // ch is character A
```

When a floating-point value is cast into a **char**, the floating-point value is first cast into an **int**, which is then cast into a **char**.

```
char ch = (char) 65.25;      // decimal 65 is assigned to ch
```

```
System.out.println(ch); // ch is character A
```

When a `char` is cast into a numeric type, the character's Unicode is cast into the specified numeric type.

```
int i = (int) &a&; // the Unicode of character A is  
assigned to i  
System.out.println(i); // i is 65
```

Implicit casting can be used if the result of a casting fits into the target variable. Otherwise, explicit casting must be used. For example, since the Unicode of `&a&` is `97`, which is within the range of a byte, these implicit castings are fine:

```
byte b = &a&;  
int i = &a&;
```

But the following casting is incorrect, because the Unicode `\uFFFF4` cannot fit into a byte:

```
byte b = &\uFFF4&;
```

To force assignment, use explicit casting, as follows:

```
byte b = (byte) &\uFFF4&;
```

Any positive integer between `0` and `FFFF` in hexadecimal can be cast into a character implicitly. Any number not in this range must be cast into a `char` explicitly.



Note

All numeric operators can be applied to `char` operands. A `char` operand is automatically cast into a number if the other operand is a number or a character. If the other operand is a string, the character is concatenated with the string. For example, the following statements

numeric operators on characters

```
int i = &2& + &3&; // (int)&2& is 50 and (int)&3& is 51  
System.out.println("i is " + i); // i is 101  
int j = 2 + &a&; // (int)&a& is 97  
System.out.println("j is " + j); // j is 99  
System.out.println(j + " is the Unicode for character "  
+ (char)j);  
System.out.println("Chapter " + &2&);  
display  
i is 101  
j is 99  
99 is the Unicode for character c  
Chapter 2
```



Note

The Unicodes for lowercase letters are consecutive integers starting from the Unicode for `&a&`, then for `&b&`, `&c&`, and `&z&`. The same is true for the uppercase letters.

Furthermore, the Unicode for `&a&` is greater than the Unicode for `&A&`. So `&a& - &A&` is the same as `&b& - &B&`. For a lowercase letter `ch`, its corresponding uppercase letter is `(char)(&A& + (ch - &a&))`.

2.14 Problem: Counting Monetary Units

Suppose you want to develop a program that classifies a given amount of money into smaller monetary units. The program lets the user enter an amount as a `double` value representing a total in dollars and cents, and outputs a report listing the monetary equivalent in dollars, quarters, dimes, nickels, and pennies, as shown in the sample run.

Your program should report the maximum number of dollars, then the maximum number of quarters, and so on, in this order.

Here are the steps in developing the program:

1. Prompt the user to enter the amount as a decimal number, such as `11.56`.
2. Convert the amount (e.g., `11.56`) into cents (`1156`).
3. Divide the cents by `100` to find the number of dollars. Obtain the remaining cents using the cents remainder `100`.
4. Divide the remaining cents by `25` to find the number of quarters. Obtain the remaining cents using the remaining cents remainder `25`.
5. Divide the remaining cents by `10` to find the number of dimes. Obtain the remaining cents using the remaining cents remainder `10`.
6. Divide the remaining cents by `5` to find the number of nickels. Obtain the remaining cents using the remaining cents remainder `5`.
7. The remaining cents are the pennies.
8. Display the result

The complete program is given in [Listing 2.10](#).

LISTING 2.10 ComputeChange.java

```
import class          1 import java.util.Scanner;
                     2
                     3 public class ComputeChange {
                     4     public static void main(String[] args) {
                     5         // Create a Scanner
                     6         Scanner input = new Scanner(System.in);
                     7
                     8         // Receive the amount
                     9         System.out.print(
                     10            "Enter an amount in double, for example 11.56: ");
                     11            double amount = input.nextDouble();
                     12
                     13            int remainingAmount = (int)(amount * 100);
                     14
                     15            // Find the number of one dollars
                     16            int numberOfOneDollars = remainingAmount / 100;
                     17            remainingAmount = remainingAmount % 100;
                     18
                     19            // Find the number of quarters in the remaining amount
                     20            int numberOfQuarters = remainingAmount / 25;
                     21            remainingAmount = remainingAmount % 25;
                     22
                     23            // Find the number of dimes in the remaining amount
                     24            int numberOfDimes = remainingAmount / 10;
                     25            remainingAmount = remainingAmount % 10;
                     26
                     27            // Find the number of nickels in the remaining amount
                     28            int numberOfNickels = remainingAmount / 5;
                     29            remainingAmount = remainingAmount % 5;
                     30
                     31            // Find the number of pennies in the remaining amount
                     32            int numberOfPennies = remainingAmount;
                     33
                     34            // Display results
                     35            System.out.println("Your amount " + amount + " consists of \n" +
                     36                "\t" + numberOfOneDollars + " dollars\n" +
                     37                "\t" + numberOfQuarters + " quarters\n" +
                     38                "\t" + numberOfDimes + " dimes\n" +
                     39                "\t" + numberOfNickels + " nickels\n" +
                     40                "\t" + numberOfPennies + " pennies");
                     41    }
                     42 }
```



Enter an amount in double, for example 11.56:

Your amount 11.56 consists of

11 dollars
2 quarters
0 dimes
1 nickels
1 pennies

| variables | line# | 11 | 13 | 16 | 17 | 20 | 21 | 24 | 25 | 28 | 29 | 32 |
|--------------------|-------|----|-------|----|----|----|----|----|----|----|----|----|
| Amount | | | 11.56 | | | | | | | | | |
| remainingAmount | | | 1156 | | 56 | | 6 | | 6 | | 1 | |
| numberOfOneDollars | | | | 11 | | | | | | | | |
| numberOfQuarters | | | | | | 2 | | | | | | |
| numberOfDimes | | | | | | | 0 | | | | | |
| numberOfNickles | | | | | | | | 1 | | | | |
| numberOfPennies | | | | | | | | | | 1 | | |

The variable `amount` stores the amount entered from the console (line 11). This variable is not changed, because the amount has to be used at the end of the program to display the results. The program introduces the variable `remainingAmount` (line 13) to store the changing `remainingAmount`.

The variable `amount` is a `double` decimal representing dollars and cents. It is converted to an `int` variable `remainingAmount`, which represents all the cents. For instance, if `amount` is `11.56`, then the initial `remainingAmount` is `1156`. The division operator yields the integer part of the division. So `1156 / 100` is `11`. The remainder operator obtains the remainder of the division. So `1156 % 100` is `56`.

The program extracts the maximum number of singles from the total amount and obtains the remaining amount in the variable `remainingAmount` (lines 16–17). It then extracts the maximum number of quarters from `remainingAmount` and obtains a new `remainingAmount` (lines 20–21). Continuing the same process, the program finds the maximum number of dimes, nickels, and pennies in the remaining amount.

One serious problem with this example is the possible loss of precision when casting a `double` amount to an `int remainingAmount`. This could lead to an inaccurate result. If you try to enter the amount `10.03`, `10.03 * 100` becomes `1002.999999999999`. You will find that the program displays `10` dollars and `2` pennies. To fix the problem, enter the amount as an integer value representing cents (see Exercise 2.9).

loss of precision

As shown in the sample run, **0** dimes, **1** nickels, and **1** pennies are displayed in the result. It would be better not to display **0** dimes, and to display **1** nickel and **1** penny using the singular forms of the words. You will learn how to use selection statements to modify this program in the next chapter (see Exercise 3.7).

2.15 The String Type

The **char** type represents only one character. To represent a string of characters, use the data type called **String**. For example, the following code declares the message to be a string with value “Welcome to Java”.

```
String message = "Welcome to Java";
```

String is actually a predefined class in the Java library just like the classes **System**, **JOptionPane**, and **Scanner**. The **String** type is not a primitive type. It is known as a *reference type*. Any Java class can be used as a reference type for a variable. Reference data types will be thoroughly discussed in [Chapter 8](#), “Objects and Classes.” For the time being, you need to know only how to declare a **String** variable, how to assign a string to the variable, and how to concatenate strings.

As first shown in [Listing 2.1](#), two strings can be concatenated. The plus sign (+) is the concatenation operator if one of the operands is a string. If one of the operands is a nonstring (e.g., a number), the nonstring value is converted into a string and concatenated with the other string. Here are some examples:

concatenating strings and numbers

```
// Three strings are concatenated
String message = "Welcome " + "to " + "Java";
// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2
// String Supplement is concatenated with character B
String s1 = "Supplement" + &B&; // s1 becomes SupplementB
```

If neither of the operands is a string, the plus sign (+) is the addition operator that adds two numbers.

The shorthand **+=** operator can also be used for string concatenation. For example, the following code appends the string “and Java is fun” with the string “Welcome to Java” in **message**.

```
message += " and Java is fun";
```

So the new **message** is “Welcome to Java and Java is fun”.

Suppose that **i = 1** and **j = 2**, what is the output of the following statement?

```
System.out.println("i + j is " + i + j);
```

The output is “i + j is 12” because “i + j is ” is concatenated with the value of i first. To force i + j to be executed first, enclose i + j in the parentheses, as follows:

```
System.out.println("i + j is " + (i + j));
```

To read a string from the console, invoke the `next()` method on a `Scanner` object. For example, the following code reads three strings from the keyboard:

reading strings

```
Scanner input = new Scanner(System.in);
System.out.println("Enter three strings: ");
String s1 = input.next();
String s2 = input.next();
String s3 = input.next();
System.out.println("s1 is " + s1);
System.out.println("s2 is " + s2);
System.out.println("s3 is " + s3);
```



Welcome to Java

Enter a string:

s1 is Welcome

s2 is to

s3 is Java

The `next()` method reads a string that ends with a whitespace character (i.e., & &, &\t&, &\f&, &\r&, or &\n&).

You can use the `nextLine()` method to read an entire line of text. The `nextLine()` method reads a string that ends with the *Enter* key pressed. For example, the following statements read a line of text.

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a string: ");
String s = input.nextLine();
System.out.println("The string entered is " + s);
```



Welcome to Java

↵ Enter

Enter a string:

The string entered is "Welcome to Java"



Important Caution

To avoid *input errors*, do not use `nextLine()` after `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()`. The reasons will be explained in §9.7.3, “How Does `Scanner` Work?”

avoiding input errors

2.16 Programming Style and Documentation

Programming style deals with what programs look like. A program can compile and run properly even if written on only one line, but writing it all on one line would be bad programming style because it would be hard to read. *Documentation* is the body of explanatory remarks and comments pertaining to a program. Programming style and documentation are as important as coding. Good programming style and appropriate documentation reduce the chance of errors and make programs easy to read. So far you have learned some good programming styles. This section summarizes them and gives several guidelines. More detailed guidelines can be found in Supplement I.D, “Java Coding Style Guidelines,” on the Companion Website.

programming style

documentation

2.16.1 Appropriate Comments and Comment Styles

Include a summary at the beginning of the program to explain what the program does, its key features, and any unique techniques it uses. In a long program, you should also include comments that introduce each major step and explain anything that is difficult to read. It is

important to make comments concise so that they do not crowd the program or make it difficult to read.

In addition to line comment `//` and block comment `/*`, Java supports comments of a special type, referred to as *javadoc comments*. javadoc comments begin with `/**` and end with `*/`. They can be extracted into an HTML file using JDK's `javadoc` command. For more information, see java.sun.com/j2se/javadoc.

javadoc comment

Use javadoc comments (`/**... */`) for commenting on an entire class or an entire method. These comments must precede the class or the method header in order to be extracted in a javadoc HTML file. For commenting on steps inside a method, use line comments (`//`).

2.16.2 Naming Conventions

Make sure that you choose descriptive names with straightforward meanings for the variables, constants, classes, and methods in your program. Names are case sensitive. Listed below are the conventions for naming variables, methods, and classes.

- Use lowercase for variables and methods. If a name consists of several words, concatenate them into one, making the first word lowercase and capitalizing the first letter of each subsequent word—for example, the variables `radius` and `area` and the method `showInputDialog`.

naming variables and methods

- Capitalize the first letter of each word in a class name—for example, the class names `ComputeArea`, `Math`, and `JOptionPane`.

naming classes

- Capitalize every letter in a constant, and use underscores between words—for example, the constants `PI` and `MAX_VALUE`.

naming constants

It is important to follow the naming conventions to make programs easy to read.



Caution

Do not choose class names that are already used in the Java library. For example, since the `Math` class is defined in Java, you should not name your class `Math`.

naming classes



Tip

Avoid using abbreviations for identifiers. Using complete words is more descriptive. For example, `numberOfStudents` is better than `numStuds`, `numOfStuds`, or `numOfStudents`.

using full descriptive names

2.16.3 Proper Indentation and Spacing

A consistent indentation style makes programs clear and easy to read, debug, and maintain. *Indentation* is used to illustrate the structural relationships between a program's components or statements. Java can read the program even if all of the statements are in a straight line, but humans find it easier to read and maintain code that is aligned properly. Indent each subcomponent or statement at least *two* spaces more than the construct within which it is nested.

indent code

A single space should be added on both sides of a binary operator, as shown in the following statement:

```
int i= 3+4 * 4; ← Bad style
```

```
int i = 3 + 4 * 4; ← Good style
```

A single space line should be used to separate segments of the code to make the program easier to read.

2.16.4 Block Styles

A block is a group of statements surrounded by braces. There are two popular styles, *next-line* style and *end-of-line* style, as shown below.

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println("Block Styles");
    }
}
```

Next-line style

```
public class Test {
    public static void main(String[] args) {
        System.out.println("Block Styles");
    }
}
```

End-of-line style

The next-line style aligns braces vertically and makes programs easy to read, whereas the end-of-line style saves space and may help avoid some subtle programming errors. Both are acceptable block styles. The choice depends on personal or organizational preference. You should use a block style consistently. Mixing styles is not recommended. This book uses the *end-of-line* style to be consistent with the Java API source code.

2.17 Programming Errors

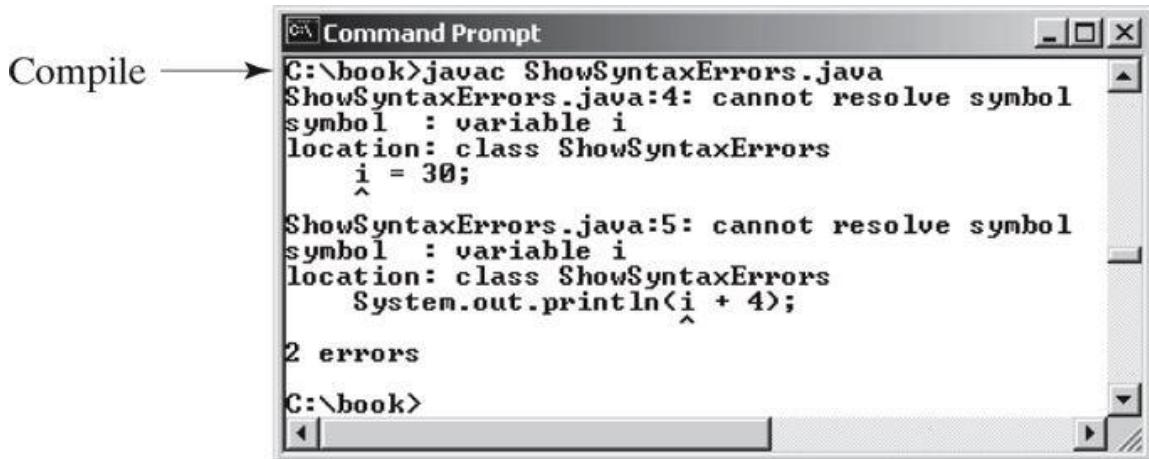
Programming errors are unavoidable, even for experienced programmers. Errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

2.17.1 Syntax Errors

Errors that occur during compilation are called *syntax errors* or *compile errors*. Syntax errors result from errors in code construction, such as mistyping a keyword, omitting some necessary punctuation, or using an opening brace without a corresponding closing brace. These errors are usually easy to detect, because the compiler tells you where they are and what caused them. For example, the following program has a syntax error, as shown in [Figure 2.3](#).

syntax errors

FIGURE 2.3 The compiler reports syntax errors.



```
1 // ShowSyntaxErrors.java: The program contains syntax errors
2 public class ShowSyntaxErrors {
3     public static void main(String[] args) {
4         i = 30;                                syntax error
5         System.out.println(i + 4);
6     }
7 }
```

Two errors are detected. Both are the result of not declaring variable `i`. Since a single error will often display many lines of compile errors, it is a good practice to start debugging from the top line and work downward. Fixing errors that occur earlier in the program may also fix additional errors that occur later.

2.17.2 Runtime Errors

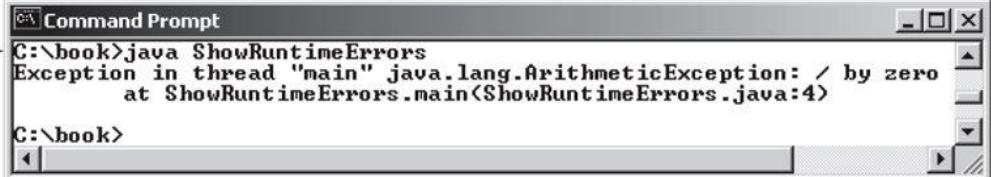
Runtime errors are errors that cause a program to terminate abnormally. They occur while a program is running if the environment detects an operation that is impossible to carry out. Input errors typically cause runtime errors.

runtime errors

An *input error* occurs when the user enters an unexpected input value that the program cannot handle. For instance, if the program expects to read in a number, but instead the user enters a string, this causes data-type errors to occur in the program. To prevent input errors, the program should prompt the user to enter values of the correct type. It may display a message such as “Please enter an integer” before reading an integer from the keyboard.

Another common source of runtime errors is division by zero. This happens when the divisor is zero for integer divisions. For instance, the following program would cause a runtime error, as shown in [Figure 2.4](#).

FIGURE 2.4 The runtime error causes the program to terminate abnormally.



Run →

```
C:\book>java ShowRuntimeErrors
Exception in thread "main" java.lang.ArithmaticException: / by zero
        at ShowRuntimeErrors.main(ShowRuntimeErrors.java:4)
C:\book>
```

runtime error

```
1 // ShowRuntimeErrors.java: Program contains runtime errors
2 public class ShowRuntimeErrors {
3     public static void main(String[] args) {
4         int i = 1 / 0;
5     }
6 }
```

2.17.3 Logic Errors

Logic errors occur when a program does not perform the way it was intended to. Errors of this kind occur for many different reasons. For example, suppose you wrote the following program to add `number1` to `number2`.

```
// ShowLogicErrors.java: The program contains a logic error
public class ShowLogicErrors {
    public static void main(String[] args) {
        // Add number1 to number2
        int number1 = 3;
        int number2 = 3;
        number2 += number1 + number2;
        System.out.println("number2 is " + number2);
    }
}
```

The program does not have syntax errors or runtime errors, but it does not print the correct result for `number2`. See if you can find the error.

2.17.4 Debugging

In general, syntax errors are easy to find and easy to correct, because the compiler gives indications as to where the errors came from and why they are wrong. Runtime errors are not difficult to find, either, since the reasons and locations of the errors are displayed on the console when the program aborts. Finding logic errors, on the other hand, can be very challenging.

Logic errors are called *bugs*. The process of finding and correcting errors is called *debugging*. A common approach is to use a combination of methods to narrow down to the part of the program where the bug is located. You can *hand-trace* the program (i.e., catch errors by reading the program), or you can insert print statements in order to show the values of the variables or the execution flow of the program. This approach might work for a short, simple program. But for a large, complex program, the most effective approach is to use a debugger utility.

bugs

debugging

hand traces



Pedagogical NOTE

An IDE not only helps debug errors but also is an effective pedagogical tool. Supplement II shows you how to use a debugger to trace programs and how debugging can help you to learn Java effectively.

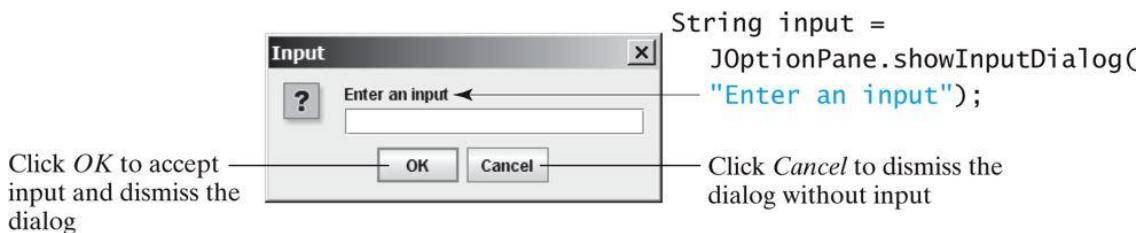
learning tool

2.18 (GUI) Getting Input from Input Dialogs

You can obtain input from the console. Alternatively, you may obtain input from an input dialog box by invoking the `JOptionPane.showInputDialog` method, as shown in [Figure 2.5](#).

`JOptionPane` class

FIGURE 2.5 The input dialog box enables the user to enter a string.



When this method is executed, a dialog is displayed to enable you to enter an input value. After entering a string, click *OK* to accept the input and dismiss the dialog box. The input is returned from the method as a string.

There are several ways to use the `showInputDialog` method. For the time being, you need to know only two ways to invoke it.

`showInputDialog` method

One is to use a statement like this one:

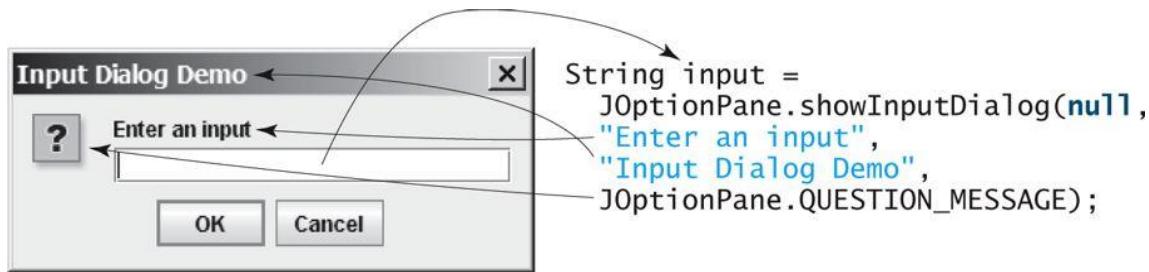
```
JOptionPane.showInputDialog(x);
```

where `x` is a string for the prompting message.

The other is to use a statement such as the following:

```
String string = JOptionPane.showInputDialog(null, x,  
y, JOptionPane.QUESTION_MESSAGE);
```

where `x` is a string for the prompting message and `y` is a string for the title of the input dialog box, as shown in the example below.



2.18.1 Converting Strings to Numbers

The input returned from the input dialog box is a string. If you enter a numeric value such as `123`, it returns `"123"`. You have to convert a string into a number to obtain the input as a number.

To convert a string into an `int` value, use the `parseInt` method in the `Integer` class, as follows:

`Integer.parseInt` method

```
int intValue = Integer.parseInt(intString);
```

where `intString` is a numeric string such as `"123"`.

To convert a string into a `double` value, use the `parseDouble` method in the `Double` class, as follows:

Double.parseDouble method

```
double doubleValue = Double.parseDouble(doubleString);
```

where `doubleString` is a numeric string such as `"123.45"`.

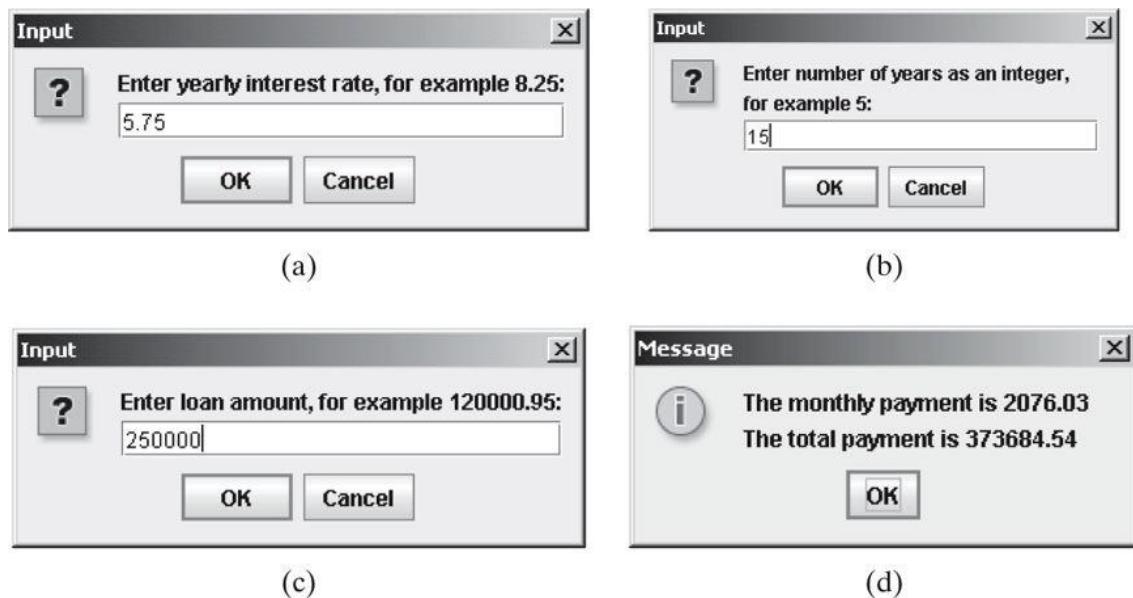
The `Integer` and `Double` classes are both included in the `java.lang` package, and thus they are automatically imported.

2.18.2 Using Input Dialog Boxes

[Listing 2.8](#), `ComputeLoan.java`, reads input from the console. Alternatively, you can use input dialog boxes.

[Listing 2.11](#) gives the complete program. [Figure 2.6](#) shows a sample run of the program.

FIGURE 2.6 The program accepts the annual interest rate (a), number of years (b), and loan amount (c), then displays the monthly payment and total payment (d).



LISTING 2.11 ComputeLoanUsingInputDialog.java

```

1 import javax.swing.JOptionPane;
2
3 public class ComputeLoanUsingInputDialog {
4     public static void main(String[] args) {
5         // Enter yearly interest rate
6         String annualInterestRateString = JOptionPane.showInputDialog(
7             "Enter yearly interest rate, for example 8.25:");
8
9         // Convert string to double
10        double annualInterestRate =
11            Double.parseDouble(annualInterestRateString);
12
13        // Obtain monthly interest rate
14        double monthlyInterestRate = annualInterestRate / 1200;
15
16        // Enter number of years
17        String numberofYearsString = JOptionPane.showInputDialog(
18            "Enter number of years as an integer, \nfor example 5:");
19
20        // Convert string to int
21        int numberOfYears = Integer.parseInt(numberofYearsString);
22
23        // Enter loan amount
24        String loanString = JOptionPane.showInputDialog(
25            "Enter loan amount, for example 120000.95:");
26
27        // Convert string to double
28        double loanAmount = Double.parseDouble(loanString);
29
30        // Calculate payment
31        double monthlyPayment = loanAmount * monthlyInterestRate / (1
32            - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
33        double totalPayment = monthlyPayment * numberOfYears * 12;           monthlyPayment
34
35        // Format to keep two digits after the decimal point
36        monthlyPayment = (int)(monthlyPayment * 100) / 100.0;                  preparing output
37        totalPayment = (int)(totalPayment * 100) / 100.0;
38
39        // Display results
40        String output = "The monthly payment is " + monthlyPayment +
41            "\nThe total payment is " + totalPayment;
42        JOptionPane.showMessageDialog(null, output);
43    }
44 }

```

The `showInputDialog` method in lines 6–7 displays an input dialog. Enter the interest rate as a double value and click *OK* to accept the input. The value is returned as a string that is assigned to the `String` variable `annualInterestRateString`. The `Double.parseDouble(annualInterestRateString)` (line 11) is used to convert the string into a `double` value. If you entered an input other than a numeric value or clicked *Cancel* in the input dialog box, a runtime error would occur. In Chapter 13, “Exception Handling,” you will learn how to handle the exception so that the program can continue to run.



Pedagogical Note

For obtaining input you can use **JOptionPane** or **Scanner**, whichever is convenient. For consistency most examples in this book use **Scanner** for getting input. You can easily revise the examples using **JOptionPane** for getting input.

JOptionPane or **Scanner**?

KEY TERMS

algorithm [24](#)

assignment operator (`=`) [30](#)

assignment statement [30](#)

backslash (`\`) [46](#)

byte type [27](#)

casting [41](#)

char type [44](#)

constant [31](#)

data type [25](#)

debugger [55](#)

debugging [55](#)

declaration [30](#)

decrement operator (`--`) [41](#)

double type [33](#)

encoding [45](#)

final [31](#)

float type [35](#)

floating-point number [33](#)

expression [31](#)

identifier [29](#)

increment operator (`++`) [41](#)

incremental development and testing [26](#)

indentation [52](#)

`int` type [34](#)

literal [35](#)

logic error [54](#)

`long` type [35](#)

narrowing (of types) [41](#)

operator [33](#)

overflow [33](#)

pseudocode [30](#)

primitive data type [25](#)

runtime error [54](#)

`short` type [27](#)

syntax error [53](#)

supplementary Unicode [45](#)

underflow [33](#)

Unicode [45](#)

Unix epoch [43](#)

variable [24](#)

widening (of types) [41](#)

whitespace [51](#)

CHAPTER SUMMARY

1. Identifiers are names for things in a program.
2. An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs (\$).
3. An identifier must start with a letter or an underscore. It cannot start with a digit.
4. An identifier cannot be a reserved word.
5. An identifier can be of any length.
6. Choosing descriptive identifiers can make programs easy to read.
7. Variables are used to store data in a program
8. To declare a variable is to tell the compiler what type of data a variable can hold.
9. By convention, variable names are in lowercase.
10. In Java, the equal sign (=) is used as the *assignment operator*.
11. A variable declared in a method must be assigned a value before it can be used.
12. A *named constant* (or simply a *constant*) represents permanent data that never changes.
13. A named constant is declared by using the keyword `final`.
14. By convention, constants are named in uppercase.
15. Java provides four integer types (`byte`, `short`, `int`, `long`) that represent integers of four different sizes.
16. Java provides two floating-point types (`float`, `double`) that represent floating-point numbers of two different precisions.
17. Java provides operators that perform numeric operations: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (remainder).
18. Integer arithmetic (`/`) yields an integer result.

- 19.** The numeric operators in a Java expression are applied the same way as in an arithmetic expression.
- 20.** Java provides shorthand operators `+=` (addition assignment), `-=` (subtraction assignment), `*=` (multiplication assignment), `/=` (division assignment), and `%=` (remainder assignment).
- 21.** The increment operator (`++`) and the decrement operator (`--`) increment or decrement a variable by `1`.
- 22.** When evaluating an expression with values of mixed types, Java automatically converts the operands to appropriate types.
- 23.** You can explicitly convert a value from one type to another using the `(type) exp` notation.
- 24.** Casting a variable of a type with a small range to a variable of a type with a larger range is known as *widening a type*.
- 25.** Casting a variable of a type with a large range to a variable of a type with a smaller range is known as *narrowing a type*.
- 26.** Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly.
- 27.** Character type (`char`) represents a single character.
- 28.** The character `\` is called the escape character.
- 29.** Java allows you to use escape sequences to represent special characters such as `\t` and `\n`.
- 30.** The characters `& &`, `\t`, `\f`, `\r`, and `\n` are known as the whitespace characters.
- 31.** In computer science, midnight of January 1, 1970, is known as the *Unix epoch*.
- 32.** Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors.
- 33.** Errors that occur during compilation are called *syntax errors* or *compile errors*.
- 34.** *Runtime errors* are errors that cause a program to terminate abnormally.

- 35.** *Logic errors* occur when a program does not perform the way it was intended to.

REVIEW QUESTIONS

Sections 2.2–2.7

- 2.1** Which of the following identifiers are valid? Which are Java keywords?

`applet, Applet, a++, --a, 4# R, $4, #44, apps
class, public, int, x, y, radius`

- 2.2** Translate the following algorithm into Java code:

- Step 1: Declare a `double` variable named `miles` with initial value `100`;
- Step 2: Declare a `double` constant named `MILES_PER_KILOMETER` with value `1.609`;
- Step 3: Declare a `double` variable named `kilometers`, multiply miles and `MILES_PER_KILOMETER`, and assign the result to `kilometers`.
- Step 4: Display `kilometers` to the console.

What is `kilometers` after Step 4?

- 2.3** What are the benefits of using constants? Declare an `int` constant `SIZE` with value `20`.

Sections 2.8–2.10

- 2.4** Assume that `int a = 1` and `double d = 1.0`, and that each expression is independent. What are the results of the following expressions?

```
a = 46 / 9;  
a = 46 % 9 + 4 * 4 - 2;  
a = 45 + 43 % 5 * (23 * 3 % 2);  
a %= 3 / a + 3;  
d = 4 + d * d + 4;  
d += 1.5 * 3 + (++a);  
d -= 1.5 * 3 + a++;
```

- 2.5** Show the result of the following remainders.

`56 % 6`
`78 % -4`
`-34 % 5`

```
-34 % -5  
5 % 1  
1 % 5
```

2.6 If today is Tuesday, what will be the day in 100 days?

2.7 Find the largest and smallest **byte**, **short**, **int**, **long**, **float**, and **double**. Which of these data types requires the least amount of memory?

2.8 What is the result of **25 / 4**? How would you rewrite the expression if you wished the result to be a floating-point number?

2.9 Are the following statements correct? If so, show the output.

```
System.out.println("25 / 4 is " + 25 / 4);  
System.out.println("25 / 4.0 is " + 25 / 4.0);  
System.out.println("3 * 2 / 4 is " + 3 *2 / 4);  
System.out.println("3.0 * 2 / 4 is " + 3.0 *2 / 4);
```

2.10 How would you write the following arithmetic expression in Java?

$$\frac{4}{3(r+34)} - 9(a+b c) + \frac{3 + d(2 + a)}{a + b d}$$

2.11 Suppose **m** and **r** are integers. Write a Java expression for **mr²** to obtain a floating-point result.

2.12 Which of these statements are true?

- (a) Any expression can be used as a statement.
- (b) The expression **x++** can be used as a statement.
- (c) The statement **x = x + 5** is also an expression.
- (d) The statement **x = y = x = 0** is illegal.

2.13 Which of the following are correct literals for floating-point numbers?

12.3, 12.3e+2, 23.4e-2, -334.4, 20, 39F, 40D

2.14 Identify and fix the errors in the following code:

```
1 public class Test {  
2     public void main(string[] args) {  
3         int i;  
4         int k = 100.0;
```

```
5     int j = i + 1;
6
7     System.out.println("j is " + j + " and
8         k is " + k);
9     }
10 }
```

- 2.15** How do you obtain the current minute using the `System.currentTimeMillis()` method?

Section 2.11

- 2.16** Can different types of numeric values be used together in a computation?

- 2.17** What does an explicit conversion from a `double` to an `int` do with the fractional part of the `double` value? Does casting change the variable being cast?

- 2.18** Show the following output.

```
float f = 12.5F;
int i = (int)f;
System.out.println("f is " + f);
System.out.println("i is " + i);
```

Section 2.13

- 2.19** Use print statements to find out the ASCII code for `&1&`, `&A&`, `&B&`, `&a&`, `&b&`. Use print statements to find out the character for the decimal code `40`, `59`, `79`, `85`, `90`. Use print statements to find out the character for the hexadecimal code `40`, `5A`, `71`, `72`, `7A`.

- 2.20** Which of the following are correct literals for characters?

`&1&`, `&\u345dE&`, `&\u3fFa&`, `&\b&`, `\t`

- 2.21** How do you display characters `\` and `"`?

- 2.22** Evaluate the following:

```
int i = &1&;
int j = &1& + &2&;
int k = &a&;
char c = 90;
```

- 2.23** Can the following conversions involving casting be allowed? If so, find the converted result.

```

char c = &A&;
i = (int)c;
float f = 1000.34f;
int i = (int)f;
double d = 1000.34;
int i = (int)d;
int i = 97;
char c = (char)i;

```

2.24 Show the output of the following program:

```

public class Test {
    public static void main(String[] args) {
        char x = &a&;
        char y = &c&;
        System.out.println(++x);
        System.out.println(y++);
        System.out.println(x - y);
    }
}

```

Section 2.15

2.25 Show the output of the following statements (write a program to verify your result):

```

System.out.println("1" + 1);
System.out.println(&1& + 1);
System.out.println("1" + 1 + 1);
System.out.println("1" + (1 + 1));
System.out.println(&1& + 1 + 1);

```

2.26 Evaluate the following expressions (write a program to verify your result):

```

1 + "Welcome" + 1 + 1
1 + "Welcome" + (1 + 1)
1 + "Welcome" + (&\u0001& + 1)
1 + "Welcome" + &a& + 1

```

Sections 2.16–2.17

2.27 What are the naming conventions for class names, method names, constants, and variables? Which of the following items can be a constant, a method, a variable, or a class according to the Java naming conventions?

MAX_VALUE, Test, read, readInt

2.28 Reformat the following program according to the programming style and documentation guidelines. Use the next-line brace style.

```
public class Test
{
    // Main method
    public static void main(String[] args) {
        /** Print a line */
        System.out.println("2 % 3 = "+2%3);
    }
}
```

2.29 Describe syntax errors, runtime errors, and logic errors.

Section 2.18

2.30 Why do you have to import `JOptionPane` but not the `Math` class?

2.31 How do you prompt the user to enter an input using a dialog box?

2.32 How do you convert a string to an integer? How do you convert a string to a double?

PROGRAMMING EXERCISES



Note

Students can run all exercises by downloading `exercise8e.zip` from www.cs.armstrong.edu/liang/intro8e/exercise8e.zip and use the command `java -cp exercise8e.zip Exercisei_j` to run Exercise *i_j*. For example, to run Exercise2_1, use

```
java -cp exercise8e.zip Exercise2_1
```

This will give you an idea how the program runs.

sample runs



Debugging TIP

The compiler usually gives a reason for a syntax error. If you don't know how to correct it, compare your program closely, character by character, with similar examples in the text.

learn from examples

Sections 2.2–2.9

2.1 (Converting Celsius to Fahrenheit) Write a program that reads a Celsius degree in double from the console, then converts it to Fahrenheit and displays the result. The formula for the conversion is as follows:

```
fahrenheit = (9 / 5) * celsius + 32
```

Hint: In Java, `9/5` is `1`, but `9.0 / 5` is `1.8`.

Here is a sample run:



Enter a degree in Celsius: **43**

43 Celsius is 109.4 Fahrenheit

2.2 (Computing the volume of a cylinder) Write a program that reads in the radius and length of a cylinder and computes volume using the following formulas:

```
area = radius * radius * π  
volume = area * length
```

Here is a sample run:



Enter the radius and length of a cylinder:

5.5 12

The area is 95.0331

The volume is 1140.4

2.3 (*Converting feet into meters*) Write a program that reads a number in feet, converts it to meters, and displays the result. One foot is **0.305** meter. Here is a sample run:



Enter a value for feet: **16**

16 feet is 4.88 meters

2.4 (*Converting pounds into kilograms*) Write a program that converts pounds into kilograms. The program prompts the user to enter a number in pounds, converts it to kilograms, and displays the result. One pound is **0.454** kilograms. Here is a sample run:



Enter a number in pounds: **55.5**

55.5 pounds is 25.197 kilograms

2.5* (*Financial application: calculating tips*) Write a program that reads the subtotal and the gratuity rate, then computes the gratuity and total. For example, if the user enters **10** for subtotal and **15%** for gratuity rate, the program displays **\$1.5** as gratuity and **\$11.5** as total. Here is a sample run:



Enter the subtotal and a gratuity rate:

15.69 15

The gratuity is 2.35 and total is 18.04

2.6** (*Summing the digits in an integer*) Write a program that reads an integer between **0** and **1000** and adds all the digits in the integer. For example, if an integer is **932**, the sum of all its digits is **14**.

Hint: Use the `%` operator to extract digits, and use the `/` operator to remove the extracted digit. For instance, **932 % 10 = 2** and **932 / 10 = 93**.

Here is a sample run:



999

Enter a number between 0 and 1000:

The sum of the digits is 27

2.7* (*Finding the number of years*) Write a program that prompts the user to enter the minutes (e.g., 1 billion) and displays the number of years and days for the minutes. For simplicity, assume a year has **365** days. Here is a sample run:



Enter the number of minutes:

1000000000

1000000000 minutes is approximately 1902 years and 214 days.

Section 2.13

2.8* (*Finding the character of an ASCII code*) Write a program that receives an ASCII code (an integer between **0** and **128**) and displays its character. For example, if the user enters **97**, the program displays character **a**. Here is a sample run:



Enter an ASCII code:

69

Enter

The character for ASCII code 69 is E

2.9* (*Financial application: monetary units*) Rewrite [Listing 2.10](#), ComputeChange.java, to fix the possible loss of accuracy when converting a `double` value to an `int` value. Enter the input as an integer whose last two digits represent the cents. For example, the input `1156` represents `11` dollars and `56` cents.

Section 2.18

2.10* (*Using the GUI input*) Rewrite [Listing 2.10](#), ComputeChange.java, using the GUI input and output.

Comprehensive

2.11* (*Financial application: payroll*) Write a program that reads the following information and prints a payroll statement:

Employee's name (e.g., Smith)

Number of hours worked in a week (e.g., 10)

Hourly pay rate (e.g., 6.75)

Federal tax withholding rate (e.g., 20%)

State tax withholding rate (e.g., 9%)

Write this program in two versions: (a) Use dialog boxes to obtain input and display output; (b) Use console input and output. A sample run of the console input and output is shown below:



Enter employee's name:

Smith

Enter

10

← Enter

Enter number of hours worked in a week:

6.75

← Enter

Enter hourly pay rate:

0.20

← Enter

Enter federal tax withholding rate:

0.09

← Enter

Enter state tax withholding rate:

Employee Name: Smith

Hours Worked: 10.0

Pay Rate: \$6.75

Gross Pay: \$67.5

Deductions:

Federal Withholding (20.0%): \$13.5

State Withholding (9.0%): \$6.07

Total Deduction: \$19.57

Net Pay: \$47.92

2.12* (*Financial application: calculating interest*) If you know the balance and the annual percentage interest rate, you can compute the interest on the next monthly payment using the following formula:

$$\text{interest} = \text{balance} \times (\text{annualInterestRate} / 1200)$$

Write a program that reads the balance and the annual percentage interest rate and displays the interest for the next month in two versions: (a) Use dialog boxes to obtain input and display output; (b) Use console input and output. Here is a sample run:



Enter balance and interest rate (e.g., 3 for 3%):

The interest is 2.91667

2.13* (*Financial application: calculating the future investment value*) Write a program that reads in investment amount, annual interest rate, and number of years, and displays the future investment value using the following formula:

```
futureInvestmentValue =  
    investmentAmount * (1 +  
        monthlyInterestRate)numberOfYears*12
```

For example, if you enter amount **1000**, annual interest rate **3.25%**, and number of years **1**, the future investment value is **1032.98**.

Hint: Use the **Math.pow(a, b)** method to compute **a** raised to the power of **b**. Here is a sample run:



Enter investment amount:

Enter monthly interest rate:

Enter number of years:

Accumulated value is 1043.34

2.14* (*Health application: computing BMI*) Body Mass Index (BMI) is a measure of health on weight. It can be calculated by taking your weight in kilograms and dividing by the square of your height in meters. Write a program that prompts the user to enter a weight in pounds and height in inches and display the BMI. Note that one pound is **0.45359237** kilograms and one inch is **0.0254** meters. Here is a sample run:



Video Note

Compute BMI



Enter weight in pounds: **95.5**

Enter height in inches: **50**

BMI is 26.8573

2.15** (*Financial application: compound value*) Suppose you save \$100 each month into a savings account with the annual interest rate 5%. So, the monthly interest rate is $0.05 / 12 = 0.00417$. After the first month, the value in the account becomes

$$100 * (1 + 0.00417) = 100.417$$

After the second month, the value in the account becomes

$$(100 + 100.417) * (1 + 0.00417) = 201.252$$

After the third month, the value in the account becomes

$$(100 + 201.252) * (1 + 0.00417) = 302.507$$

and so on.

Write a program to display the account value after the sixth month. (In Exercise 4.30, you will use a loop to simplify the code and display the account value for any month.)

2.16 (*Science: calculating energy*) Write a program that calculates the energy needed to heat water from an initial temperature to a final temperature. Your program

should prompt the user to enter the amount of water in kilograms and the initial and final temperatures of the water. The formula to compute the energy is

$$Q = M * (\text{final temperature} - \text{initial temperature}) * 4184$$

where **M** is the weight of water in kilograms, temperatures are in degrees Celsius, and energy **Q** is measured in joules. Here is a sample run;



Enter the amount of water in kilograms:

Enter the initial temperature:

Enter the final temperature:

The energy needed is 1.62548e+06

2.17* (*Science: wind-chill temperature*) How cold is it outside? The temperature alone is not enough to provide the answer. Other factors including wind speed, relative humidity, and sunshine play important roles in determining coldness outside. In 2001, the National Weather Service (NWS) implemented the new wind-chill temperature to measure the coldness using temperature and wind speed. The formula is given as follows:

$$t_{wc} = 35.74 + 0.6215t_a - 35.75v^{0.16} + 0.4275t_av^{0.16}$$

where t_a is the outside temperature measured in degrees Fahrenheit and v is the speed measured in miles per hour. t_{wc} is the wind-chill temperature. The formula cannot be used for wind speeds below 2 mph or temperatures below -58°F or above 41°F .

Write a program that prompts the user to enter a temperature between -58°F and 41°F and a wind speed greater than or equal to 2 and displays the wind-chill temperature. Use **Math.pow(a, b)** to compute $v^{0.16}$. Here is a sample run:



5 . 3

← Enter

Enter the temperature in Fahrenheit:

6

← Enter

Enter the wind speed miles per hour:

The wind chill index is -5.56707

2.18 (*Printing a table*) Write a program that displays the following table:

```
a  
b  
pow(a, b)  
1  
2  
1  
2  
3  
8  
3  
4  
81  
4  
5  
1024  
5  
6
```

2.19 (*Random character*) Write a program that displays a random uppercase letter using the `System.currentTimeMillis()` method.

2.20 (*Geometry: distance of two points*) Write a program that prompts the user to enter two points `(x1, y1)` and `(x2, y2)` and displays their distances. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Note you can use the `Math.pow(a, 0.5)` to compute \sqrt{a} . Here is a sample run:



Enter x1 and y1: **1.5 -3.4 ↵ Enter**

Enter x2 and y2: **4 5 ↵ Enter**

The distance of the two points is 8.764131445842194

2.21* (*Geometry: area of a triangle*) Write a program that prompts the user to enter three points `(x1, y1)`, `(x2, y2)`, `(x3, y3)` of a triangle and displays its area. The formula for computing the area of a triangle is

$$s = (\text{side 1} + \text{side 2} + \text{side 3}) / 2;$$

$$\text{area} = \sqrt{s(s - \text{side 1})(s - \text{side 2})(s - \text{side 3})}$$

Here is a sample run.



Enter three points for a triangle:

1.5 -3.4 4.6 5 9.5 -3.4 ↵ Enter

The area of the triangle is 33.6

2.22 (*Geometry: area of a hexagon*) Write a program that prompts the user to enter the side of a hexagon and displays its area. The formula for computing the area of a hexagon is

$$\text{Area} = \frac{3\sqrt{3}}{2}s^2,$$

where s is the length of a side. Here is a sample run:



Enter the side:

The area of the hexagon is 78.5895

2.23 (*Physics: acceleration*) Average acceleration is defined as the change of velocity divided by the time taken to make the change, as shown in the following formula:

$$a = \frac{v_1 - v_0}{t}$$

Write a program that prompts the user to enter the starting velocity in meters/second, the ending velocity v_1 in meters/second, and the time span t in seconds, and displays the average acceleration. Here is a sample run:



Enter v0, v1, and t:

The average acceleration is 10.0889

2.24 (*Physics: finding runway length*) Given an airplane's acceleration a and take-off speed v , you can compute the minimum runway length needed for an airplane to take off using the following formula:

$$\text{length} = \frac{v^2}{2a}$$

Write a program that prompts the user to enter v in meters/second (m/s) and the acceleration a in meters/second squared (m/s^2), and displays the minimum runway length. Here is a sample run:



Enter v and a :

| | |
|--------|---------|
| 60 3.5 | ← Enter |
|--------|---------|

The minimum runway length for this airplane is 514.286

2.25* (*Current time*) [Listing 2.6](#), ShowCurrentTime.java, gives a program that displays the current time in GMT. Revise the program so that it prompts the user to enter the time zone offset to GMT and displays the time in the specified time zone. Here is a sample run:



Enter the time zone offset to GMT:

| | |
|----|---------|
| -5 | ← Enter |
|----|---------|

The current time is 4:50:34

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 36).
<vbk:9781256335153#outline(6.9.5)>

CHAPTER 3 SELECTIONS

Objectives

- To declare `boolean` type and write Boolean expressions using comparison operators ([§3.2](#)).
- To program `AdditionQuiz` using Boolean expressions ([§3.3](#)).
- To implement selection control using one-way `if` statements ([§3.4](#))
- To program the `GuessBirthday` game using one-way if statements ([§3.5](#)).
- To implement selection control using two-way `if` statements ([§3.6](#)).
- To implement selection control using nested `if` statements ([§3.7](#)).
- To avoid common errors in `if` statements ([§3.8](#)).
- To program using selection statements for a variety of examples (`SubtractionQuiz`, `BMI`, `ComputeTax`) ([§3.9–3.11](#)).
- To generate random numbers using the `Math.random()` method ([§3.9](#)).
- To combine conditions using logical operators (`&&`, `||<`, and `!`) ([§3.12](#)).
- To program using selection statements with combined conditions (`LeapYear`, `Lottery`) ([§§3.13–3.14](#)).
- To implement selection control using `switch` statements ([§3.15](#)).
- To write expressions using the conditional operator ([§3.16](#)).
- To format output using the `System.out.printf` method and to format strings using the `String.format` method ([§3.17](#)).
- To examine the rules governing operator precedence and associativity ([§3.18](#)).
- (GUI) To get user confirmation using confirmation dialogs ([§3.19](#)).

3.1 Introduction

If you enter a negative value for `radius` in [Listing 2.2](#), `ComputeAreaWithConsoleInput.java`, the program prints an invalid result. If the radius is negative, you don't want the program to compute the area. How can you deal with this situation?

problem

Like all high-level programming languages, Java provides selection statements that let you choose actions with two or more alternative courses. You can use the following selection statement to replace lines 12–17 in [Listing 2.2](#):

```
if (radius < 0)
    System.out.println("Incorrect input");
else {
    area = radius * radius * 3.14159;
    System.out.println("Area is " + area);
}
```

Selection statements use conditions. Conditions are Boolean expressions. This chapter first introduces Boolean types, values, comparison operators, and expressions.

3.2 boolean Data Type

How do you compare two values, such as whether a radius is greater than `0`, equal to `0`, or less than `0`? Java provides six *comparison operators* (also known as *relational operators*), shown in [Table 3.1](#), which can be used to compare two values (assume radius is `5` in the table).

comparison operators

TABLE 3.1 Comparison Operators

Operator

Name

Example

Result

<

less than

radius < 0

false

<=

less than or equal to

radius <= 0

false

>

greater than

radius > 0

true

>=

greater than or equal to

radius >= 0

true

==

equal to

radius == 0

false

!=

not equal to

radius!= 0

true



Note

You can also compare characters. Comparing characters is the same as comparing their Unicodes. For example, `a` is larger than `A` because the Unicode of `a` is larger than the Unicode of `A`. See [Appendix B](#), “The ASCII Character Sets,” to find the order of characters.

compare characters



Caution

The equality comparison operator is two equal signs (`==`), not a single equal sign (`=`). The latter symbol is for assignment.

`==` vs. `=`

The result of the comparison is a Boolean value: `true` or `false`. For example, the following statement displays `true`:

```
double radius = 1;  
System.out.println(radius > 0);
```

Boolean variable

A variable that holds a Boolean value is known as a *Boolean variable*. The `boolean` data type is used to declare Boolean variables. A `boolean` variable can hold one of the two values: `true` and `false`. For example, the following statement assigns `true` to the variable `lightsOn`:

```
boolean lightsOn = true;
```

`true` and `false` are literals, just like a number such as `10`. They are reserved words and cannot be used as identifiers in your program.

Boolean literals

3.3 Problem: A Simple Math Learning Tool

Suppose you want to develop a program to let a first-grader practice addition. The program randomly generates two single-digit integers, `number1` and `number2`, and displays to the

student a question such as “What is $7 + 9$ ”, as shown in the sample run. After the student types the answer, the program displays a message to indicate whether it is true or false.



Video Note

Program addition quiz

There are several ways to generate random numbers. For now, generate the first integer using `System.currentTimeMillis() % 10` and the second using `System.currentTimeMillis() * 7 % 10`. [Listing 3.1](#) gives the program. Lines 5–6 generate two numbers, `number1` and `number2`. Line 14 obtains an answer from the user. The answer is graded in line 18 using a Boolean expression `number1 + number2 == answer`.

LISTING 3.1 AdditionQuiz.java

```
1 import java.util.Scanner;
2
3 public class AdditionQuiz {
4     public static void main(String[] args) {
5         int number1 = (int)(System.currentTimeMillis() % 10);           generate number1
6         int number2 = (int)(System.currentTimeMillis() * 7 % 10);           generate number2
7
8         // Create a Scanner
9         Scanner input = new Scanner(System.in);
10
11        System.out.print(                                         show question
12            "What is " + number1 + " + " + number2 + "? ");
13
14        int answer = input.nextInt();
15
16        System.out.println(                                         display result
17            number1 + " + " + number2 + " = " + answer + " is "
18            (number1 + number2 == answer));
19    }
20 }
```



What is $1 + 7$? 8

$1 + 7 = 8$ is true



What is $4 + 8$? **9**

$4 + 8 = 9$ is false



line

number1

number2

answer

output

5

4

6

8

14

$4 + 8 = 9$ is false

3.4 if Statements

The preceding program displays a message such as “ $6 + 2 = 7$ is false.” If you wish the message to be “ $6 + 2 = 7$ is incorrect,” you have to use a selection statement to carry out this minor change.

why **if** statement?

This section introduces selection statements. Java has several types of selection statements: one-way **if** statements, two-way **if** statements, nested **if** statements, **switch** statements, and conditional expressions.

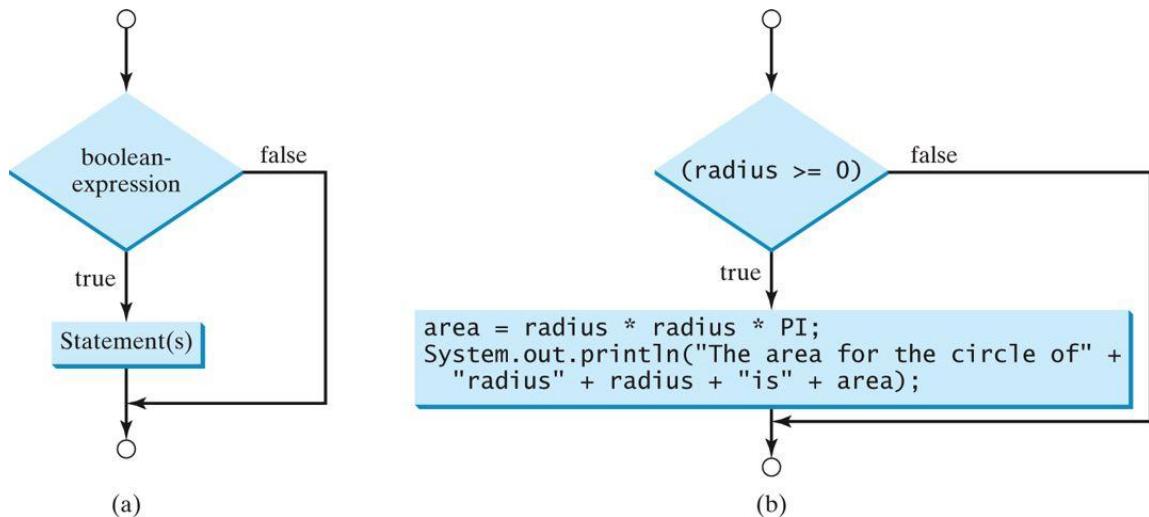
3.4.1 One-Way if Statements

A one-way **if** statement executes an action if and only if the condition is **true**. The syntax for a one-way **if** statement is shown below:

```
if statement
if (boolean-expression) {
    statement(s);
}
```

The execution flow chart is shown in [Figure 3.1\(a\)](#).

FIGURE 3.1 An if statement executes statements if the boolean-expression evaluates to true.



If the **boolean-expression** evaluates to **true**, the statements in the block are executed. As an example, see the following code:

```

if (radius >= 0) {
    area = radius * radius * PI;
    System.out.println("The area for the circle of radius
" +
    radius + " is " + area);
}
  
```

The flow chart of the preceding statement is shown in [Figure 3.1\(b\)](#). If the value of **radius** is greater than or equal to **0**, then the **area** is computed and the result is displayed; otherwise, the two statements in the block will not be executed.

The **boolean-expression** is enclosed in parentheses. For example, the code in (a) below is wrong. It should be corrected, as shown in (b).

| | |
|--|--|
| <pre> if i > 0 { System.out.println("i is positive"); } </pre> | <pre> if (i > 0) { System.out.println("i is positive"); } </pre> |
|--|--|

(a) Wrong
(b) Correct

The block braces can be omitted if they enclose a single statement. For example, the following statements are equivalent.

<pre> if (i > 0) { System.out.println("i is positive"); } </pre>	<u>Equivalent</u>	<pre> if (i > 0) System.out.println("i is positive"); </pre>
--	-------------------	--

(a)

(b)

[Listing 3.2](#) gives a program that prompts the user to enter an integer. If the number is a multiple of 5, print **HiFive**. If the number is divisible by 2, print **HiEven**.

LISTING 3.2 SimpleIfDemo.java

```
1 import java.util.Scanner;  
2  
3 public class SimpleIfDemo {  
4     public static void main(String[] args) {  
5         Scanner input = new Scanner(System.in);  
6         System.out.println("Enter an integer: ");  
7         int number = input.nextInt();  
8  
9         if (number % 5 == 0)  
10             System.out.println("HiFive");  
11  
12         if (number % 2 == 0)  
13             System.out.println("HiEven");  
14     }  
15 }
```

enter input
check 5
check even



Enter an integer: 4 

HiEven



Enter an integer: 30 

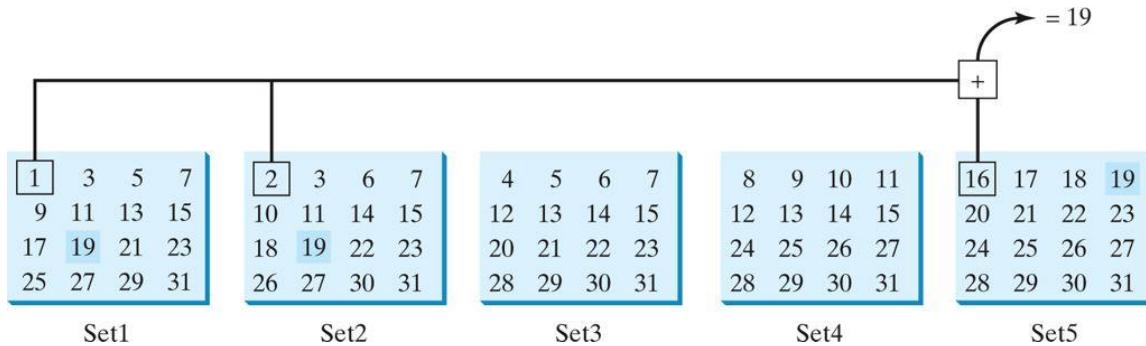
HiFive

HiEven

The program prompts the user to enter an integer (line 7) and displays **HiFive** if it is divisible by 5 (lines 9–10) and **HiEven** if it is divisible by 2 (lines 12–13).

3.5 Problem: Guessing Birthdays

You can find out the date of the month when your friend was born by asking five questions. Each question asks whether the day is in one of the five sets of numbers.



The birthday is the sum of the first numbers in the sets where the day appears. For example, if the birthday is **19**, it appears in Set1, Set2, and Set5. The first numbers in these three sets are **1**, **2**, and **16**. Their sum is **19**.

[Listing 3.3](#) gives a program that prompts the user to answer whether the day is in Set1 (lines 41–47), in Set2 (lines 50–56), in Set3 (lines 59–65), in Set4 (lines 68–74), and in Set5 (lines 77–83). If the number is in the set, the program adds the first number in the set to **day** (lines 47, 56, 65, 74, 83).

LISTING 3.3 GuessBirthday.java

```

1 import java.util.Scanner;
2
3 public class GuessBirthday {
4     public static void main(String[] args) {
5         String set1 =
6             " 1 3 5 7\n" +
7             " 9 11 13 15\n" +
8             "17 19 21 23\n" +
9             "25 27 29 31";
10
11        String set2 =
12            " 2 3 6 7\n" +
13            "10 11 14 15\n" +
14            "18 19 22 23\n" +
15            "26 27 30 31";
16
17        String set3 =
18            " 4 5 6 7\n" +
19            "12 13 14 15\n" +
20            "20 21 22 23\n" +
21            "28 29 30 31";
22
23        String set4 =
24            " 8 9 10 11\n" +
25            "12 13 14 15\n" +
26            "24 25 26 27\n" +
27            "28 29 30 31";
28
29        String set5 =
30            "16 17 18 19\n" +
31            "20 21 22 23\n" +
32            "24 25 26 27\n" +
33            "28 29 30 31";
34
35    int day = 0;
36
37    // Create a Scanner
38    Scanner input = new Scanner(System.in);
39
40    // Prompt the user to answer questions
41    System.out.print("Is your birthday in Set1?\n");
42    System.out.print(set1);
43    System.out.print("\nEnter 0 for No and 1 for Yes: ");
44    int answer = input.nextInt();
45
46    if (answer == 1)
47        day += 1;
48

```

day to be determined

in Set1?

```

49 // Prompt the user to answer questions
50 System.out.print("\nIs your birthday in Set2?\n");
51 System.out.print(set2);
52 System.out.print("\nEnter 0 for No and 1 for Yes: ");
53 answer = input.nextInt();
54
55 if (answer == 1)                                in Set2?
56     day += 2;
57
58 // Prompt the user to answer questions
59 System.out.print("Is your birthday in Set3?\n");
60 System.out.print(set3);
61 System.out.print("\nEnter 0 for No and 1 for Yes: ");
62 answer = input.nextInt();
63
64 if (answer == 1)                                in Set3?
65     day += 4;
66
67 // Prompt the user to answer questions
68 System.out.print("\nIs your birthday in Set4?\n");
69 System.out.print(set4);
70 System.out.print("\nEnter 0 for No and 1 for Yes: ");
71 answer = input.nextInt();
72
73 if (answer == 1)                                in Set4?
74     day += 8;
75
76 // Prompt the user to answer questions
77 System.out.print("\nIs your birthday in Set5?\n");
78 System.out.print(set5);
79 System.out.print("\nEnter 0 for No and 1 for Yes: ");
80 answer = input.nextInt();
81
82 if (answer == 1)                                in Set5?
83     day += 16;
84
85 System.out.println("\nYour birthday is " + day + "!");
86 }
87 }

```



Is your birthday in Set1?

1 3 5 7

9 11 13 15

17 19 21 23

25 27 29 31



Enter 0 for No and 1 for Yes: 1

Is your birthday in Set2?

2 3 6 7

10 11 14 15

18 19 22 23

26 27 30 31



Enter 0 for No and 1 for Yes: 1

Is your birthday in Set3?

4 5 6 7

12 13 14 15

20 21 22 23

28 29 30 31



Enter 0 for No and 1 for Yes: 0

Is your birthday in Set4?

8 9 10 11

12 13 14 15

24 25 26 27

28 29 30 31



Enter 0 for No and 1 for Yes: 0

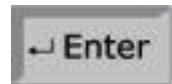
Is your birthday in Set5?

16 17 18 19

20 21 22 23

24 25 26 27

28 29 30 31



Enter 0 for No and 1 for Yes: 1

Your birthday is 19



line #

day

answer

output

35

0

44

1

47

1

53

1

56

3

62

0

71

0

80

1

83

19

Your birthday is 19

The game is easy to program. You may wonder how the game was created. The mathematics behind the game is actually quite simple. The numbers are not grouped together by accident. The way they are placed in the five sets is deliberate. The starting numbers in the five sets are **1, 2, 4, 8, and 16**, which correspond to **1, 10, 100, 1000, and 10000** in binary. A binary number for decimal integers between **1** and **31** has at most five digits, as shown in [Figure 3.2\(a\)](#). Let it be $b_5 b_4 b_3 b_2 b_1$. So, $b_5 b_4 b_3 b_2 b_1 = b_5 \cdot 0000 + b_4 \cdot 000 + b_3 \cdot 00 + b_2 \cdot 0 + b_1$, as shown in [Figure 3.2\(b\)](#). If a day's binary number has a digit **1** in b_k , the number should appear in Set k . For example, number **19** is binary **10011**, so it appears in Set1, Set2, and Set5. It is binary **1 + 10 + 10000 = 10011** or decimal **1 + 2 + 16 = 19**. Number **31** is binary **11111**, so it appears in Set1, Set2, Set3, Set4, and Set5. It is binary **1 + 10 + 100 + 1000 + 10000 = 11111** or decimal **1 + 2 + 4 + 8 + 16 = 31**.

mathematics behind the game

FIGURE 3.2 (a) A number between 1 and 31 can be represented using a 5-digit binary number. (b) A 5-digit binary number can be obtained by adding binary numbers 1, 10, 100, 1000, or 10000.

Decimal	Binary
1	00001
2	00010
3	00011
...	
19	10011
...	
31	11111

(a)

b_5	0	0	0	0	10000
b_4	0	0	0		1000
b_3	0	0		10000	100
b_2	0			10	10
+				<u>$\frac{1}{10011}$</u>	<u>$\frac{1}{11111}$</u>
				19	31
				$b_5 b_4 b_3 b_2 b_1$	

(b)

3.6 Two-Way if Statements

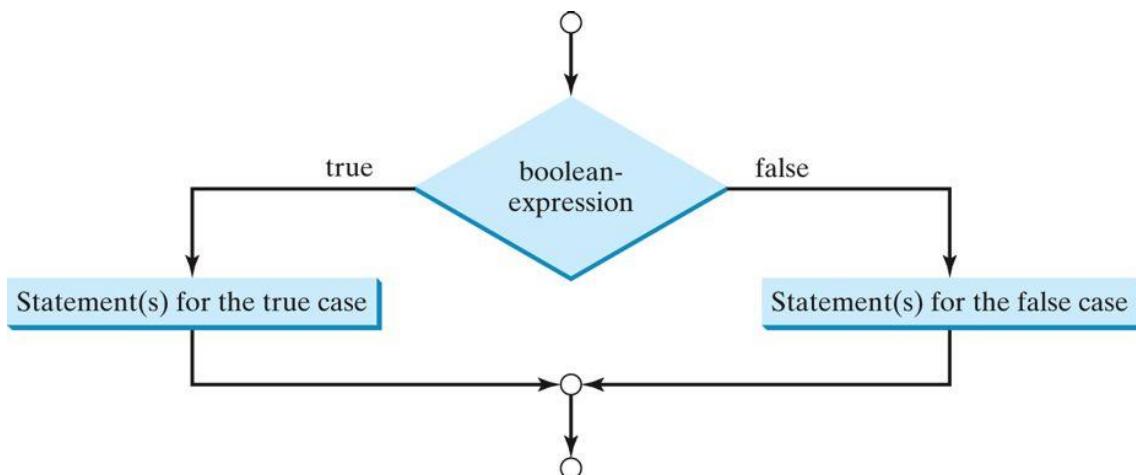
A one-way **if** statement takes an action if the specified condition is **true**. If the condition is **false**, nothing is done. But what if you want to take alternative actions when the condition is **false**? You can use a two-way **if** statement. The actions that a two-way **if** statement specifies differ based on whether the condition is **true** or **false**.

Here is the syntax for a two-way **if** statement:

```
if (boolean-expression) {  
    statement(s)-for-the-true-case;  
}  
else {  
    statement(s)-for-the-false-case;  
}
```

The flow chart of the statement is shown in [Figure 3.3](#).

FIGURE 3.3 An if... else statement executes statements for the true case if the boolean-expression evaluates to true; otherwise, statements for the false case are executed.



If the **boolean-expression** evaluates to **true**, the statement(s) for the true case are executed; otherwise, the statement(s) for the false case are executed. For example, consider the following code:

```
if (radius >= 0) {  
    area = radius * radius * PI;  
    System.out.println("The area for the circle of radius " +  
        radius + " is " + area);  
}  
else {  
    System.out.println("Negative input");  
}
```

two-way if statement

If `radius >= 0` is `true`, `area` is computed and displayed; if it is `false`, the message "`Negative input`" is printed.

As usual, the braces can be omitted if there is only one statement within them. The braces enclosing the `System.out.println("Negative input")` statement can therefore be omitted in the preceding example.

Here is another example of using the `if... else` statement. The example checks whether a number is even or odd, as follows:

```
if (number % 2 == 0)
    System.out.println(number + " is even.");
else
    System.out.println(number + " is odd.");
```

3.7 Nested if Statements

The statement in an `if` or `if... else` statement can be any legal Java statement, including another `if` or `if... else` statement. The inner `if` statement is said to be *nested* inside the outer `if` statement. The inner `if` statement can contain another `if` statement; in fact, there is no limit to the depth of the nesting. For example, the following is a nested `if` statement:

```
if (i > k) {
    if (j > k)
        System.out.println("i and j are greater than k");
}
else
    System.out.println("i is less than or equal to k");
```

The `if (j > k)` statement is nested inside the `if (i > k)` statement.

The nested `if` statement can be used to implement multiple alternatives. The statement given in [Figure 3.4\(a\)](#), for instance, assigns a letter grade to the variable `grade` according to the score, with multiple alternatives.

FIGURE 3.4 A preferred format for multiple alternative if statements is shown in (b).

```

if (score >= 90.0)
    grade = 'A';
else
    if (score >= 80.0)
        grade = 'B';
    else
        if (score >= 70.0)
            grade = 'C';
        else
            if (score >= 60.0)
                grade = 'D';
            else
                grade = 'F';

```

(a)

Equivalent

This is better

```

if (score >= 90.0)
    grade = 'A';
else if (score >= 80.0)
    grade = 'B';
else if (score >= 70.0)
    grade = 'C';
else if (score >= 60.0)
    grade = 'D';
else
    grade = 'F';

```

(b)

The execution of this `if` statement proceeds as follows. The first condition (`score >= 90.0`) is tested. If it is `true`, the grade becomes `& A &`. If it is `false`, the second condition (`score >= 80.0`) is tested. If the second condition is `true`, the grade becomes `& B &`. If that condition is `false`, the third condition and the rest of the conditions (if necessary) continue to be tested until a condition is met or all of the conditions prove to be `false`. If all of the conditions are `false`, the grade becomes `& F &`. Note that a condition is tested only when all of the conditions that come before it are `false`.

The `if` statement in [Figure 3.4\(a\)](#) is equivalent to the `if` statement in [Figure 3.4\(b\)](#). In fact, [Figure 3.4\(b\)](#) is the preferred writing style for multiple alternative `if` statements. This style avoids deep indentation and makes the program easy to read.



Tip

Often, to assign a test condition to a `boolean` variable, new programmers write code as in (a) below:

```

if (number % 2 == 0)
    even = true;
else
    even = false;

```

(a)

Equivalent

This is shorter

```

boolean even
= number % 2 == 0;

```

(b)

The code can be simplified by assigning the test value directly to the variable, as shown in (b).

assign **boolean** variable

3.8 Common Errors in Selection Statements

The following errors are common among new programmers.

Common Error 1: Forgetting Necessary Braces

The braces can be omitted if the block contains a single statement. However, forgetting the braces when they are needed for grouping multiple statements is a common programming error. If you modify the code by adding new statements in an **if** statement without braces, you will have to insert the braces. For example, the code in (a) below is wrong. It should be written with braces to group multiple statements, as shown in (b).

```
if (radius >= 0)
    area = radius * radius * PI;
    System.out.println("The area "
        + " is " + area);
```

(a) Wrong

```
if (radius >= 0) {
    area = radius * radius * PI;
    System.out.println("The area "
        + " is " + area);
}
```

(b) Correct

Common Error 2: Wrong Semicolon at the **if** Line

Adding a semicolon at the **if** line, as shown in (a) below, is a common mistake.

Logic Error

```
if (radius >= 0); {
    area = radius * radius * PI;
    System.out.println("The area "
        + " is " + area);
}
```

Equivalent

Empty Block

```
if (radius >= 0) {};
{
    area = radius * radius * PI;
    System.out.println("The area "
        + " is " + area);
}
```

(b)

This mistake is hard to find, because it is neither a compilation error nor a runtime error; it is a logic error. The code in (a) is equivalent to that in (b) with an empty block.

This error often occurs when you use the next-line block style. Using the end-of-line block style can help prevent the error.

Common Error 3: Redundant Testing of Boolean Values

To test whether a **boolean** variable is **true** or **false** in a test condition, it is redundant to use the equality comparison operator like the code in (a):

```
if (even == true)
    System.out.println(
        "It is even.");
```

(a)

Equivalent

```
if (even)
    System.out.println(
        "It is even.");
```

(b)

This is better

Instead, it is better to test the `boolean` variable directly, as shown in (b). Another good reason for doing this is to avoid errors that are difficult to detect. Using the `=` operator instead of the `==` operator to compare equality of two items in a test condition is a common error. It could lead to the following erroneous statement:

```
if (even = true)
    System.out.println("It is even.");
```

This statement does not have syntax errors. It assigns `true` to `even`, so that `even` is always `true`.

Common Error 4: Dangling `else` Ambiguity

The code in (a) below has two `if` clauses and one `else` clause. Which `if` clause is matched by the `else` clause? The indentation indicates that the `else` clause matches the first `if` clause. However, the `else` clause actually matches the second `if` clause. This situation is known as the *dangling-else ambiguity*. The `else` clause always matches the most recent unmatched `if` clause in the same block. So, the statement in (a) is equivalent to the code in (b).

```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
    if (i > k)
        System.out.println("A");
else
    System.out.println("B");
```

(a)

Equivalent

This is better
with correct
indentation

```
int i = 1;
int j = 2;
int k = 3;

if (i > j)
    if (i > k)
        System.out.println("A");
else
    System.out.println("B");
```

(b)

Since `(i > j)` is false, nothing is printed from the statement in (a) and (b). To force the `else` clause to match the first `if` clause, you must add a pair of braces:

```
int i = 1, j = 2, k = 3;  
  
if (i > j) {  
    if (i > k)  
        System.out.println("A");  
}  
else  
    System.out.println("B");
```

This statement prints **B**.

3.9 Problem: An Improved Math Learning Tool

Suppose you want to develop a program for a first-grader to practice subtraction. The program randomly generates two single-digit integers, `number1` and `number2`, with `number1 >= number2` and displays to the student a question such as “What is $9 - 2$?” After the student enters the answer, the program displays a message indicating whether it is correct.



Video Note

Program subtraction quiz

The previous programs generate random numbers using `System.currentTimeMillis()`. A better approach is to use the `random()` method in the `Math` class. Invoking this method returns a random double value `d` such that $0.0 \leq d < 1.0$. So, `(int) (Math.random() * 10)` returns a random single-digit integer (i.e., a number between 0 and 9).

`random()` method

The program may work as follows:

- Generate two single-digit integers into `number1` and `number2`.
- If `number1 < number2`, swap `number1` with `number2`.
- Prompt the student to answer “What is `number1 – number2`?”
- Check the student’s answer and display whether the answer is correct.

The complete program is shown in [Listing 3.4](#).

LISTING 3.4 SubtractionQuiz.java

```
1 import java.util.Scanner;
2
3 public class SubtractionQuiz {
4     public static void main(String[] args) {
5         // 1. Generate two random single-digit integers
6         int number1 = (int)(Math.random() * 10);           random numbers
7         int number2 = (int)(Math.random() * 10);
8
9         // 2. If number1 < number2, swap number1 with number2
10        if (number1 < number2) {
11            int temp = number1;
12            number1 = number2;
13            number2 = temp;
14        }
15
16        // 3. Prompt the student to answer "What is number1 – number2?"
17        System.out.print
18            ("What is " + number1 + " - " + number2 + "? ");
19        Scanner input = new Scanner(System.in);
20        int answer = input.nextInt();                      get answer
21
22        // 4. Grade the answer and display the result
23        if (number1 - number2 == answer)                  check the answer
24            System.out.println("You are correct!");
25        else
26            System.out.println("Your answer is wrong\n" + number1 + " - "
27                + number2 + " should be " + (number1 - number2));
28    }
29 }
```



What is $6 - 6$?

You are correct!



What is 9 - 2? **5** 

Your answer is wrong

9 - 2 should be 7



line#

number1

number2

temp

answer

output

6

2

7

9

11

2

12

9

13

2

20

5

26

Your answer is wrong 9 – 2 should be 7

To swap two variables `number1` and `number2`, a temporary variable `temp` (line 11) is used to first hold the value in `number1`. The value in `number2` is assigned to `number1` (line 12), and the value in `temp` is assigned to `number2` (line 13).

3.10 Problem: Computing Body Mass Index

Body Mass Index (BMI) is a measure of health on weight. It can be calculated by taking your weight in kilograms and dividing by the square of your height in meters. The interpretation of BMI for people 16 years or older is as follows:

BMI

Interpretation

below 16

seriously underweight

16–18

underweight

18–24

normal weight

24–29

overweight

29–35

seriously overweight

above 35

gravely overweight

Write a program that prompts the user to enter a weight in pounds and height in inches and display the BMI. Note that one pound is **0.45359237** kilograms and one inch is **0.0254** meters. [Listing 3.5](#) gives the program.

LISTING 3.5 ComputBMT.java

```

1 import java.util.Scanner;
2
3 public class ComputeAndInterpretBMI {
4     public static void main(String[] args) {
5         Scanner input = new Scanner(System.in);
6
7         // Prompt the user to enter weight in pounds
8         System.out.print("Enter weight in pounds: ");
9         double weight = input.nextDouble();
10
11        // Prompt the user to enter height in inches
12        System.out.print("Enter height in inches: ");
13        double height = input.nextDouble();
14
15        final double KILOGRAMS_PER_POUND = 0.45359237; // Constant
16        final double METERS_PER_INCH = 0.0254; // Constant
17
18        // Compute BMI
19        double weightInKilograms = weight * KILOGRAMS_PER_POUND;
20        double heightInMeters = height * METERS_PER_INCH;
21        double bmi = weightInKilograms /
22            (heightInMeters * heightInMeters);
23
24        // Display result
25        System.out.printf("Your BMI is %5.2f\n", bmi);
26        if (bmi < 16)
27            System.out.println("You are seriously underweight");
28        else if (bmi < 18)
29            System.out.println("You are underweight");
30        else if (bmi < 24)
31            System.out.println("You are normal weight");
32        else if (bmi < 29)
33            System.out.println("You are overweight");
34        else if (bmi < 35)
35            System.out.println("You are seriously overweight");
36        else
37            System.out.println("You are gravely overweight");
38    }
39 }
```



Enter weight in pounds: 146



Enter height in inches: 70

Your BMI is 20.948603801493316

You are normal weight



line #

weight

height

WeightInKilograms

heightInMeters

bmi

output

9

146

13

70

19

66.22448602

20

1.778

21

20.9486

25

Your BMI is 20.95

31

You are normal weight

Two constants `KILOGRAMS_PER_POUND` and `METERS_PER_INCH` are defined in lines 15–16. Using constants here makes programs easy to read.

3.11 Problem: Computing Taxes

The United States federal personal income tax is calculated based on filing status and taxable income. There are four filing statuses: single filers, married filing jointly, married filing separately, and head of household. The tax rates vary every year. [Table 3.2](#) shows the rates for 2009. If you are, say, single with a taxable income of \$10,000, the first \$8,350 is taxed at 10% and the other \$1,650 is taxed at 15%. So, your tax is \$1,082.5



Video Note

Use multiple alternative if statements

TABLE 3.2 2009 U.S. Federal Personal Tax Rates

Marginal Tax Rate

Single

Married Filing Jointly or Qualified Widow(er)

Married Filing Separately

Head of Household

10%

\$0 – \$8,350

\$0 – \$16,700 \$0 -

\$8,350

\$0 – \$11,950

15%

\$8,351 – \$33,950

\$16,701 – \$67,900 \$8,

351 – \$33,950

\$11,951 – \$45,500

25%

\$33,951 – \$82,250

\$67,901 – \$137,050

\$33,951 – \$68,525

\$45,501 – \$117,450

28%

\$82,251 – \$171,550

\$137,051 – \$208,850

\$68,525 – \$104,425

\$117,451 – \$190,200

33%

\$171,551 – \$372,950

\$208,851 – \$372,950

\$104,426 – \$186,475

\$190,201 – \$372,950

35%

\$372,951+

\$372,951+

\$186,476+

\$372,951+

You are to write a program to compute personal income tax. Your program should prompt the user to enter the filing status and taxable income and compute the tax. Enter **0** for single filers, **1** for married filing jointly, **2** for married filing separately, and **3** for head of household.

Your program computes the tax for the taxable income based on the filing status. The filing status can be determined using **if** statements outlined as follows:

```
if (status == 0) {
    // Compute tax for single filers
}
else if (status == 1) {
    // Compute tax for married filing jointly
}
else if (status == 2) {
    // Compute tax for married filing separately
}
else if (status == 3) {
    // Compute tax for head of household
}
else {
    // Display wrong status
}
```

For each filing status there are six tax rates. Each rate is applied to a certain amount of taxable income. For example, of a taxable income of \$400,000 for single filers, \$8,350 is taxed at 10%, $(33,950 - 8,350)$ at 15%, $(82,250 - 33,950)$ at 25%, $(171,550 - 82,250)$ at 28%, $(372,950 - 171,550)$ at 33%, and $(400,000 - 372,950)$ at 35%.

[Listing 3.6](#) gives the solution to compute taxes for single filers. The complete solution is left as an exercise.

LISTING 3.6 ComputeTax.java

```
1 import java.util.Scanner;
2
3 public class ComputeTax {
4     public static void main(String[] args) {
5         // Create a Scanner
6         Scanner input = new Scanner(System.in);
7
8         // Prompt the user to enter filing status
9         System.out.print(
10             "(0-single filer, 1-married jointly,\n" +
11             "2-married separately, 3-head of household)\n" +
12             "Enter the filing status: ");
13         int status = input.nextInt();
14
15         // Prompt the user to enter taxable income
16         System.out.print("Enter the taxable income: ");
17         double income = input.nextDouble();
18
19         // Compute tax
20         double tax = 0;
21
22         if (status == 0) { // Compute tax for single filers
23             if (income <= 8350)
24                 tax = income * 0.10;
25             else if (income <= 33950)
26                 tax = 8350 * 0.10 + (income - 8350) * 0.15;
27             else if (income <= 82250)
28                 tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +

```

```

29             (income - 33950) * 0.25;
30     else if (income <= 171550)
31         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
32             (82250 - 33950) * 0.25 + (income - 82250) * 0.28;
33     else if (income <= 372950)
34         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
35             (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
36             (income - 171550) * 0.35;
37     else
38         tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
39             (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
40             (372950 - 171550) * 0.33 + (income - 372950) * 0.35;
41 }
42 else if (status == 1) { // Compute tax for married file jointly
43     // Left as exercise
44 }
45 else if (status == 2) { // Compute tax for married separately
46     // Left as exercise
47 }
48 else if (status == 3) { // Compute tax for head of household
49     // Left as exercise
50 }
51 else {
52     System.out.println("Error: invalid status");
53     System.exit(0);                                exit program
54 }
55
56 // Display the result
57 System.out.println("Tax is " + (int)(tax * 100) / 100.0);    display output
58 }
59 }

```



(0-single filer, 1-married jointly,
2-married separately, 3-head of household)

Enter the filing status: 0



Enter the taxable income: 400000



Tax is 117683.5



line #

status

income

tax

output

13

0

17

400000

20

0

38

117683.5

```
Tax is 117683.5
```

The program receives the filing status and taxable income. The multiple alternative `if` statements (lines 22, 42, 45, 48, 51) check the filing status and compute the tax based on the filing status.

`System.exit(0)` (line 53) is defined in the `System` class. Invoking this method terminates the program. The argument `0` indicates that the program is terminated normally.

`System.exit(0)`

An initial value of `0` is assigned to `tax` (line 20). A syntax error would occur if it had no initial value, because all of the other statements that assign values to `tax` are within the `if` statement. The compiler thinks that these statements may not be executed and therefore reports a syntax error.

To test a program, you should provide the input that covers all cases. For this program, your input should cover all statuses (`0, 1, 2, 3`). For each status, test the tax for each of the six brackets. So, there are a total of 24 cases.

test all cases



Tip

For all programs, you should write a small amount of code and test it before moving on to add more code. This is called *incremental development and testing*. This approach makes debugging easier, because the errors are likely in the new code you just added.

incremental development and testing

3.12 Logical Operators

Sometimes, whether a statement is executed is determined by a combination of several conditions. You can use logical operators to combine these conditions. *Logical operators*, also known as *Boolean operators*, operate on Boolean values to create a new Boolean value. [Table 3.3](#) gives a list of Boolean operators. [Table 3.4](#) defines the not (!) operator. The not (!) operator negates **true** to **false** and **false** to **true**. [Table 3.5](#) defines the and (&&) operator. The and (&&) of two Boolean operands is **true** if and only if both operands are **true**. [Table 3.6](#) defines the or (||) operator. The or (||) of two Boolean operands is **true** if at least one of the operands is **true**. [Table 3.7](#) defines the exclusive or (^) operator. The exclusive or (^) of two Boolean operands is **true** if and only if the two operands have different Boolean values.

TABLE 3.3 Boolean Operators

<i>Operator</i>	<i>Name</i>	<i>Description</i>
!	not	logical negation
&&	and	logical conjunction
	or	logical disjunction
^	exclusive or	logical exclusion

TABLE 3.4 Truth Table for Operator !

p

!p

Example (assume) age = 24, gender = 'F'

true

false

! (age>18) is **false**, because **(age>18)** is **true**. 1age 7 182! 1age 7 182

false

true

! (gender == &M&) is **true**, because **(gender == &M&)** is **false**.

TABLE 3.5 Truth Table for Operator **&&**

p1

p2

p1 && p2

Example (assume) age = 24, gender = 'F'

false

false

false

(age > 18) && (gender == &F&) is **true**, because **(age > 18)** and **(gender == &F&)** are both **true**.

false

true

false

true

```
false  
false  
  
(age > 18) && (gender != &F&) is false, because (gender != &F&) is false.  
  
true  
true  
true
```

TABLE 3.6 Truth Table for Operator ||

p1

p2

p1 || p2

Example (assume age = 24, gender = 'F')

```
false  
false  
false  
  
(age > 34) || (gender == &F&) is true, because (gender == &F&) is true.  
  
false  
true  
true  
  
true  
false  
true
```

`(age > 34) || (gender == &M&)` is **false**, because `(age > 34)` and `(gender == &M&)` are both **false**.

true

true

true

TABLE 3.7 Truth Table for Operator ^

p1

p2

p1 ^ p2

Example (assume age = 24, gender = 'F')

false

false

false

`(age > 34) ^ (gender == &F&)` is **true**, because `(age > 34)` is **false** but `(gender == &F&)` is **true**.

false

true

true

true

false

true

`(age > 34) || (gender == &M&)` is **false**, because `(age > 34)` and `(gender == &M&)` are both **false**.

true
true
false

[Listing 3.7](#) gives a program that checks whether a number is divisible by 2 and 3, by 2 or 3, and by 2 or 3 but not both:

LISTING 3.7 TestBooleanOperators.java

```
1 import java.util.Scanner;                                import class
2
3 public class TestBooleanOperators {
4     public static void main(String[] args) {
5         // Create a Scanner
6         Scanner input = new Scanner(System.in);
7
8         // Receive an input
9         System.out.print("Enter an integer: ");
10        int number = input.nextInt();                      input
11
12        System.out.println("Is " + number +
13            "\n\tdivisible by 2 and 3? " +
14            (number % 2 == 0 && number % 3 == 0 )           and
15            + "\n\tdivisible by 2 or 3? " +
16            (number % 2 == 0 || number % 3 == 0 ) +          or
17            "\n\tdivisible by 2 or 3, but not both? " +
18            + (number % 2 == 0 ^ number % 3 == 0 ));          exclusive or
19    }
20 }
```



Enter an integer: 18

Is 18

```
    divisible by 2 and 3? true
    divisible by 2 or 3? true
    divisible by 2 or 3, but not both? false
```

A long string is formed by concatenating the substrings in lines 12–18. The three `\n` characters display the string in four lines. (`number % 2 == 0 && number % 3 ==`

0) (line 14) checks whether the number is divisible by and 3. (number % 2 == 0 || number % 3 == 0) (line 16) checks whether the number is divisible by 2 or 3. (number % 2 == 0 ^ number % 3 == 0) (line 20) checks whether the number is divisible by 2 or 3, but not both.



Caution

In mathematics, the expression

```
1 <= numberOfDaysInAMonth <= 31
```

is correct. However, it is incorrect in Java, because 1 <= numberOfDaysInAMonth is evaluated to a boolean value, which cannot be compared with 31. Here, two operands (a boolean value and a numeric value) are *incompatible*. The correct expression in Java is

```
1 <= numberOfDaysInAMonth) && (numberOfDaysInAMonth <= 31)
```

incompatible operands



Note

As shown in the preceding chapter, a char value can be cast into an int value, and vice versa. A boolean value, however, cannot be cast into a value of another type, nor can a value of another type be cast into a boolean value.

cannot cast boolean



Note

De Morgan's law, named after Indian-born British mathematician and logician Augustus De Morgan (1806–1871), can be used to simplify Boolean expressions. The law states
! (condition1 && condition2) is same as! condition1 || ! condition2
! (condition1 || condition2) is same as! condition1 && ! condition2

For example,

```
! (n == 2 || n == 3) is same as n != 2 && n != 3  
!(n % 2 == 0 && n % 3 == 0) is same as n % 2 != 0 || n %  
3 != 0
```

De Morgan's law

If one of the operands of an `&&` operator is `false`, the expression is `false`; if one of the operands of an `||` operator is `true`, the expression is `true`. Java uses these properties to improve the performance of these operators. When evaluating `p1 && p2`, Java first evaluates `p1` and then, if `p1` is `true`, evaluates `p2`; if `p1` is `false`, it does not evaluate `p2`. When evaluating `p1 || p2`, Java first evaluates `p1` and then, if `p1` is `false`, evaluates `p2`; if `p1` is `true`, it does not evaluate `p2`. Therefore, `&&` is referred to as the *conditional* or *short-circuit AND* operator, and is referred to as the *conditional* or *short-circuit OR* operator.

conditional operator

short-circuit operator

3.13 Problem: Determining Leap Year

A year is a *leap year* if it is divisible by `4` but not by `100` or if it is divisible by `400`. So you can use the following Boolean expressions to check whether a year is a leap year:

leap year

```
// A leap year is divisible by 4  
boolean isLeapYear = (year % 4 == 0);  
// A leap year is divisible by 4 but not by 100  
isLeapYear = isLeapYear && (year % 100 != 0);  
// A leap year is divisible by 4 but not by 100 or  
// divisible by 400  
isLeapYear = isLeapYear || (year % 400 == 0);
```

or you can combine all these expressions into one like this:

```
isLeapYear = (year % 4 == 0 && year % 100 != 0) || (year %  
400 == 0);
```

[Listing 3.8](#) gives the program that lets the user enter a year and checks whether it is a leap year.

LISTING 3.8 LeapYear.java

```

1 import java.util.Scanner;
2
3 public class LeapYear {
4     public static void main(String[] args) {
5         // Create a Scanner
6         Scanner input = new Scanner(System.in);
7         System.out.print("Enter a year: ");
8         int year = input.nextInt();           input
9
10        // Check if the year is a leap year
11        boolean isLeapYear =               leap year?
12            (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
13
14        // Display the result
15        System.out.println(year + " is a leap year? " + isLeapYear);   display result
16    }
17 }

```



Enter a year: 2008 

2008 is a leap year? true

Enter a year: 2002 

2002 is a leap year? false

3.14 Problem: Lottery

Suppose you want to develop a program to play lottery. The program randomly generates a lottery of a two-digit number, prompts the user to enter a two-digit number, and determines whether the user wins according to the following rule:

1. If the user input matches the lottery in exact order, the award is \$10,000.
2. If all the digits in the user input match all the digits in the lottery, the award is \$3,000.
3. If one digit in the user input matches a digit in the lottery, the award is \$1,000.

The complete program is shown in [Listing 3.9](#).

LISTING 3.9 Lottery.java

```

1 import java.util.Scanner;
2
3 public class Lottery {
4     public static void main(String[] args) {
5         // Generate a lottery
6         int lottery = (int)(Math.random() * 100);           generate a lottery
7
8         // Prompt the user to enter a guess
9         Scanner input = new Scanner(System.in);
10        System.out.print("Enter your lottery pick (two digits): ");
11        int guess = input.nextInt();                      enter a guess
12
13        // Get digits from lottery
14        int lotteryDigit1 = lottery / 10;
15
16                int lotteryDigit2 = lottery % 10;
17
18                // Get digits from guess
19                int guessDigit1 = guess / 10;
20                int guessDigit2 = guess % 10;
21
22                System.out.println("The lottery number is " + lottery);
23
24                // Check the guess
25                if (guess == lottery)
26                    System.out.println("Exact match: you win $10,000");
27                else if (guessDigit2 == lotteryDigit1
28                            && guessDigit1 == lotteryDigit2)
29                    System.out.println("Match all digits: you win $3,000");
30                else if (guessDigit1 == lotteryDigit1
31                            || guessDigit1 == lotteryDigit2
32                            || guessDigit2 == lotteryDigit1
33                            || guessDigit2 == lotteryDigit2)
34                    System.out.println("Match one digit: you win $1,000");
35                else
36                    System.out.println("Sorry, no match");
37    }

```



Enter your lottery pick (two digits): 45

The lottery number is 12

Sorry, no match



Enter your lottery pick: 23

The lottery number is 34

Match one digit: you win \$1,000



variable	line#	6	11	14	15	18	19	33
lottery		34						
guess			23					
lotteryDigit1				3				
lotteryDigit2					4			
guessDigit1						2		
guessDigit2							3	
output								Match one digit: you win \$1,000

The program generates a lottery using the `random()` method (line 6) and prompts the user to enter a guess (line 11). Note that `guess % 10` obtains the last digit from `guess` and `guess / 10` obtains the first digit from `guess`, since `guess` is a two-digit number (lines 18–19).

The program checks the guess against the lottery number in this order:

1. First check whether the guess matches the lottery exactly (line 24).
2. If not, check whether the reversal of the guess matches the lottery (lines 26–27).
3. If not, check whether one digit is in the lottery (lines 29–32).
4. If not, nothing matches.

3.15 switch Statements

The `if` statement in [Listing 3.6](#), `ComputeTax.java`, makes selections based on a single `true` or `false` condition. There are four cases for computing taxes, which depend on the value of `status`. To fully account for all the cases, nested `if` statements were used. Overuse of nested `if` statements makes a program difficult to read. Java provides a `switch` statement to handle multiple conditions efficiently. You could write the following `switch` statement to replace the nested `if` statement in [Listing 3.6](#):

```
switch (status) {  
    case 0: compute taxes for single filers;  
        break;  
    case 1: compute taxes for married filing jointly;
```

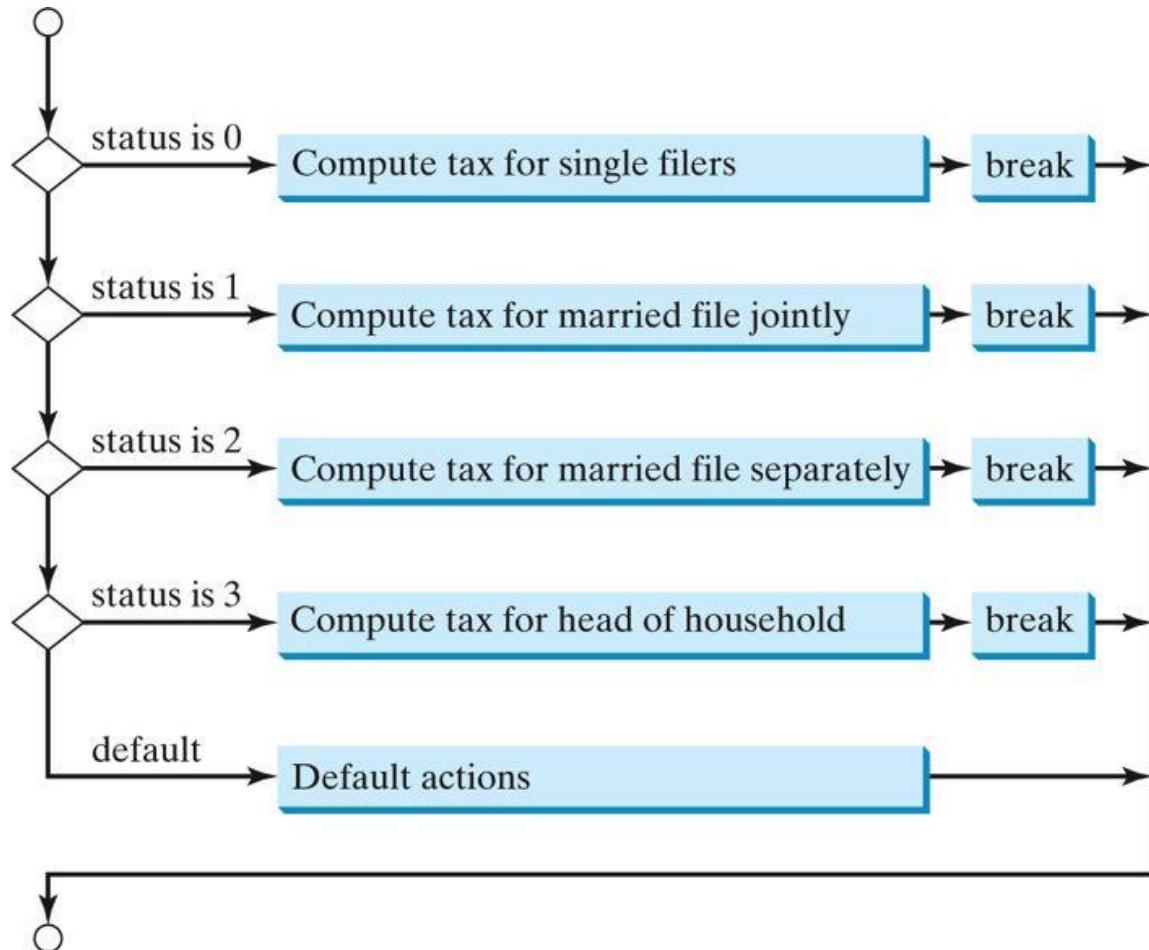
```

        break;
case 2: compute taxes for married filing separately;
        break;
case 3: compute taxes for head of household;
        break;
default: System.out.println("Errors: invalid status");
        System.exit(0);
}

```

The flow chart of the preceding `switch` statement is shown in [Figure 3.5](#).

FIGURE 3.5 The `switch` statement checks all cases and executes the statements in the matched case.



This statement checks to see whether the status matches the value `0`, `1`, `2`, or `3`, in that order. If matched, the corresponding tax is computed; if not matched, a message is displayed. Here is the full syntax for the `switch` statement:

```

switch (switch-expression) {
    case value1: statement(s)1;
        break;

    case value2: statement(s)2;
        break;
    ...
    case valueN: statement(s)N;
        break;
    default:      statement(s)-for-default;
}

```

switch statement

The **switch** statement observes the following rules:

- The **switch-expression** must yield a value of **char**, **byte**, **short**, or **int** type and must always be enclosed in parentheses.
- The **value1**, and **valueN**... must have the same data type as the value of the **switch-expression**. Note that **value1**, and **valueN** are constant expressions, meaning that they cannot contain variables, such as **1 + x**.
- When the value in a **case** statement matches the value of the **switch-expression**, the statements *starting from this case* are executed until either a **break** statement or the end of the switch statement is reached.
- The keyword **break** is optional. The **break** statement immediately ends the **switch** statement.
- The **default** case, which is optional, can be used to perform actions when none of the specified cases matches the **switch-expression**.
- The **case** statements are checked in sequential order, but the order of the cases (including the default case) does not matter. However, it is good programming style to follow the logical sequence of the cases and place the default case at the end.



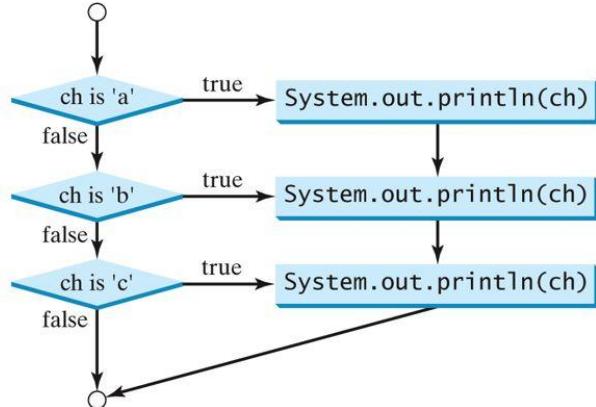
Caution

Do not forget to use a **break** statement when one is needed. Once a case is matched, the statements starting from the matched case are executed until a **break** statement or the end of the **switch** statement is reached. This is referred to as fall-through behavior. For example, the following code prints character **a** three times if **ch** is **&a&**:

```

switch (ch) {
    case 'a': System.out.println(ch);
    case 'b': System.out.println(ch);
    case 'c': System.out.println(ch);
}

```



without **break**

fall-through behavior



Tip

To avoid programming errors and improve code maintainability, it is a good idea to put a comment in a case clause if **break** is purposely omitted.

3.16 Conditional Expressions

You might want to assign a value to a variable that is restricted by certain conditions. For example, the following statement assigns **1** to **y** if **x** is greater than **0**, and **-1** to **y** if **x** is less than or equal to **0**.

```

if (x > 0)
    y = 1;
else
    y = -1;

```

Alternatively, as in this example, you can use a conditional expression to achieve the same result.

```
y = (x > 0) ? 1 : -1
```

Conditional expressions are in a completely different style, with no explicit **if** in the statement. The syntax is shown below:

```
boolean-expression? expression1: expression2;
```

The result of this conditional expression is **expression1** if **boolean-expression** is true; otherwise the result is **expression2**.

conditional expression

Suppose you want to assign the larger number between variable `num1` and `num2` to `max`. You can simply write a statement using the conditional expression:

```
max = (num1 > num2) ? num1 : num2;
```

For another example, the following statement displays the message “`num` is even” if `num` is even, and otherwise displays “`num` is odd.”

```
System.out.println((num % 2 == 0) ? "num is even": "num is odd");
```



Note

The symbols `?` and `:` appear together in a conditional expression. They form a conditional operator. It is called a *ternary operator* because it uses three operands. It is the only ternary operator in Java.

3.17 Formatting Console Output

If you wish to display only two digits after the decimal point in a floating-point value, you may write the code like this:

```
double x = 2.0 / 3;
System.out.println("x is " + (int)(x * 100) / 100.0);
```



```
x is 0.66
```

However, a better way to accomplish this task is to format the output using the `printf` method. The syntax to invoke this method is

```
System.out.printf(format, item1, item2, ..., itemk)
```

printf

where `format` is a string that may consist of substrings and format specifiers.

specifier

A format specifier specifies how an item should be displayed. An item may be a numeric value, a character, a Boolean value, or a string. A simple specifier consists of a percent sign (%) followed by a conversion code. [Table 3.8](#) lists some frequently used simple specifiers:

TABLE 3.8 Frequently Used Specifiers

Specifier

Output

Example

`%b`

a Boolean value

true or false

`%c`

a character

‘a’

`%d`

a decimal integer

200

`%f`

a floating-point number

45.460000

`%e`

a number in standard scientific notation

4.556000e+01

`%s`

a string

“Java is cool”

Here is an example:

```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);
```

display count is 5 and amount is 45.560000

Items must match the specifiers in order, in number, and in exact type. For example, the specifier for `count` is `%d` and for `amount` is `%f`. By default, a floating-point value is displayed with six digits after the decimal point. You can specify the width and precision in a specifier, as shown in the examples in [Table 3.9](#).

TABLE 3.9 Examples of Specifying Width and Precision

Example

Output

`%5c`

Output the character and add four spaces before the character item.

`%6b`

Output the Boolean value and add one space before the false value and two spaces before the true value.

`%5d`

Output the integer item with width at least 5. If the number of digits in the item is < 5, add spaces before the number. If the number of digits in the item is > 5, the width is automatically increased.

`%10.2f`

Output the floating-point item with width at least 10 including a decimal point and two digits after the point. Thus there are 7 digits allocated before the decimal point. If the number of digits before the decimal point in the item < 7, is add spaces before

the number. If the number of digits before the decimal point in the item is > 7, the width is automatically increased.

%10.2e

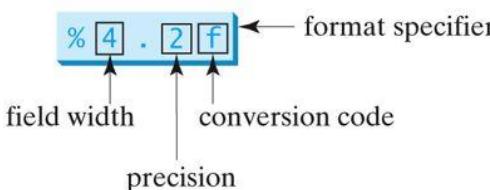
Output the floating-point item with width at least 10 including a decimal point, two digits after the point and the exponent part. If the displayed number in scientific notation has width less than 10, add spaces before the number.

%12s

Output the string with width at least 12 characters. If the string item has less than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.

The code presented in the beginning of this section for displaying only two digits after the decimal point in a floating-point value can be revised using the `printf` method as follows:

```
double x = 2.0 / 3;  
System.out.printf("x is %4.2f", x);  
display          x is 0.67
```



By default, the output is right justified. You can put the minus sign (-) in the specifier to specify that the item is left justified in the output within the specified field. For example, the following statements

```
System.out.printf("%8d%8s%8.1f\n", 1234, "Java", 5.6);  
System.out.printf("%-8d%-8s%-8.1f \n", 1234, "Java", 5.6);
```

display

8 characters	8 characters	8 characters
1 2 3 4	J a v a	5 . 6



Caution

The items must match the specifiers in exact type. The item for the specifier `%f` or `%e` must be a floating-point type value such as 40.0, not 40. Thus an `int` variable cannot match `%f` or `%e`.



Tip

The `%` sign denotes a specifier. To output a literal `%` in the format string, use `%%`.

3.18 Operator Precedence and Associativity

Operator precedence and associativity determine the order in which operators are evaluated. Suppose that you have this expression:

`3 + 4 * 4 > 5 * (4 + 3) - 1`

What is its value? What is the execution order of the operators?

Arithmetically, the expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule.

The precedence rule defines precedence for operators, as shown in [Table 3.10](#), which contains the operators you have learned so far. Operators are listed in decreasing order of precedence from top to bottom. Operators with the same precedence appear in the same group. (See [Appendix C](#), “Operator Precedence Chart,” for a complete list of Java operators and their precedence.)

precedence

If operators with the same precedence are next to each other, their *associativity* determines the order of evaluation. All binary operators except assignment operators are *left associative*. For example, since `+` and `-` are of the same precedence and are left associative, the expression

associativity

$$a - b + c - d \underset{\text{equivalent}}{=} ((a - b) + c) - d$$

TABLE 3.10 Operator Precedence Chart

<i>Precedence</i>	<i>Operator</i>
	<code>var++</code> and <code>var--</code> (Postfix)
	<code>+, -</code> (Unary plus and minus), <code>++var</code> and <code>--var</code> (Prefix)
	(type) (Casting)
	<code>!</code> (Not)
	<code>*, /, %</code> (Multiplication, division, and remainder)
	<code>+, -</code> (Binary addition and subtraction)
	<code><, <=, >, >=</code> (Comparison)
	<code>==, !=</code> (Equality)
	<code>^</code> (Exclusive OR)
	<code>&&</code> (AND)
	<code> </code> (OR)
	<code>=, +=, -=, *=, /=, %=</code> (Assignment operator)

Assignment operators are *right associative*. Therefore, the expression

$$a = b += c = 5 \quad \text{equivalent} \quad a = (b += (c = 5))$$

Suppose `a`, `b`, and `c` are 1 before the assignment; after the whole expression is evaluated, `a` becomes 6, `b` becomes 6, and `c` becomes 5. Note that left associativity for the assignment operator would not make sense.



Note

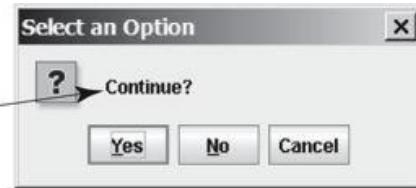
Java has its own way to evaluate an expression internally. The result of a Java evaluation is the same as that of its corresponding arithmetic evaluation. Interested readers may refer to Supplement III.B for more discussions on how an expression is evaluated in Java *behind the scenes*.

behind the scenes

3.19 (GUI) Confirmation Dialogs

You have used `showMessageDialog` to display a message dialog box and `showInputDialog` to display an input dialog box. Occasionally it is useful to answer a question with a confirmation dialog box. A confirmation dialog can be created using the following statement:

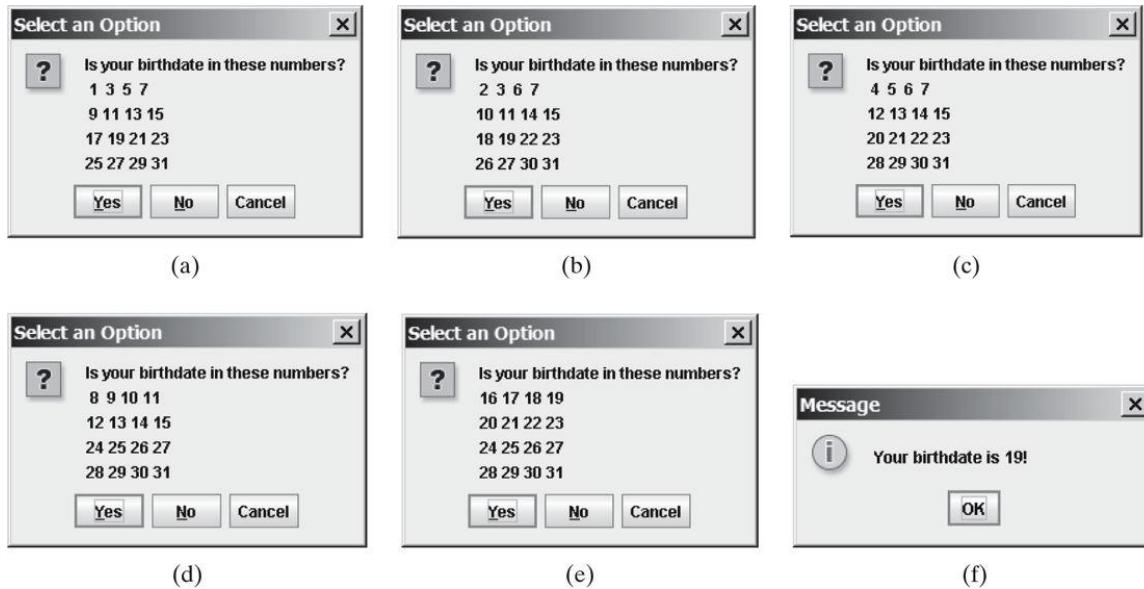
```
int option =  
    JOptionPane.showConfirmDialog  
    (null, "Continue");
```



When a button is clicked, the method returns an option value. The value is `JOptionPane.YES_OPTION` (0) for the *Yes* button, `JOptionPane.NO_OPTION` (1) for the *No* button, and `JOptionPane.CANCEL_OPTION` (2) for the *Cancel* button.

You may rewrite the guess-birthday program in [Listing 3.3](#) using confirmation dialog boxes, as shown in [Listing 3.10](#). [Figure 3.6](#) shows a sample run of the program for the day **19**.

FIGURE 3.6 Click Yes in (a), Yes in (b), No in (c), No in (d), and Yes in (e).



LISTING 3.10
GuessBirthdayUsingConfirmationDialog.java

```
1 import javax.swing.JOptionPane; import class
2
3 public class GuessBirthdayUsingConfirmationDialog {
4     public static void main(String[] args) {
5         String set1 = set1
6             " 1 3 5 7\n" +
7             " 9 11 13 15\n" +
8             "17 19 21 23\n" +
9             "25 27 29 31";
10
11     String set2 = set2
12         " 2 3 6 7\n" +
13         "10 11 14 15\n" +
14         "18 19 22 23\n" +
15         "26 27 30 31";
16
17     String set3 = set3
18         " 4 5 6 7\n" +
19         "12 13 14 15\n" +
20         "20 21 22 23\n" +
21         "28 29 30 31";
22
23     String set4 = set4
24         " 8 9 10 11\n" +
25         "12 13 14 15\n" +
26         "24 25 26 27\n" +
27         "28 29 30 31";
28
29     String set5 = set5
30         "16 17 18 19\n" +
31         "20 21 22 23\n" +
32         "24 25 26 27\n" +
33         "28 29 30 31";
34
```

```

35     int day = 0;
36
37     // Prompt the user to answer questions
38     int answer = JOptionPane.showConfirmDialog(null,
39         "Is your birthday in these numbers?\n" + set1);
40
41     if (answer == JOptionPane.YES_OPTION)
42         day += 1;
43
44     answer = JOptionPane.showConfirmDialog(null,
45         "Is your birthday in these numbers?\n" + set2);
46
47     if (answer == JOptionPane.YES_OPTION)
48         day += 2;
49
50     answer = JOptionPane.showConfirmDialog(null,
51         "Is your birthday in these numbers?\n" + set3);
52
53     if (answer == JOptionPane.YES_OPTION)
54         day += 4;
55
56     answer = JOptionPane.showConfirmDialog(null,
57         "Is your birthday in these numbers?\n" + set4);
58
59     if (answer == JOptionPane.YES_OPTION)
60         day += 8;
61
62     answer = JOptionPane.showConfirmDialog(null,
63         "Is your birthday in these numbers?\n" + set5);
64
65     if (answer == JOptionPane.YES_OPTION)
66         day += 16;
67
68     JOptionPane.showMessageDialog(null, "Your birthday is " +
69         day + "!");
70 }
71 }
```

The program displays confirmation dialog boxes to prompt the user to answer whether a number is in Set1 (line 38), Set2 (line 44), Set3 (line 50), Set4 (line 56), and Set5 (line 62). If the answer is Yes, the first number in the set is added to `day` (lines 42, 48, 54, 60, and 66).

KEY TERMS

Boolean expression [72](#)

Boolean value [72](#)

`boolean` type [72](#)

`break` statement [94](#)

conditional operator [90](#)

dangling-`else` ambiguity [82](#)

fall-through behavior [94](#)

operator associativity [97](#)

operator precedence [97](#)

selection statement [74](#)

short-circuit evaluation [90](#)

CHAPTER SUMMARY

1. A **boolean** variable stores a **true** or **false** value.
2. The relational operators (**<**, **<=**, **==**, **!=**, **>**, **>=**) work with numbers and characters, and yield a Boolean value.
3. The Boolean operators **&&**, **||**, **!**, and **^** operate with Boolean values and variables.
4. When evaluating **p1 && p2**, Java first evaluates **p1** and then evaluates **p2** if **p1** is **true**; if **p1** is **false**, it does not evaluate **p2**. When evaluating **p1 p2**, Java first evaluates **p1** and then evaluates **p2** if **p1** is **false**; if **p1** is **true**, it does not evaluate **p2**. Therefore, **&&** is referred to as the conditional or short-circuit AND operator, and **||** is referred to as the conditional or short-circuit OR operator.
5. Selection statements are used for programming with alternative courses. There are several types of selection statements: **if** statements, **if ... else** statements, nested **if** statements, **switch** statements, and conditional expressions.
6. The various **if** statements all make control decisions based on a Boolean expression. Based on the **true** or **false** evaluation of the expression, these statements take one of two possible courses.
7. The **switch** statement makes control decisions based on a switch expression of type **char**, **byte**, **short**, or **int**.
8. The keyword **break** is optional in a switch statement, but it is normally used at the end of each case in order to terminate the remainder of the **switch** statement. If the **break** statement is not present, the next **case** statement will be executed.

REVIEW QUESTIONS

Section 3.2

3.1 List six comparison operators.

3.2 Can the following conversions involving casting be allowed? If so, find the converted result.

```
boolean b = true;
i = (int)b;
int i = 1;
boolean b = (boolean)i;
```

Sections 3.3–3.11

3.3 What is the printout of the code in (a) and (b) if `number` is **30** and **35**, respectively?

```
if (number % 2 == 0)
    System.out.println(number + " is even.");
System.out.println(number + " is odd.");
```

(a)

```
if (number % 2 == 0)
    System.out.println(number + " is even.");
else
    System.out.println(number + " is odd.");
```

(b)

3.4 Suppose `x=3` and `y=2`; show the output, if any, of the following code. What is the output if `x=3` and `y=4`? What is the output if `x=2` and `y=2`? Draw a flow chart of the code:

```
if (x > 2) {
    if (y > 2) {
        z = x + y;
        System.out.println("z is " + z);
    }
}
else
    System.out.println("x is " + x);
```

3.5 Which of the following statements are equivalent? Which ones are correctly indented?

```
if (i > 0) if
(j > 0)
x = 0; else
if (k > 0) y = 0;
else z = 0;
```

(a)

```
if (i > 0) {
    if (j > 0)
        x = 0;
    else if (k > 0)
        y = 0;
}
else
    z = 0;
```

(b)

```
if (i > 0)
    if (j > 0)
        x = 0;
    else if (k > 0)
        y = 0;
    else
        z = 0;
```

(c)

```
if (i > 0)
    if (j > 0)
        x = 0;
    else if (k > 0)
        y = 0;
    else
        z = 0;
```

(d)

3.6 Suppose **x=2** and **y=3**. Show the output, if any, of the following code. What is the output if **x=3** and **y=2**? What is the output if **x=3** and **y=3**?

(Hint: Indent the statement correctly first.)

```
if (x > 2)
    if (y > 2) {
        int z = x + y;
        System.out.println("z is " + z);
    }
else
    System.out.println("x is " + x);
```

3.7 Are the following two statements equivalent?

```
if (income <= 10000)
    tax = income * 0.1;
else if (income <= 20000)
    tax = 1000 +
        (income - 10000) * 0.15;
```

```
if (income <= 10000)
    tax = income * 0.1;
else if (income > 10000 &&
         income <= 20000)
    tax = 1000 +
        (income - 10000) * 0.15;
```

3.8 Which of the following is a possible output from invoking **Math.random()**?

323.4, 0.5, 34, 1.0, 0.0, 0.234

3.9 How do you generate a random integer **i** such that $0 \leq i \leq 20$? How do you generate a random integer **i** such that $10 \leq i \leq 20$? How do you generate a random integer **i** such that $10 \leq i \leq 50$?

3.10 Write an **if** statement that assigns **1** to **x** if **y** is greater than **0**.

3.11 (a) Write an **if** statement that increases **pay** by 3% if **score** is greater than **90**. (b) Write an **if** statement that increases **pay** by 3% if **score** is greater than **90**, otherwise increases **pay** by 1%.

3.12 What is wrong in the following code?

```
if (score >= 60.0)
    grade = &D&;
else if (score >= 70.0)
    grade = &C&;
else if (score >= 80.0)
    grade = &B&;
else if (score >= 90.0)
    grade = &A&;
else
```

```
grade = &F&;
```

- 3.13** Rewrite the following statement using a Boolean expression:

```
if (count % 10 == 0)
    newLine = true;
else
    newLine = false;
```

Sections 3.12–3.14

- 3.14** Assuming that **x** is 1, show the result of the following Boolean expressions.

```
(true) && (3 > 4)
!(x > 0) && (x > 0)
(x > 0) || (x < 0)
(x != 0) || (x == 0)
(x >= 0) || (x < 0)
(x != 1) == !(x == 1)
```

- 3.15** Write a Boolean expression that evaluates to **true** if a number stored in variable **num** is between 1 and 100.

- 3.16** Write a Boolean expression that evaluates to **true** if a number stored in variable **num** is between 1 and 100 or the number is negative.

- 3.17** Assume that **x** and **y** are **int** type. Which of the following are legal Java expressions?

```
x > y > 0
x = y && y
x /= y
x or y
x and y
(x != 0) || (x = 0)
```

- 3.18** Suppose that **x** is 1. What is **x** after the evaluation of the following expression?

```
(x >= 1) && (x++ > 1)
(x > 1) && (x++ > 1)
```

- 3.19** What is the value of the expression **ch >= & A & && ch <= & Z &** if **ch** is **& A &, & P &, & E &, or & 5 &**?

- 3.20** Suppose, when you run the program, you enter input **236** from the console. What is the output?

```
public class Test {
```

```

public static void main(String[] args) {
    java.util.Scanner input = new
    java.util.Scanner(System.in);
    double x = input.nextDouble();
    double y = input.nextDouble();
    double z = input.nextDouble();
    System.out.println("(x < y && y < z) is " + (x
    < y && y < z));
    System.out.println("(x < y || y < z) is " + (x
    < y || y < z));
    System.out.println("!(x < y) is " + !(x < y));
    System.out.println("(x + y < z) is " + (x + y <
    z));
    System.out.println("(x + y < z) is " + (x + y <
    z));
}
}

```

3.21 Write a Boolean expression that evaluates **true** if **age** is greater than **13** and less than **18**.

3.22 Write a Boolean expression that evaluates **true** if **weight** is greater than **50** or height is greater than **160**.

3.23 Write a Boolean expression that evaluates **true** if **weight** is greater than **50** and height is greater than **160**.

3.24 Write a Boolean expression that evaluates **true** if either **weight** is greater than **50** or height is greater than **160**, but not both.

Section 3.15

3.25 What data types are required for a **switch** variable? If the keyword **break** is not used after a case is processed, what is the next statement to be executed? Can you convert a **switch** statement to an equivalent **if** statement, or vice versa? What are the advantages of using a **switch** statement?

3.26 What is **y** after the following **switch** statement is executed?

```

x= 3; y = 3;
switch (x + 3) {
    case 6: y = 1;
    default: y += 1 ;
}

```

3.27 Use a **switch** statement to rewrite the following **if** statement and draw the flow chart for the **switch** statement:

```
if (a == 1)
    x += 5;
else if (a == 2)
    x += 10;
else if (a == 3)
    x += 16;
else if (a == 4)
    x += 34;
```

3.28 Write a **switch** statement that assigns a **String** variable **dayName** with Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, if **day** is **0, 1, 2, 3, 4, 5, 6**, accordingly.

Section 3.16

3.29 Rewrite the following **if** statement using the conditional operator:

```
if (count % 10 == 0)
    System.out.print(count + "\n");
else
    System.out.print(count + " ");
```

3.30 Rewrite the following statement using a conditional expression:

```
if (temperature > 90)
    pay = pay * 1.5;
else
    pay = pay * 1.1 ;
```

Section 3.17

3.31 What are the specifiers for outputting a Boolean value, a character, a decimal integer, a floating-point number, and a string?

3.32 What is wrong in the following statements?

- (a) `System.out.printf("%5d %d", 1, 2, 3);`
- (b) `System.out.printf("%5d %f", 1);`
- (c) `System.out.printf("%5d %f", 1, 2);`

3.33 Show the output of the following statements.

- (a) `System.out.printf("amount is %f %e\n", 32.32, 32.32);`
- (b) `System.out.printf("amount is %5.4f %5.4e\n", 32.32, 32.32);`

- (c) `System.out.printf("%6b\n", (1>2));`
- (d) `System.out.printf("%6s\n", "Java");`
- (e) `System.out.printf("%-6b%s\n", 1>2), "Java");`
- (f) `System.out.printf("%6b%-s\n", (1>2), "Java");`

3.34 How do you create a formatted string?

Section 3.18

3.35 List the precedence order of the Boolean operators. Evaluate the following expressions:

`true | true&&false`
`true&&true | false`

3.36 True or false? All the binary operators except = are left associative.

3.37 Evaluate the following expressions:

`2*2 -3>2&&4-2 >5`
`2 *2 -3 >2 || 4 -2 >5`

3.38 Is `(x > 0 && x < 10)` the same as `((x > 0) && (x < 10))`?

Is `(x > 0 || x < 10)` the same as `((x > 0) || (x < 10))`? Is `(x >0 || x<10&&y<0)` the same as `(x > 0 || (x < 10 && y < 0))`?

Section 3.19

3.39 How do you display a confirmation dialog? What value is returned when invoking `JOptionPane.showConfirmDialog`?

PROGRAMMING EXERCISES



Pedagogical Note

For each exercise, students should carefully analyze the problem requirements and design strategies for solving the problem before coding.

think before coding



Pedagogical Note

Instructors may ask students to document analysis and design for selected exercises. Students should use their own words to analyze the problem, including the input, output, and what needs to be computed, and describe how to solve the problem in pseudocode.

document analysis and design



Debugging Tip

Before you ask for help, read and explain the program to yourself, and trace it using several representative inputs by hand or using an IDE debugger. You learn how to program by debugging your own mistakes.

learn from mistakes

Section 3.2

3.1* (*Algebra: solving quadratic equations*) The two roots of a quadratic equation $ax^2 + bx + c = 0$ can be obtained using the following formula:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ and } r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$b^2 - 4ac$ is called the discriminant of the quadratic equation. If it is positive, the equation has two real roots. If it is zero, the equation has one root. If it is negative, the equation has no real roots.

Write a program that prompts the user to enter values for a , b , and c and displays the result based on the discriminant. If the discriminant is positive, display two roots. If the discriminant is 0, display one root. Otherwise, display “The equation has no real roots”.

Note you can use `Math.pow(x, 0.5)` to compute \sqrt{x} . Here are some sample runs.



Enter a, b, c: 1.0 3 1

The roots are -0.381966 and -2.61803



Enter a, b, c: 1 2.0 1

The root is -1



Enter a, b, c: 1 2 3

The equation has no real roots

3.2 (*Checking whether a number is even*) Write a program that reads an integer and checks whether it is even. Here are the sample runs of this program:



Enter an integer: 25

Is 25 an even number? false



Enter an integer: **2000** 

Is 2000 an even number? true

Sections 3.3–3.8

3.3* (*Algebra: solving 2×2 linear equations*) You can use Cramer's rule to solve the following 2×2 system of linear equation:

$$\begin{aligned} ax + by &= e \\ cx + dy &= f \end{aligned} \quad x = \frac{ed - bf}{ad - bc} \quad y = \frac{af - ec}{ad - bc}$$

Write a program that prompts the user to enter **a**, **b**, **c**, **d**, **e**, and **f** and display the result. If is **0**, report that “The equation has no solution”.



Enter a, b, c, d, e, f: **9.0 4.0 3.0 -5.0 -6.0 -21.0**



x is -2.0 and y is 3.0



Enter a, b, c, d, e, f: **1.0 2.0 2.0 4.0 4.0 5.0**



The equation has no solution

3.4** (*Game: learning addition*) Write a program that generates two integers under 100 and prompts the user to enter the sum of these two integers. The program then reports true if the answer is correct, false otherwise. The program is similar to [Listing 3.1](#).

3.5** (*Game: addition for three numbers*) The program in [Listing 3.1](#) generates two integers and prompts the user to enter the sum of these two integers. Revise the

program to generate three single-digit integers and prompt the user to enter the sum of these three integers.

3.6* (*Health application: BMI*) Revise [Listing 3.5](#), ComputeBMI.java, to let the user enter weight, feet, and inches. For example, if a person is 5 feet and 10 inches, you will enter 5 for feet and 10 for inches.

3.7 (*Financial application: monetary units*) Modify [Listing 2.10](#), ComputeChange.java, to display the nonzero denominations only, using singular words for single units such as 1 dollar and 1 penny, and plural words for more than one unit such as 2 dollars and 3 pennies. (Use input 23.67 to test your program.)



Video Note

Sort three integers

3.8* (*Sorting three integers*) Write a program that sorts three integers. The integers are entered from the input dialogs and stored in variables `num1`, `num2`, and `num3`, respectively. The program sorts the numbers so that $num1 \leq num2 \leq num3$.

3.9 (*Business: checking ISBN*) An ISBN (International Standard Book Number) consists of 10 digits $d_1d_2d_3d_4d_5d_6d_7d_8d_9d_{10}$. The last digit d_{10} is a checksum, which is calculated from the other nine digits using the following formula:

$$(d_1 \times 1 + d_2 \times 2 + d_3 \times 3 + d_4 \times 4 + d_5 \times 5 + \\ (d_6 \times 6 + d_7 \times 7 + d_8 \times 8 + d_9 \times 9)) \% 11$$

If the checksum is 10, the last digit is denoted X according to the ISBN convention. Write a program that prompts the user to enter the first 9 digits and displays the 10-digit ISBN (including leading zeros). Your program should read the input as an integer. For example, if you enter 013601267, the program should display 0136012671.

3.10* (*Game: addition quiz*) [Listing 3.4](#), SubtractionQuiz.java, randomly generates a subtraction question. Revise the program to randomly generate an addition question with two integers less than [100](#).

Sections 3.9–3.19

3.11* (*Finding the number of days in a month*) Write a program that prompts the user to enter the month and year and displays the number of days in the month. For example, if the user entered month [2](#) and year [2000](#), the program should display that February 2000 has 29 days. If the user entered month [3](#) and year [2005](#), the program should display that March 2005 has 31 days.

3.12 (*Checking a number*) Write a program that prompts the user to enter an integer and checks whether the number is divisible by both [5](#) and [6](#), or neither of them, or just one of them. Here are some sample runs for inputs [10](#), [30](#), and [23](#).

```
10 is divisible by 5 or 6, but not both
30 is divisible by both 5 and 6
23 is not divisible by either 5 or 6
```

3.13 (*Financial application: computing taxes*) [Listing 3.6](#), ComputeTax.java, gives the source code to compute taxes for single filers. Complete [Listing 3.6](#) to give the complete source code.

3.14 (*Game: head or tail*) Write a program that lets the user guess the head or tail of a coin. The program randomly generates an integer [0](#) or [1](#), which represents head or tail. The program prompts the user to enter a guess and reports whether the guess is correct or incorrect.

3.15* (*Game: lottery*) Revise [Listing 3.9](#), Lottery.java, to generate a lottery of a three-digit number. The program prompts the user to enter a three-digit number and determines whether the user wins according to the following rule:

1. If the user input matches the lottery in exact order, the award is \$10,000.
2. If all the digits in the user input match all the digits in the lottery, the award is \$3,000.
3. If one digit in the user input matches a digit in the lottery, the award is \$1,000.

3.16 (*Random character*) Write a program that displays a random uppercase letter using the [Math.random\(\)](#) method.

3.17* (*Game: scissor, rock, paper*) Write a program that plays the popular scissor-rock-paper game. (A scissor can cut a paper, a rock can knock a scissor, and a paper can wrap a rock.) The program randomly generates a number **0**, **1**, or **2** representing scissor, rock, and paper. The program prompts the user to enter a number **0**, **1**, or **2** and displays a message indicating whether the user or the computer wins, loses, or draws. Here are sample runs:



scissor (0), rock (1), paper (2): **1**



The computer is scissor. You are rock. You won



scissor (0), rock (1), paper (2): **2**



The computer is paper. You are paper too. It is a draw

3.18* (*Using the input dialog box*) Rewrite [Listing 3.8](#), LeapYear.java, using the input dialog box.

3.19 (*Validating triangles*) Write a program that reads three edges for a triangle and determines whether the input is valid. The input is valid if the sum of any two edges is greater than the third edge. Here are the sample runs of this program:



Enter three edges: **1 2.5 1**



Can edges 1, 2.5, and 1 form a triangle? **false**



← Enter

Enter three edges: 2.5 2 1

Can edges 2.5, 2, and 1 form a triangle? true

3.20* (*Science: wind-chill temperature*) Exercise 2.17 gives a formula to compute the wind-chill temperature. The formula is valid for temperature in the range between and 41°F and wind speed greater than or equal to 2. Write a program that prompts the user to enter a temperature and a wind speed. The program displays the wind-chill temperature if the input is valid, otherwise displays a message indicating whether the temperature and/or wind speed is invalid.

Comprehensives

3.21** (*Science: day of the week*) Zeller's congruence is an algorithm developed by Christian Zeller to calculate the day of the week. The formula is

$$h = \left(q + \left\lfloor \frac{26(m+1)}{10} \right\rfloor + k + \left\lfloor \frac{k}{4} \right\rfloor + \left\lfloor \frac{j}{4} \right\rfloor + 5j \right) \% 7$$

where

- **h** is the day of the week (0: Saturday, 1: Sunday, 2: Monday, 3: Tuesday, 4: Wednesday, 5: Thursday, 6: Friday).
- **q** is the day of the month.
- **m** is the month (3: March, 4: April, ..., 12: December). January and February are counted as months 13 and 14 of the previous year.
- **j** is the century (i.e., $\left\lfloor \frac{\text{year}}{100} \right\rfloor$).
- **k** is the year of the century (i.e., year \% 7).

Write a program that prompts the user to enter a year, month, and day of the month, and displays the name of the day of the week. Here are some sample runs:



Enter year: (e.g., 2008): 2002

Enter

Enter month: 1-12: 3

Enter

Enter the day of the month: 1-31: 26

Enter

Day of the week is Tuesday



Enter year: (e.g., 2008): 2011

Enter

Enter month: 1-12: 5

Enter

Enter the day of the month: 1-31: 2

Enter

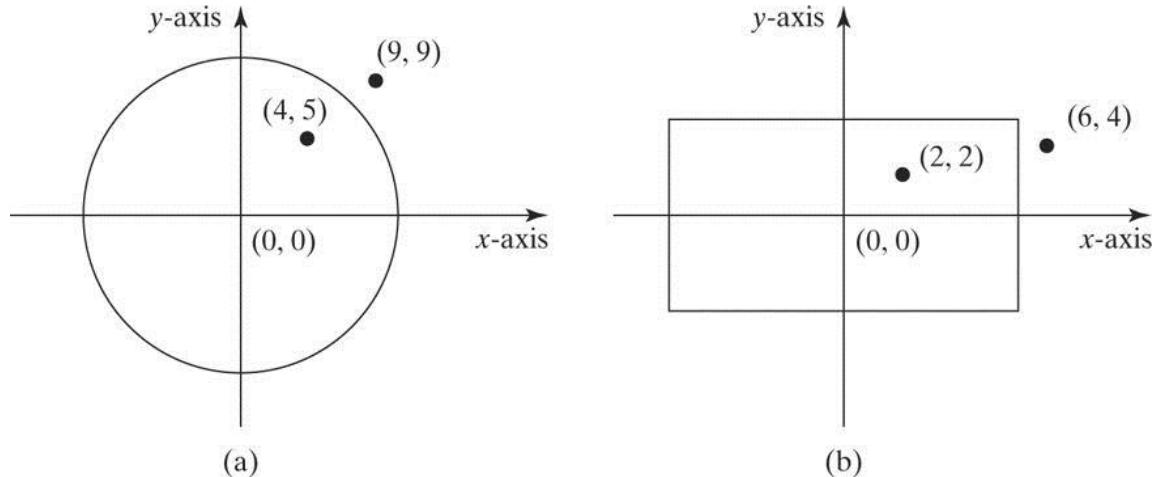
Day of the week is Thursday

(Hint: $\lfloor n \rfloor = (\text{int}) n$ for a positive n . January and February are counted as **13** and **14** in the formula. So you need to convert the user input **1** to **13** and **2** to **14** for the month and change the year to the previous year.)

3.22** (*Geometry: point in a circle?*) Write a program that prompts the user to enter a point (**x**, **y**) and checks whether the point is within the circle centered at (**0, 0**) with radius **10**. For example, (**4, 5**) is inside the circle and (**9, 9**) is outside the circle, as shown in [Figure 3.7\(a\)](#).

(Hint: A point is in the circle if its distance from the center is equal to 10. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$.) Two sample runs are shown below.)

FIGURE 3.7 (a) Points inside and outside of the circle; (b) Points inside and outside of the rectangle.



Enter a point with two coordinates:

Point (4.0, 5.0) is in the circle



Enter a point with two coordinates:

Point (9.0, 9.0) is not in the circle

3.23 (Geometry: point in a rectangle?)** Write a program that prompts the user to enter a point (`x`, `y`) and checks whether the point is within the rectangle

centered at $(0, 0)$ with width **10** and height **5**. For example, $(2, 2)$ is inside the rectangle and $(6, 4)$ is outside the circle, as shown in [Figure 3.7\(b\)](#).

(Hint: A point is in the rectangle if its horizontal distance to $(0, 0)$ is less than or equal to $10 / 2$ and its vertical distance to $(0, 0)$ is less than or equal to $5/2$.) Here are two sample runs. Two sample runs are shown below.)



Enter a point with two coordinates: **2 2**

Point $(2.0, 2.0)$ is in the rectangle



Enter a point with two coordinates: **6 4**

Point $(6.0, 4.0)$ is not in the rectangle

3.24** (*Game: picking a card*) Write a program that simulates picking a card from a deck of **52** cards. Your program should display the rank (**Ace**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **10**, **Jack**, **Queen**, **King**) and suit (**Clubs**, **Diamonds**, **Hearts**, **Spades**) of the card. Here is a sample run of the program:



The card you picked is Jack of Hearts

3.25** (*Computing the perimeter of a triangle*) Write a program that reads three edges for a triangle and computes the perimeter if the input is valid. Otherwise, display that the input is invalid. The input is valid if the sum of any two edges is greater than the third edge.

3.26 (*Using the `&&`, `||` and `^` operators*) Write a program that prompts the user to enter an integer and determines whether it is divisible by 5 and 6, whether it is

divisible by 5 or 6, and whether it is divisible by 5 or 6, but not both. Here is a sample run of this program:

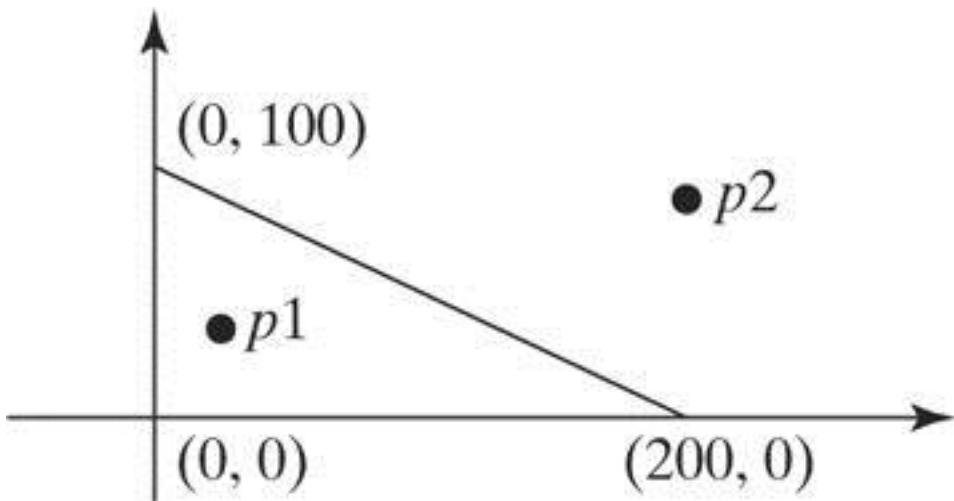


Is 10 divisible by 5 and 6? false

Is 10 divisible by 5 or 6? true

Is 10 divisible by 5 or 6, but not both? true

3.27** (*Geometry: points in triangle?*) Suppose a right triangle is placed in a plane as shown below. The right-angle point is placed at $(0, 0)$, and the other two points are placed at $(200, 0)$, and $(0, 100)$. Write a program that prompts the user to enter a point with x- and y-coordinates and determines whether the point is inside the triangle. Here are the sample runs:



Enter a point's x- and y-coordinates: **100.5 25.5**

↵ Enter

The point is in the triangle



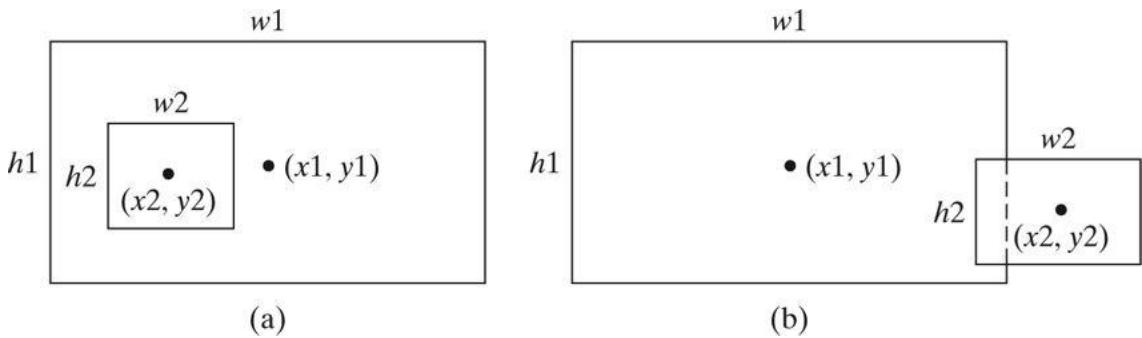
Enter a point's x- and y-coordinates: 100.5 50.5

↙ Enter

The point is not in the triangle

3.28** (*Geometry: two rectangles*) Write a program that prompts the user to enter the center x -, y -coordinates, width, and height of two rectangles and determines whether the second rectangle is inside the first or overlaps with the first, as shown in [Figure 3.8](#).

FIGURE 3.8 (a) A rectangle is inside another one. (b) A rectangle overlaps another one.



Here are the sample runs:



Enter r1's center x-, y-coordinates, width, and height:

2.5 4 2.5 43

↙ Enter

Enter r2's center x-, y-coordinates, width, and height:

1.5 5 0.5 3

↙ Enter

r2 is inside r1



Enter r1's center x-, y-coordinates, width, and height:

1 2 3 5.5

Enter r2's center x-, y-coordinates, width, and height:

3 4 4.5 5



Enter r1's center x-, y-coordinates, width, and height:

1 2 3 3

Enter r2's center x-, y-coordinates, width, and height:

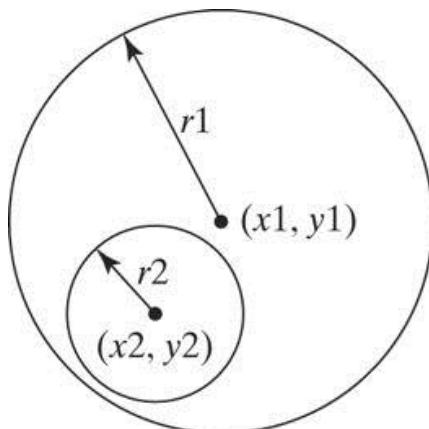
40 45 3 2

r2 does not overlap r1

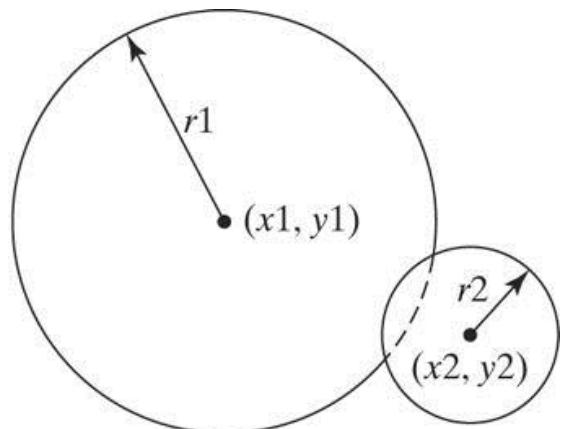
3.29** (*Geometry: two circles*) Write a program that prompts the user to enter the center coordinates and radii of two circles and determines whether the second circle is inside the first or overlaps with the first, as shown in [Figure 3.9](#).

(Hint: circle2 is inside circle1 if the distance between the two centers $\leq |r1 - r2|$ and circle2 overlaps circle1 if the distance between the two centers $\leq r1 + r2$.)

FIGURE 3.9 (a) A circle is inside another circle. (b) A circle overlaps another circle.



(a)



(b)

Here are the sample runs:



Enter circle1's center x-, y-coordinates, and radius:

0.5 5.1 13
 ↓ Enter

Enter circle2's center x-, y-coordinates, and radius: 1

1.7 4.5
 ↓ Enter

circle2 is inside circle1



Enter circle1's center x-, y-coordinates, and radius:

3.4 5.7 5.5
 ↓ Enter

Enter circle2's center x-, y-coordinates, and radius:

6.7 3.5 3
 ↓ Enter

circle2 overlaps circle1



Enter circle1's center x-, y-coordinates, and radius:

3.4 5.5 1

← Enter

Enter circle2's center x-, y-coordinates, and radius:

5.5 7.2 1

← Enter

circle2 does not overlap circle1

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 71).
<vbk:9781256335153#outline(7)>

CHAPTER 4 LOOPS

Objectives

- To write programs for executing statements repeatedly using a `while` loop ([§4.2](#)).
- To develop a program for `GuessNumber` ([§4.2.1](#)).
- To follow the loop design strategy to develop loops ([§4.2.2](#)).
- To develop a program for `SubtractionQuizLoop` ([§4.2.3](#)).
- To control a loop with a sentinel value ([§4.2.4](#)).
- To obtain large input from a file using input redirection rather than typing from the keyboard ([§4.2.4](#)).
- To write loops using `do-while` statements ([§4.3](#)).
- To write loops using `for` statements ([§4.4](#)).
- To discover the similarities and differences of three types of loop statements ([§4.5](#)).
- To write nested loops ([§4.6](#)).
- To learn the techniques for minimizing numerical errors ([§4.7](#)).
- To learn loops from a variety of examples (`GCD`, `FutureTuition`, `MonteCarloSimulation`) ([§4.8](#)).
- To implement program control with `break` and `continue` ([§4.9](#)).
- (GUI) To control a loop with a confirmation dialog ([§4.10](#)).

4.1 Introduction

Suppose that you need to print a string (e.g., `"Welcome to Java!"`) a hundred times. It would be tedious to have to write the following statement a hundred times:

100 times {
 `System.out.println("Welcome to Java!");`
 `System.out.println("Welcome to Java!");`
 `...`
 `System.out.println("Welcome to Java!");`

So, how do you solve this problem?

problem

why loop?

Java provides a powerful construct called a *loop* that controls how many times an operation or a sequence of operations is performed in succession. Using a loop statement, you simply tell the computer to print a string a hundred times without having to code the print statement a hundred times, as follows:

```
int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java!");
    count++;
}
```

The variable `count` is initially `0`. The loop checks whether (`count < 100`) is `true`. If so, it executes the loop body to print the message `"Welcome to Java!"` and increments `count` by `1`. It repeatedly executes the loop body until (`count < 100`) becomes `false`. When (`count < 100`) is `false` (i.e., when `count` reaches `100`), the loop terminates and the next statement after the loop statement is executed.

Loops are constructs that control repeated executions of a block of statements. The concept of looping is fundamental to programming. Java provides three types of loop statements: `while` loops, `do-while` loops, and `for` loops.

4.2 The `while` Loop

The syntax for the `while` loop is as follows:

```
while (loop-continuation-condition) {
    // Loop body
    Statement(s);
}
```

while loop

[Figure 4.1\(a\)](#) shows the `while`-loop flow chart. The part of the loop that contains the statements to be repeated is called the *loop body*. A one-time execution of a loop body is referred to as an *iteration of the loop*. Each loop contains a loop-continuation-condition, a Boolean expression that controls the execution of the body. It is evaluated each time to determine if the loop body is executed. If its evaluation is `true`, the loop body is executed; if its evaluation is `false`, the entire loop terminates and the program control turns to the statement that follows the `while` loop.

loop body

iteration

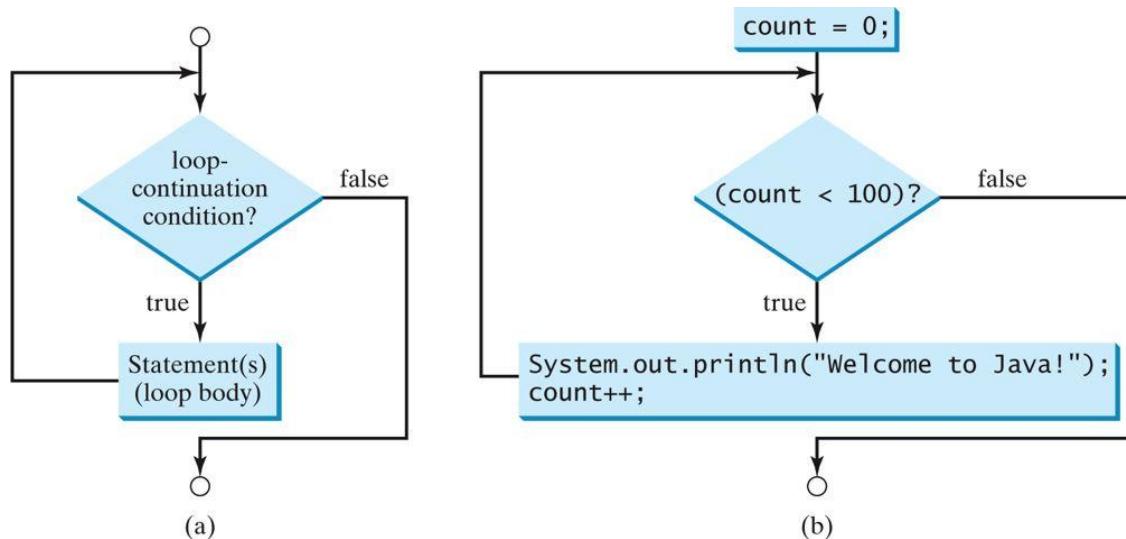
The loop for printing `Welcome to Java!` a hundred times introduced in the preceding section is an example of a `while` loop. Its flow chart is shown in [Figure 4.1\(b\)](#). The **loop-continuation-condition** is `(count < 100)` and loop body contains two statements as shown below:

```
int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java!");
    count++;
}
```

loop-continuation-condition

loop body

FIGURE 4.1 The `while` loop repeatedly executes the statements in the loop body when the loop-continuation-condition evaluates to true.



In this example, you know exactly how many times the loop body needs to be executed. So a control variable `count` is used to count the number of executions. This type of loop is known as a *counter-controlled loop*.

counter-controlled loop



Note

The **loop-continuation-condition** must always appear inside the parentheses. The braces enclosing the loop body can be omitted only if the loop body contains one or no statement.

Here is another example to help understand how a loop works.

```
int sum = 0, i = 1;
while (i < 10) {
    sum = sum + i;
    i++;
}
System.out.println("sum is " + sum); // sum is 45
```

If **i<10** is **true**, the program adds **i** to **sum**. Variable **i** is initially set to **1**, then incremented to **2**, **3**, and up to **10**. When **i** is **10**, **i<10** is **false**, the loop exits. So, the sum is **1+2+3+ ... +9=45**.

What happens if the loop is mistakenly written as follows:

```
int sum = 0, i = 1;
while (i < 10) {
    sum = sum + i;
}
```

This loop is infinite, because **i** is always **1** and **i<10** will always be **true**.



Caution

Make sure that the **loop-continuation-condition** eventually becomes **false** so that the program will terminate. A common programming error involves infinite loops. That is, the program cannot terminate because of a mistake in the **loop-continuation-condition**.

Programmers often make mistakes to execute a loop one more or less time. This is commonly known as the *off-by-one error*. For example, the following loop displays **Welcome to Java** 101 times rather than 100 times. The error lies in the condition, which should be **count < 100** rather than **count <= 100**.

infinite loop

off-by-one error

```
int count = 0;
while (count <= 100) {
    System.out.println("Welcome to Java!");
    count++;
}
```

4.2.1 Problem: Guessing Numbers

The problem is to guess what a number a computer has in mind. You will write a program that randomly generates an integer between 0 and 100, inclusive. The program prompts the user to enter a number continuously until the number matches the randomly generated number. For each user input, the program tells the user whether the input is too low or too high, so the user can make the next guess intelligently. Here is a sample run:



Video Note

Guess a number

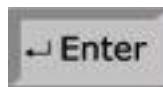


Guess a magic number between 0 and 100



Enter your guess: 50

Your guess is too high



Enter your guess: 25

Your guess is too high



Enter your guess: 12

Your guess is too high

Enter your guess: 6



Your guess is too low

Enter your guess: 9



Yes, the number is 9

The magic number is between 0 and 100. To minimize the number of guesses, enter 50 first. If your guess is too high, the magic number is between 0 and 49. If your guess is too low, the magic number is between 51 and 100. So, you can eliminate half of the numbers from further consideration after one guess.

intelligent guess

How do you write this program? Do you immediately begin coding? No. It is important to *think before coding*. Think how you would solve the problem without writing a program. You need first to generate a random number between 0 and 100, inclusive, then to prompt the user to enter a guess, and then to compare the guess with the random number.

think before coding

It is a good practice to *code incrementally* one step at a time. For programs involving loops, if you don't know how to write a loop right away, you may first write the code for executing the loop one time, and then figure out how to repeatedly execute the code in a loop. For this program, you may create an initial draft, as shown in [Listing 4.1](#):

code incrementally

LISTING 4.1 GuessNumberOneTime.java

```
1 import java.util.Scanner;
2
3 public class GuessNumberOneTime {
4     public static void main(String[] args) {
5         // Generate a random number to be guessed
6         int number = (int)(Math.random() * 101);
7
8         Scanner input = new Scanner(System.in);
9         System.out.println("Guess a magic number between 0 and 100");
10
11        // Prompt the user to guess the number
12        System.out.print("\nEnter your guess: ");
13        int guess = input.nextInt();
```

generate a number

enter a guess

```

1 import java.util.Scanner;
2
3 public class GuessNumberOneTime {
4     public static void main(String[] args) {
5         // Generate a random number to be guessed
6         int number = (int)(Math.random() * 101);
7
8         Scanner input = new Scanner(System.in);
9         System.out.println("Guess a magic number between 0 and 100");
10
11        // Prompt the user to guess the number
12        System.out.print("\nEnter your guess: ");
13        int guess = input.nextInt();

```

generate a number
enter a guess

When you run this program, it prompts the user to enter a guess only once. To let the user enter a guess repeatedly, you may put the code in lines 11–20 in a loop as follows:

```

while (true) {
    // Prompt the user to guess the number
    System.out.print("\nEnter your guess: ");
    guess = input.nextInt();

    if (guess == number)
        System.out.println("Yes, the number is " + number);
    else if (guess > number)
        System.out.println("Your guess is too high");
    else
        System.out.println("Your guess is too low");
} // End of loop

```

This loop repeatedly prompts the user to enter a guess. However, this loop is not correct, because it never terminates. When **guess** matches **number**, the loop should end. So, the loop can be revised as follows:

```

while (guess != number) {
    // Prompt the user to guess the number
    System.out.print("\nEnter your guess: ");
    guess = input.nextInt();

    if (guess == number)
        System.out.println("Yes, the number is " + number);
    else if (guess > number)
        System.out.println("Your guess is too high");
    else
        System.out.println("Your guess is too low");
} // End of loop

```

The complete code is given in [Listing 4.2](#).

LISTING 4.2 GuessNumber.java

```
1 import java.util.Scanner;
2
3 public class GuessNumber {
4     public static void main(String[] args) {
5         // Generate a random number to be guessed
6         int number = (int)(Math.random() * 101);           generate a number
7
8         Scanner input = new Scanner(System.in);
9         System.out.println("Guess a magic number between 0 and 100");
10
11        int guess = -1;
12        while (guess != number) {
13
enter a guess          13            // Prompt the user to guess the number
14            System.out.print("\nEnter your guess: ");
15            guess = input.nextInt();
16
17            if (guess == number)
18                System.out.println("Yes, the number is " + number);
19            else if (guess > number)
20                System.out.println("Your guess is too high");
21            else
22                System.out.println("Your guess is too low");
23        } // End of loop
24    }
25 }
```



line#	number	guess	output
6	8		
11		-1	
iteration 1 { 15		50	Your guess is too high
20			
iteration 2 { 15		25	Your guess is too high
20			
iteration 3 { 15		12	Your guess is too high
20			
iteration 4 { 15		6	Your guess is too low
22			
iteration 5 { 15		9	Yes, the number is 9
20			

The program generates the magic number in line 6 and prompts the user to enter a guess continuously in a loop (lines 12–23). For each guess, the program checks whether the guess is correct, too high, or too low (lines 17–22). When the guess is correct, the program exits

the loop (line 12). Note that `guess` is initialized to `-1`. Initializing it to a value between `0` and `100` would be wrong, because that could be the number to be guessed.

4.2.2 Loop Design Strategies

Writing a correct loop is not an easy task for novice programmers. Consider three steps when writing a loop.

Step 1: Identify the statements that need to be repeated.

Step 2: Wrap these statements in a loop like this:

```
while (true) {  
    Statements;  
}
```

Step 3: Code the loop-continuation-condition and add appropriate statements for controlling the loop.

```
while (loop-continuation-condition) {  
    Statements;  
    Additional statements for controlling the loop;  
}
```

4.2.3 Problem: An Advanced Math Learning Tool

The Math subtraction learning tool program in [Listing 3.4](#), SubtractionQuiz.java, generates just one question for each run. You can use a loop to generate questions repeatedly. How do you write the code to generate five questions? Follow the loop design strategy. First identify the statements that need to be repeated. These are the statements for obtaining two random numbers, prompting the user with a subtraction question, and grading the question. Second, wrap the statements in a loop. Third, add a loop control variable and the loop-continuation-condition to execute the loop five times.



Video Note

Multiple subtraction quiz

[Listing 4.3](#) gives a program that generates five questions and, after a student answers all five, reports the number of correct answers. The program also displays the time spent on the test and lists all the questions.

LISTING 4.3 SubtractionQuizLoop.java

```
1 import java.util.Scanner;
2
3 public class SubtractionQuizLoop {
4     public static void main(String[] args) {
5         final int NUMBER_OF_QUESTIONS = 5; // Number of questions
6         int correctCount = 0; // Count the number of correct answers
7         int count = 0; // Count the number of questions
8         long startTime = System.currentTimeMillis(); get start time
9         String output = ""; // output string is initially empty
10        Scanner input = new Scanner(System.in);
11
12        while (count < NUMBER_OF_QUESTIONS) { loop
13            // 1. Generate two random single-digit integers
14            int number1 = (int)(Math.random() * 10);
15            int number2 = (int)(Math.random() * 10);
16
17            // 2. If number1 < number2, swap number1 with number2
18            if (number1 < number2) {
19                int temp = number1;
20                number1 = number2;
21                number2 = temp;
22            }
23
24            // 3. Prompt the student to answer "What is number1 - number2?"
25            System.out.print(display a question
26                "What is " + number1 + " - " + number2 + "? ");
27            int answer = input.nextInt();
28
29            // 4. Grade the answer and display the result
30            if (number1 - number2 == answer) { grade an answer
31                System.out.println("You are correct!");
32                correctCount++;
33            }
34            else
35                System.out.println("Your answer is wrong.\n" + number1
36                    + " - " + number2 + " should be " + (number1 - number2));
37
38            // Increase the count
39            count++;
40
41            prepare output
42            output += "\n" + number1 + "-" + number2 + "=" + answer +
43                ((number1 - number2 == answer) ? " correct" : " wrong");
44
45            end loop
46            long endTime = System.currentTimeMillis();
47            long testTime = endTime - startTime;
48
49            display result
50            System.out.println("Correct count is " + correctCount +
51                "\nTest time is " + testTime / 1000 + " seconds\n" + output);
52        }
53    }
54}
```



What is $9 - 2$? 7

You are correct!

What is $3 - 0$? 3

You are correct!

What is $3 - 2$? 1

You are correct!

What is $7 - 4$? 4

Your answer is wrong.

$7 - 4$ should be 3

What is $7 - 5$? 4

Your answer is wrong.

$7 - 5$ should be 2

Correct count is 3

Test time is 1021 seconds

$9-2=7$ correct

$3-0=3$ correct

$3-2=1$ correct

$7-4=4$ wrong

$7-5=4$ wrong

The program uses the control variable `count` to control the execution of the loop. `count` is initially `0` (line 7) and is increased by `1` in each iteration (line 39). A subtraction question is displayed and processed in each iteration. The program obtains the time before the test starts in line 8 and the time after the test ends in line 45, and computes the test time in line 46. The test time is in milliseconds and is converted to seconds in line 49.

4.2.4 Controlling a Loop with a Sentinel Value

Another common technique for controlling a loop is to designate a special value when reading and processing a set of values. This special input value, known as a *sentinel value*, signifies the end of the loop. A loop that uses a sentinel value to control its execution is called a *sentinel-controlled loop*.

sentinel value

[Listing 4.4](#) writes a program that reads and calculates the sum of an unspecified number of integers. The input `0` signifies the end of the input. Do you need to declare a new variable for each input value? No. Just use one variable named `data` (line 12) to store the input value and use a variable named `sum` (line 15) to store the total. Whenever a value is read, assign it to `data` and, if it is not zero, add it to `sum` (line 17).

LISTING 4.4 SentinelValue.java

```
1 import java.util.Scanner;
2
3 public class SentinelValue {
4     /** Main method */
5     public static void main(String[] args) {
6         // Create a Scanner
7         Scanner input = new Scanner(System.in);
8
9         // Read an initial data
10        System.out.print(
11            "Enter an int value (the program exits if the input is 0): ");
12        int data = input.nextInt();                                input
13
14        // Keep reading data until the input is 0
15        int sum = 0;
16        while (data != 0) {                                     loop
17            sum += data;
18
19            // Read the next data
20            System.out.print(
21                "Enter an int value (the program exits if the input is 0): ");
22            data = input.nextInt();
23        }                                                 end of loop
24
25        System.out.println("The sum is " + sum);                  display result
26    }
27 }
```



Enter an int value (the program exits if the input is 0):**2**

↙ Enter

Enter an int value (the program exits if the input is 0):**3**

↙ Enter

Enter an int value (the program exits if the input is 0):**4**

↙ Enter

Enter an int value (the program exits if the input is 0):**0**

↙ Enter

The sum is 9

line#	data	sum	output
12	2		
15		0	
iteration 1 {	17	2	
	22	3	
iteration 2 {	17	5	
	22	4	
iteration 3 {	17	9	
	22	0	
	25		The sum is 9



If **data** is not **0**, it is added to **sum** (line 17) and the next item of input data is read (lines 20–22). If **data** is **0**, the loop body is no longer executed and the **while** loop terminates. The input value **0** is the sentinel value for this loop. Note that if the first input read is **0**, the loop body never executes, and the resulting sum is **0**.



Caution

Don't use floating-point values for equality checking in a loop control. Since floating-point values are approximations for some values, using them could result in imprecise counter values and inaccurate results.

Consider the following code for computing `1 + 0.9 + 0.8 + ... + 0.1`:

```
double item = 1; double sum = 0;
while (item != 0) { // No guarantee item will be 0
    sum += item;
    item -= 0.1;
}
System.out.println(sum);
```

Variable `item` starts with `1` and is reduced by `0.1` every time the loop body is executed. The loop should terminate when `item` becomes `0`. However, there is no guarantee that item will be exactly `0`, because the floating-point arithmetic is approximated. This loop seems OK on the surface, but it is actually an infinite loop.

numeric error

4.2.5 Input and Output Redirections

In the preceding example, if you have a large number of data to enter, it would be cumbersome to type from the keyboard. You may store the data separated by whitespaces in a text file, say `input.txt`, and run the program using the following command:

```
java SentinelValue < input.txt
```

input redirection

This command is called *input redirection*. The program takes the input from the file `input.txt` rather than having the user to type the data from the keyboard at runtime. Suppose the contents of the file are

```
2 3 4 5 6 7 8 9 12 23 32
23 45 67 89 92 12 34 35 3 1 2 4 0
```

The program should get `sum` to be `518`.

output redirection

Similarly, there is *output redirection*, which sends the output to a file rather than displaying it on the console. The command for output redirection is:

```
java ClassName > output.txt
```

Input and output redirection can be used in the same command. For example, the following command gets input from `input.txt` and sends output to `output.txt`:

```
java SentinelValue < input.txt > output.txt
```

Please run the program and see what contents are in output.txt.

4.3 The do-while Loop

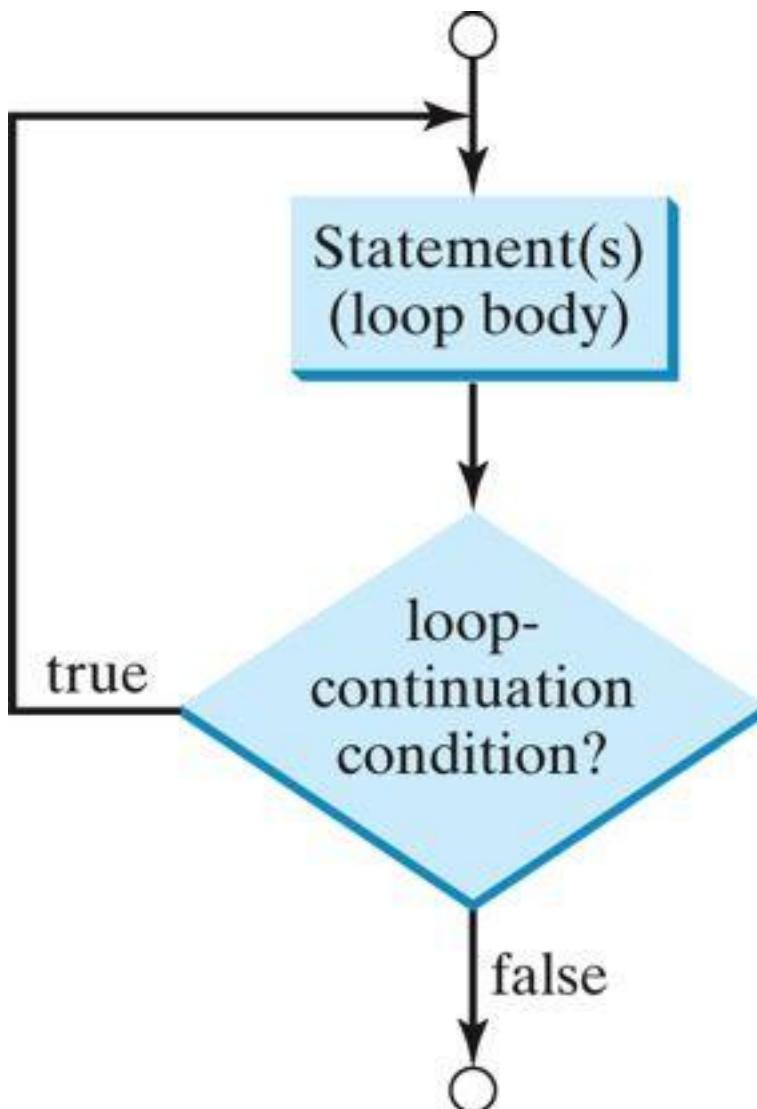
The **do-while** loop is a variation of the **while** loop. Its syntax is given below:

do-while loop

```
do {  
    // Loop body;  
    Statement(s);  
} while (loop-continuation-condition);
```

Its execution flow chart is shown in [Figure 4.2](#).

FIGURE 4.2 The do-while loop executes the loop body first, then checks the loop-continuation-condition to determine whether to continue or terminate the loop.



The loop body is executed first. Then the `loop-continuation-condition` is evaluated. If the evaluation is `true`, the loop body is executed again; if it is `false`, the `do-while` loop terminates. The difference between a `while` loop and a `do-while` loop is the order in which the `loop-continuation-condition` is evaluated and the loop body executed. The `while` loop and the `do-while` loop have equal expressive power. Sometimes one is a more convenient choice than the other. For example, you can rewrite the `while` loop in [Listing 4.4](#) using a `do-while` loop, as shown in [Listing 4.5](#):

LISTING 4.5 TestDoWhile.java

```

1 import java.util.Scanner;
2
3 public class TestDoWhile {
4     /** Main method */
5     public static void main(String[] args) {
6         int data;
7         int sum = 0;
8
9         // Create a Scanner
10        Scanner input = new Scanner(System.in);
11
12        // Keep reading data until the input is 0
13        do {                                loop
14            // Read the next data
15            System.out.print(
16                "Enter an int value (the program exits if the input is 0): ");
17            data = input.nextInt();
18
19            sum += data;
20        } while (data != 0);                  end loop
21
22        System.out.println("The sum is " + sum);
23    }
24 }

```



Enter an int value (the program exits if the input is 0): 3



Enter an int value (the program exits if the input is 0): 5



Enter an int value (the program exits if the input is 0): 6



Enter an int value (the program exits if the input is 0): 0



The sum is 14



Tip

Use the **do-while** loop if you have statements inside the loop that must be executed *at least once*, as in the case of the **do-while** loop in the preceding [TestDoWhile](#) program. These statements must appear before the loop as well as inside it if you use a **while loop**.

4.4 The for Loop

Often you write a loop in the following common form:

```
i = initialValue; // Initialize loop control variable
while (i < endValue) {
    // Loop body
    ...
    i++; // Adjust loop control variable
}
```

A **for** loop can be used to simplify the preceding loop:

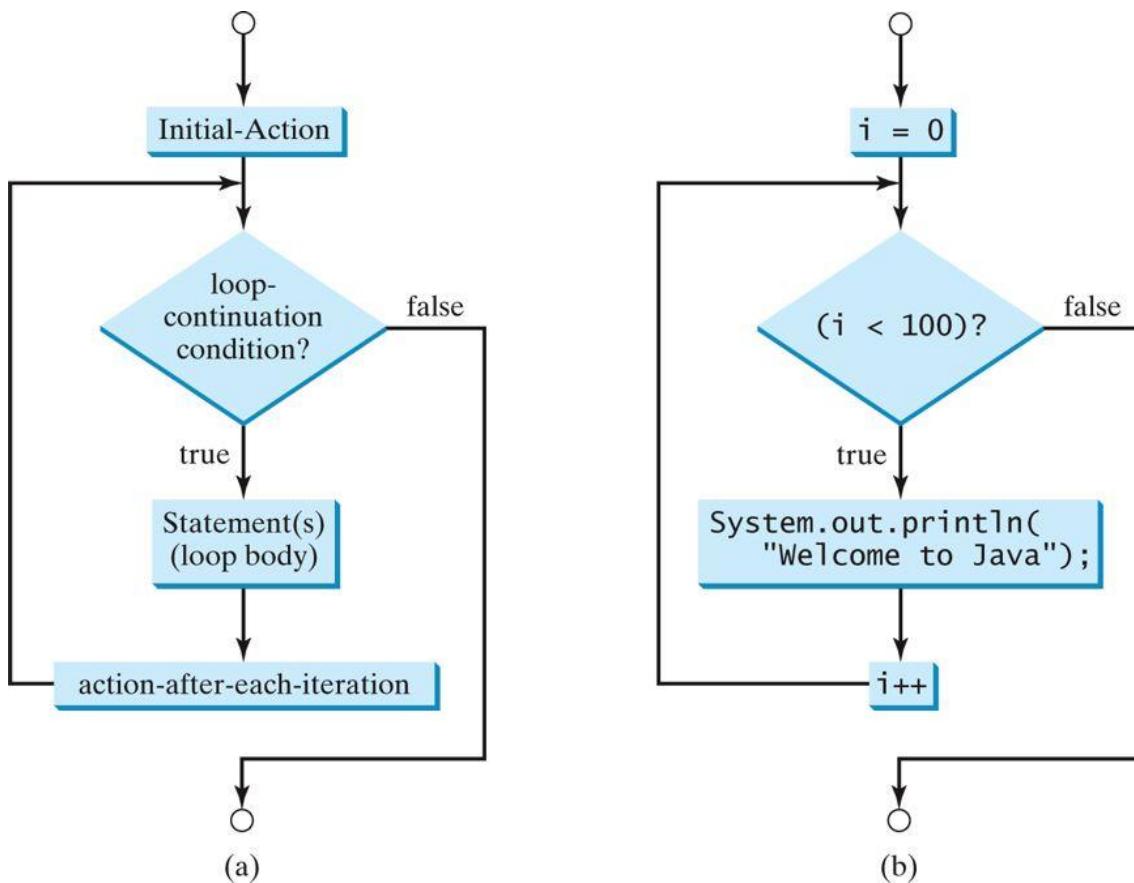
```
for (i = initialValue; i < endValue; i++) {
    // Loop body
    ...
}
```

In general, the syntax of a **for** loop is as shown below:

for loop	for (initial-action; loop-continuation-condition; action-after-each-iteration) { // Loop body; Statement(s); }
----------	---

The flow chart of the **for** loop is shown in [Figure 4.3\(a\)](#).

FIGURE 4.3 A for loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the loop-continuation-condition evaluates to true.



The **for** loop statement starts with the keyword **for**, followed by a pair of parentheses enclosing the control structure of the loop. This structure consists of **initial-action**, **loop-continuation-condition**, and **action-after-each-iteration**. The control structure is followed by the loop body enclosed inside braces. The **initial-action**, **loop-continuation-condition**, and **action-after-each-iteration** are separated by semicolons.

A **for** loop generally uses a variable to control how many times the loop body is executed and when the loop terminates. This variable is referred to as a *control variable*. The **initial-action** often initializes a control variable, the **action-after-each-iteration** usually increments or decrements the control variable, and the **loop-continuation-condition** tests whether the control variable has reached a termination value. For example, the following **for** loop prints **Welcome to Java!** a hundred times:

control variable

```

int i;
for (i = 0; i < 100; i++) {
    System.out.println("Welcome to Java!");
}

```

The flow chart of the statement is shown in [Figure 4.3\(b\)](#). The `for` loop initializes `i` to `0`, then repeatedly executes the `println` statement and evaluates `i++` while `i` is less than `100`.

The `initial-action`, `i = 0`, initializes the control variable, `i`. The `loop-continuation-condition`, `i < 100`, is a Boolean expression. The expression is evaluated right after the initialization and at the beginning of each iteration. If this condition is `true`, the loop body is executed. If it is `false`, the loop terminates and the program control turns to the line following the loop.

`initial-action`

The `action-after-each-iteration`, `i++`, is a statement that adjusts the control variable. This statement is executed after each iteration. It increments the control variable. Eventually, the value of the control variable should force the `loop-continuation-condition` to become `false`. Otherwise the loop is infinite.

`action-after-eachiteration`

The loop control variable can be declared and initialized in the for loop. Here is an example:

```
for (int i = 0; i < 100; i++) {  
    System.out.println("Welcome to Java!");  
}
```

If there is only one statement in the loop body, as in this example, the braces can be omitted.

omitting braces



Tip

The control variable must be declared inside the control structure of the loop or before the loop. If the loop control variable is used only in the loop, and not elsewhere, it is good programming practice to declare it in the `initial-action` of the `for` loop. If the variable is declared inside the loop control structure, it cannot be referenced outside the loop. In the preceding code, for example, you cannot reference `i` outside the `for` loop, because it is declared inside the `for` loop.

declare control variable

for loop variations



Note

The **initial-action** in a **for** loop can be a list of zero or more comma-separated variable declaration statements or assignment expressions. For example,

```
for (int i = 0, j = 0; (i + j < 10); i++, j++) {  
    // Do something  
}
```

The **action-after-each-iteration** in a **for** loop can be a list of zero or more commaseparated statements. For example,

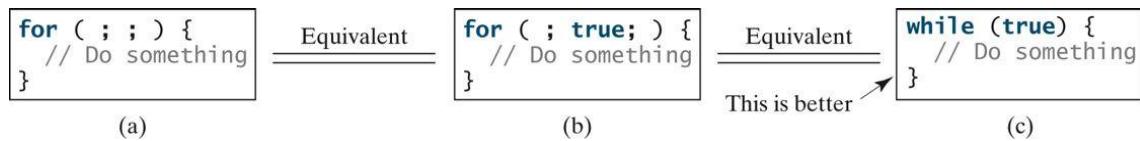
```
for (int i = 1; i < 100; System.out.println(i), i++);
```

This example is correct, but it is a bad example, because it makes the code difficult to read. Normally, you declare and initialize a control variable as an initial action and increment or decrement the control variable as an action after each iteration.



Note

If the **loop-continuation-condition** in a **for** loop is omitted, it is implicitly **true**. Thus the statement given below in (a), which is an infinite loop, is the same as in (b). To avoid confusion, though, it is better to use the equivalent loop in (c):



This is better

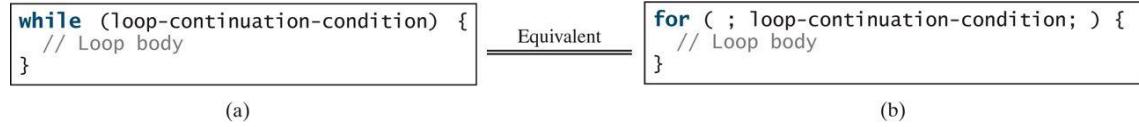
4.5 Which Loop to Use?

The **while** loop and **for** loop are called *pretest loops* because the continuation condition is checked before the loop body is executed. The **do-while** loop is called a *posttest loop* because the condition is checked after the loop body is executed. The three forms of loop statements, **while**, **do-while**, and **for**, are expressively equivalent; that is, you can write

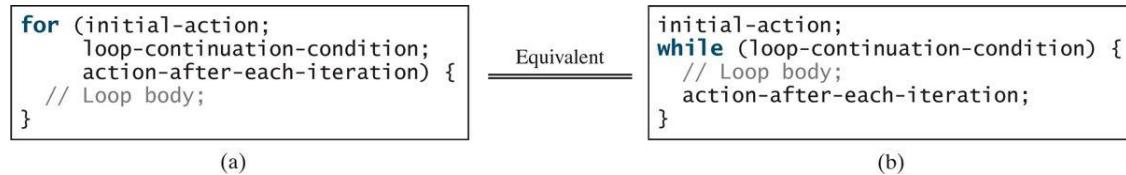
a loop in any of these three forms. For example, a **while** loop in (a) in the following figure can always be converted into the **for** loop in (b):

pretest loop

posttest loop



A **for** loop in (a) in the next figure can generally be converted into the **while** loop in (b) except in certain special cases (see Review Question 4.17 for such a case):

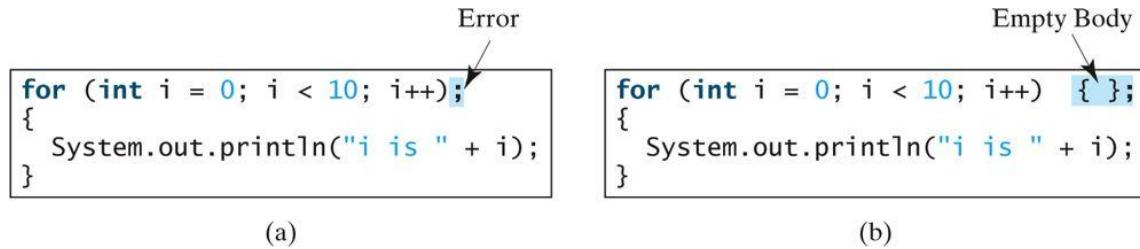


Use the loop statement that is most intuitive and comfortable for you. In general, a **for** loop may be used if the number of repetitions is known in advance, as, for example, when you need to print a message a hundred times. A **while** loop may be used if the number of repetitions is not fixed, as in the case of reading the numbers until the input is **0**. A **do-while** loop can be used to replace a **while** loop if the loop body has to be executed before the continuation condition is tested.



Caution

Adding a semicolon at the end of the **for** clause before the loop body is a common mistake, as shown below in (a). In (a), the semicolon signifies the end of the loop prematurely. The loop body is actually empty, as shown in (b). (a) and (b) are equivalent.(b)



Similarly, the loop in (c) is also wrong. (c) is equivalent to (d).

```
int i = 0;  
while (i < 10);  
{  
    System.out.println("i is " + i);  
    i++;  
}
```

(c)

```
int i = 0;  
while (i < 10) {};  
{  
    System.out.println("i is " + i);  
    i++;  
}
```

(d)

These errors often occur when you use the next-line block style. Using the end-of-line block style can avoid errors of this type.

In the case of the **do-while** loop, the semicolon is needed to end the loop.

```
int i = 0;  
do {  
    System.out.println("i is " + i);  
    i++;  
} while (i < 10);
```

Correct

4.6 Nested Loops

Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, and started anew.

[Listing 4.6](#) presents a program that uses nested **for** loops to print a multiplication table.

LISTING 4.6 MultiplicationTable.java

```

1 public class MultiplicationTable {
2     /** Main method */
3     public static void main(String[] args) {
4         // Display the table heading
5         System.out.println("Multiplication Table");           table title
6
7         // Display the number title
8         System.out.print("    ");
9         for (int j = 1; j <= 9; j++) {
10            System.out.print("    " + j);
11
12        System.out.println("\n-----");
13
14        // Print table body
15        for (int i = 1; i <= 9; i++) {
16            System.out.print(i + " | ");
17            for (int j = 1; j <= 9; j++) {
18                // Display the product and align properly
19                System.out.printf("%4d", i * j);
20            }
21            System.out.println()
22        }
23    }
24 }
```



Multiplication Table

1

2

3

4

5

6

7

8

9

1 |

1

2

3

4

5

6

7

8

9

2 |

2

4

6

8

10

12

14

16

18

3 |

3

6

9

12

15

18

21

24

27

4 |

4

8

12

16

20

24

28

32

36

5 |

5

10

15

20

25

30

35

40

45

6 |

6

12

18

24

30

36

42

48

54

7 |

7

14

21

28

35

42

49

56

63

8 |

8

16

24

32

40

48

56

64

72

9 |

9

18

27

36

45

54

63

The program displays a title (line 5) on the first line in the output. The first `for` loop (lines 9–10) displays the numbers `1` through `9` on the second line. A dash (`-`) line is displayed on the third line (line 12).

The next loop (lines 15–22) is a nested `for` loop with the control variable `i` in the outer loop and `j` in the inner loop. For each `i`, the product `i * j` is displayed on a line in the inner loop, with `j` being `1, 2, 3,..., 9`.

4.7 Minimizing Numeric Errors

Numeric errors involving floating-point numbers are inevitable. This section discusses how to minimize such errors through an example.



Video Note

Minimize numeric errors

[Listing 4.7](#) presents an example summing a series that starts with `0.01` and ends with `1.0`. The numbers in the series will increment by `0.01`, as follows: `0.01 + 0.02 + 0.03` and so on.

LISTING 4.7 TestSum.java

```

1 public class TestSum {
2     public static void main(String[] args) {
3         // Initialize sum
4         float sum = 0;
5
6         // Add 0.01, 0.02, ..., 0.99, 1 to sum
7         for (float i = 0.01f; i <= 1.0f; i = i + 0.01f)
8             sum += i;
9
10        // Display result
11        System.out.println("The sum is " + sum);
12    }
13 }
```



The sum is 50.499985

The `for` loop (lines 7–8) repeatedly adds the control variable `i` to `sum`. This variable, which begins with `0.01`, is incremented by `0.01` after each iteration. The loop terminates when `i` exceeds `1.0`.

The `for` loop initial action can be any statement, but it is often used to initialize a control variable. From this example, you can see that a control variable can be a `float` type. In fact, it can be any data type.

The exact `sum` should be `50.50`, but the answer is `50.499985`. The result is imprecise because computers use a fixed number of bits to represent floating-point numbers, and thus they cannot represent some floating-point numbers exactly. If you change `float` in the program to `double`, as follows, you should see a slight improvement in precision, because a `double` variable takes 64 bits, whereas a `float` variable takes 32.

double precision

```
// Initialize sum
double sum = 0;

// Add 0.01, 0.02, ..., 0.99, 1 to sum
for (double i = 0.01; i <= 1.0; i = i + 0.01)
    sum += i;
```

However, you will be stunned to see that the result is actually `49.5000000000003`. What went wrong? If you print out `i` for each iteration in the loop, you will see that the last `i` is slightly larger than `1` (not exactly `1`). This causes the last `i` not to be added into `sum`. The fundamental problem is that the floating-point numbers are represented by approximation. To fix the problem, use an integer count to ensure that all the numbers are added to `sum`. Here is the new loop:

numeric error

```
double currentValue = 0.01;  
  
for (int count = 0; count < 100; count++) {  
    sum += currentValue;  
    currentValue += 0.01;  
}
```

After this loop, `sum` is `50.5000000000003`. This loop adds the numbers from small to big. What happens if you add numbers from big to small (i.e., `1.0, 0.99, 0.98, ..., 0.02, 0.01` in this order) as follows:

```
double currentValue = 1.0;  
  
for (int count = 0; count < 100; count++) {  
    sum += currentValue;  
    currentValue -= 0.01;  
}
```

After this loop, `sum` is `50.4999999999995`. Adding from big to small is less accurate than adding from small to big. This phenomenon is an artifact of the finite-precision arithmetic. Adding a very small number to a very big number can have no effect if the result requires more precision than the variable can store. For example, the inaccurate result of `100000000.0 + 0.00000001` is `100000000.0`. To obtain more accurate results, carefully select the order of computation. Adding the smaller numbers before the big numbers is one way to minimize error.

avoiding numeric error

4.8 Case Studies

Loops are fundamental in programming. The ability to write loops is essential in learning Java programming. *If you can write programs using loops, you know how to program!* For this reason, this section presents three additional examples of solving problems using loops.

4.8.1 Problem: Finding the Greatest Common Divisor

The greatest common divisor of two integers `4` and `2` is `2`. The greatest common divisor of two integers `16` and `24` is `8`. How do you find the greatest common divisor? Let the two input integers be `n1` and `n2`. You know that number `1` is a common divisor, but it may not

be the greatest common divisor. So, you can check whether **k** (for **k = 2, 3, 4**, and so on) is a common divisor for **n1** and **n2**, until **k** is greater than **n1** or **n2**. Store the common divisor in a variable named **gcd**. Initially, **gcd** is **1**. Whenever a new common divisor is found, it becomes the new **gcd**. When you have checked all the possible common divisors from **2** up to **n1** or **n2**, the value in variable **gcd** is the greatest common divisor. The idea can be translated into the following loop:

```
gcd
int gcd = 1; // Initial gcd is 1
int k = 2; // Possible gcd
while (k <= n1 && k <= n2) {
    if (n1 % k == 0 && n2 % k == 0)
        gcd = k; // Update gcd
    k++; // Next possible gcd
}
// After the loop, gcd is the greatest common divisor for
n1 and n2
```

[Listing 4.8](#) presents the program that prompts the user to enter two positive integers and finds their greatest common divisor.

LISTING 4.8 GreatestCommonDivisor.java

```
1 import java.util.Scanner;
2
3 public class GreatestCommonDivisor {
4     /** Main method */
5     public static void main(String[] args) {
6         // Create a Scanner
7         Scanner input = new Scanner(System.in);
8
9         // Prompt the user to enter two integers
10        System.out.print("Enter first integer: ");
11        int n1 = input.nextInt();
12        System.out.print("Enter second integer: ");
13        int n2 = input.nextInt();
14
15        int gcd = 1; // Initial gcd is 1
16        int k = 2; // Possible gcd
17        while (k <= n1 && k <= n2) {
18            if (n1 % k == 0 && n2 % k == 0)
19                gcd = k; // Update gcd
20            k++;
21        }
22
23        System.out.println("The greatest common divisor for " + n1 +
24                    " and " + n2 + " is " + gcd);
25    }
26 }
```



Enter first integer: 125

← Enter

Enter second integer: 2525

← Enter

The greatest common divisor for 125 and 2525 is 25

How did you write this program? Did you immediately begin to write the code? No. It is important to *think before you type*. Thinking enables you to generate a logical solution for the problem without concern about how to write the code. Once you have a logical solution, type think before you type the code to translate the solution into a Java program. The translation is not unique. For example, you could use a **for** loop to rewrite the code as follows:

think before you type

```
for (int k = 2; k <= n1 && k <= n2; k++) {  
    if (n1 % k == 0 && n2 % k == 0)  
        gcd = k;  
}
```

A problem often has multiple solutions. The gcd problem can be solved in many ways. Exercise 4.15 suggests another solution. A more efficient solution is to use the classic Euclidean algorithm. See <http://www.cut-the-knot.org/blue/Euclid.shtml> for more information.

multiple solutions

You might think that a divisor for a number **n1** cannot be greater than **n1 / 2**. So you would attempt to improve the program using the following loop:

erroneous solutions

```
for (int k = 2; k <= n1 / 2 && k <= n2 / 2; k++) {  
    if (n1 % k == 0 && n2 % k == 0)  
        gcd = k;  
}
```

This revision is wrong. Can you find the reason? See Review Question 4.14 for the answer.

4.8.2 Problem: Predicating the Future Tuition

Suppose that the tuition for a university is \$10,000 this year and tuition increases 7% every year. In how many years will the tuition be doubled?

Before you can write a program to solve this problem, first consider how to solve it by hand. The tuition for the second year is the tuition for the first year *1.07. The tuition for a future year is the tuition of its preceding year *1.07. So, the tuition for each year can be computed as follows:

```
double tuition = 10000;    int year = 1    // Year 1
tuition = tuition * 1.07;  year++;           // Year 2
tuition = tuition * 1.07;  year++;           // Year 3
tuition = tuition * 1.07;  year++;           // Year 4
...
...
```

Keep computing tuition for a new year until it is at least 20000. By then you will know how many years it will take for the tuition to be doubled. You can now translate the logic into the following loop:

```
double tuition = 10000;    // Year 1
int year = 1;
while (tuition < 20000) {
    tuition = tuition * 1.07;
    year++;
}
```

The complete program is shown in [Listing 4.9](#).

LISTING 4.9 FutureTuition.java

```
1 public class FutureTuition {
2     public static void main(String[] args) {
3         double tuition = 10000;    // Year 1
4         int year = 1;
5         while (tuition < 20000) {           loop
next year's tuition          6             tuition = tuition * 1.07;
                                7             year++;
                                8         }
                                9
                                10            System.out.println("Tuition will be doubled in "
                                11                + year + " years");
                                12        }
                                13 }
```



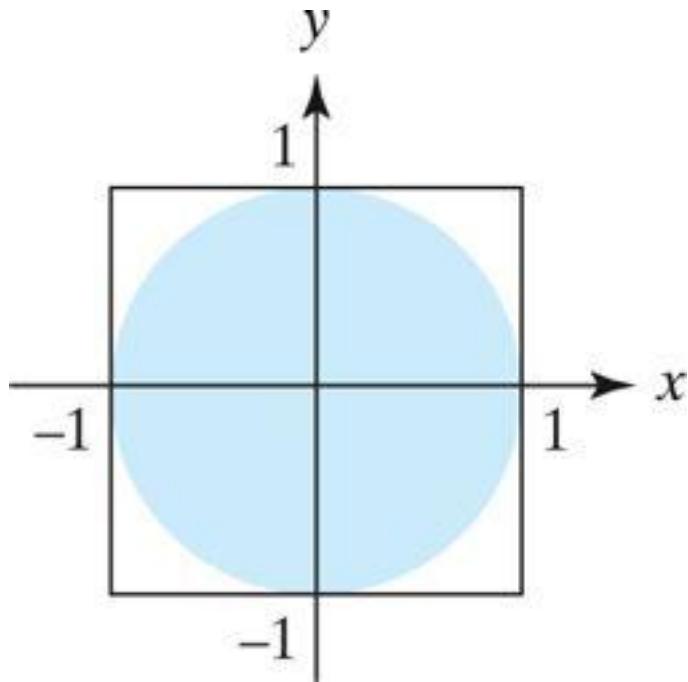
Tuition will be doubled in 12 years

The `while` loop (lines 5–8) is used to repeatedly compute the tuition for a new year. The loop terminates when tuition is greater than or equal to 20000.

4.8.3 Problem: Monte Carlo Simulation

Monte Carlo simulation uses random numbers and probability to solve problems. This method has a wide range of applications in computational mathematics, physics, chemistry, and finance. This section gives an example of using Monte Carlo simulation for estimating

To estimate π using the Monte Carlo method, draw a circle with its bounding square as shown below.



Assume the radius of the circle is 1. So, the circle area is π and the square area is 4. Randomly generate a point in the square. The probability for the point to fall in the circle is `circleArea / squareArea = $\pi/4$` .

Write a program that randomly generates 1000000 points in the square and let `numberOfHits` denote the number of points that fall in the circle. So, `numberOfHits` is approximately `1000000 * ($\pi/4$)`. can be approximated as `4 * numberOfHits / 1000000`. The complete program is shown in [Listing 4.10](#).

LISTING 4.10 MonteCarloSimulation.java

```

1 public class MonteCarloSimulation {
2     public static void main(String[] args) {
3         final int NUMBER_OF_TRIALS = 10000000;
4         int numberOfHits = 0;
5
6         generate random points
7         for (int i = 0; i < NUMBER_OF_TRIALS; i++) {
8             double x = Math.random() * 2.0 - 1;
9             double y = Math.random() * 2.0 - 1;
10            if (x * x + y * y <= 1)
11                numberOfHits++;
12
13        double pi = 4.0 * numberOfHits / NUMBER_OF_TRIALS;
14
15        System.out.println("PI is " + pi);
16    }

```



PI is 3.14124

The program repeatedly generates a random point (x, y) in the square in lines 7–8:

```

double x = Math.random() * 2.0 - 1 ;
double y = Math.random() * 2.0 - 1 ;

```

If $x^2 + y^2 \leq 1$, the point is inside the circle and `numberOfHits` is incremented by 1. π is approximately `4 * numberOfHits / NUMBER_OF_TRIALS` (line 13).

4.9 Keywords `break` and `continue`



Pedagogical Note

Two keywords, `break` and `continue`, can be used in loop statements to provide additional controls. Using `break` and `continue` can simplify programming in some cases. Overusing or improperly using them, however, can make programs difficult to read and debug. (Note to instructors: You may skip this section without affecting the rest of the book.)

You have used the keyword **break** in a **switch** statement. You can also use **break** in a loop to immediately terminate the loop. [Listing 4.11](#) presents a program to demonstrate the effect of using **break** in a loop.

break

LISTING 4.11 TestBreak.java

```
1 public class TestBreak {  
2     public static void main(String[] args) {  
3         int sum = 0;  
4         int number = 0;  
5  
6         while (number < 20) {  
7             number++;  
8             sum += number;  
9             if (sum >= 100)  
10                break;                                break  
11        }  
12        System.out.println("The number is " + number);  
13        System.out.println("The sum is " + sum);  
14    }  
15}  
16 }
```



The number is 14

The sum is 105

The program in [Listing 4.11](#) adds integers from **1** to **20** in this order to **sum** until **sum** is greater than or equal to **100**. Without the **if** statement (line 9), the program calculates the sum of the numbers from **1** to **20**. But with the **if** statement, the loop terminates when **sum** becomes greater than or equal to **100**. Without the **if** statement, the output would be:



The number is 20

The sum is 210

You can also use the **continue** keyword in a loop. When it is encountered, it ends the current iteration. Program control goes to the end of the loop body. In other words,

`continue` breaks out of an iteration while the `break` keyword breaks out of a loop. Listing 4.12 presents a program to demonstrate the effect of using `continue` in a loop.

`continue`

LISTING 4.12 TestContinue.java

```
1 public class TestContinue {  
2     public static void main(String[] args) {  
3         int sum = 0;  
4         int number = 0;  
5  
6         while (number < 20) {  
7             number++;  
8             if (number == 10 || number == 11)  
9                 continue;  
10            sum += number;  
11        }  
12  
13        System.out.println("The sum is " + sum);  
14    }  
15 }
```



The sum is 189

The program in Listing 4.12 adds integers from 1 to 20 except 10 and 11 to `sum`. With the `if` statement in the program (line 8), the `continue` statement is executed when `number` becomes 10 or 11. The `continue` statement ends the current iteration so that the rest of the statement in the loop body is not executed; therefore, `number` is not added to `sum` when it is 10 or 11. Without the `if` statement in the program, the output would be as follows:



The sum is 210

In this case, all of the numbers are added to `sum`, even when `number` is 10 or 11. Therefore, the result is 210, which is 21 more than it was with the `if` statement.



Note

The `continue` statement is always inside a loop. In the `while` and `do-while` loops, the `loop-continuation-condition` is evaluated immediately after the `continue` statement. In the `for` loop, the `action-after-each-iteration` is performed, then the `loop-continuation-condition` is evaluated, immediately after the `continue` statement.

You can always write a program without using `break` or `continue` in a loop. See Review Question 4.18. In general, using `break` and `continue` is appropriate only if it simplifies coding and makes programs easier to read.

[Listing 4.2](#) gives a program for guessing a number. You can rewrite it using a `break` statement, as shown in [Listing 4.13](#).

LISTING 4.13 GuessNumberUsingBreak.java

```
1 import java.util.Scanner;
2
3 public class GuessNumberUsingBreak {
4     public static void main(String[] args) {
5         // Generate a random number to be guessed
6         int number = (int)(Math.random() * 101);           generate a number
7
8         Scanner input = new Scanner(System.in);
9         System.out.println("Guess a magic number between 0 and 100");
10
11        while (true) {                                loop continuously
12            // Prompt the user to guess the number
13            System.out.print("\nEnter your guess: ");      enter a guess
14            int guess = input.nextInt();
15
16            if (guess == number) {
17                System.out.println("Yes, the number is " + number);
18                break;                                     break
19            }
20            else if (guess > number)
21                System.out.println("Your guess is too high");
22            else
23                System.out.println("Your guess is too low");
24        } // End of loop
25    }
26 }
```

Using the `break` statement makes this program simpler and easier to read. However, you should use `break` and `continue` with caution. Too many `break` and `continue` statements will produce a loop with many exit points and make the program difficult to read.



Note

Some programming languages have a `goto` statement. The `goto` statement indiscriminately transfers control to any statement in the program and executes it. This makes your program vulnerable to errors. The `break` and `continue` statements in Java are different from `goto` statements. They operate only in a loop or a `switch` statement. The `break` statement breaks out of the loop, and the `continue` statement breaks out of the current iteration in the loop.

`goto`

4.9.1 Problem: Displaying Prime Numbers

An integer greater than `1` is *prime* if its only positive divisor is `1` or itself. For example, `2`, `3`, `5`, and `7` are prime numbers, but `4`, `6`, `8`, and `9` are not.

The problem is to display the first 50 prime numbers in five lines, each of which contains ten numbers. The problem can be broken into the following tasks:

- Determine whether a given number is prime.
- For `number 2, 3, 4, 5, 6`, test whether it is prime.
- Count the prime numbers.
- Print each prime number, and print ten numbers per line.

Obviously, you need to write a loop and repeatedly test whether a new `number` is prime. If the `number` is prime, increase the count by `1`. The `count` is `0` initially. When it reaches `50`, the loop terminates.

Here is the algorithm for the problem:

Set the number of prime numbers to be printed as a constant NUMBER_OF_PRIMES;
Use count to track the number of prime numbers and set an initial count to 0;
Set an initial number to 2;

```
while (count < NUMBER_OF_PRIMES) {  
    Test whether number is prime;  
  
    if number is prime {  
        Print the prime number and increase the count;  
    }  
  
    Increment number by 1;  
}
```

To test whether a number is prime, check whether it is divisible by 2, 3, 4, up to $\text{number}/2$. If a divisor is found, the number is not a prime. The algorithm can be described as follows:

```
Use a boolean variable isPrime to denote whether  
the number is prime; Set isPrime to true initially;  
for (int divisor = 2; divisor <= number / 2; divisor++) {  
    if (number % divisor == 0) {  
        Set isPrime to false  
        Exit the loop;  
    }  
}
```

The complete program is given in [Listing 4.14](#).

LISTING 4.14 PrimeNumber.java

```

1 public class PrimeNumber {
2     public static void main(String[] args) {
3         final int NUMBER_OF_PRIMES = 50; // Number of primes to display
4         final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
5         int count = 0; // Count the number of prime numbers
6         int number = 2; // A number to be tested for primeness
7
8         System.out.println("The first 50 prime numbers are \n");
9
10        // Repeatedly find prime numbers
11        while (count < NUMBER_OF_PRIMES) {
12            // Assume the number is prime
13            boolean isPrime = true; // Is the current number prime?
14
15            // Test whether number is prime
16            for (int divisor = 2; divisor <= number / 2; divisor++) {
17                if (number % divisor == 0) { // If true, number is not prime
18                    isPrime = false; // Set isPrime to false
19                    break; // Exit the for loop
20                }
21            }
22
23            // Print the prime number and increase the count
24            if (isPrime) {
25                count++; // Increase the count
26
27            if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
28                // Print the number and advance to the new line
29                System.out.println(number);
30            }
31            else
32                System.out.print(number + " ");
33        }
34
35        // Check if the next number is prime
36        number++;
37    }
38 }
39 }
```



The first 50 prime numbers are

```

2 3 5 7 11 13 17 19 23 29
31 37 41 43 47 53 59 61 67 71
73 79 83 89 97 101 103 107 109 113
127 131 137 139 149 151 157 163 167 173
```

```
179 181 191 193 197 199 211 223 227 229
```

This is a complex program for novice programmers. The key to developing a programmatic solution to this problem, and to many other problems, is to break it into subproblems and develop solutions for each of them in turn. Do not attempt to develop a complete solution in the first trial. Instead, begin by writing the code to determine whether a given number is prime, then expand the program to test whether other numbers are prime in a loop.

subproblem

To determine whether a number is prime, check whether it is divisible by a number between **2** and **number/2** inclusive (line 16). If so, it is not a prime number (line 18); otherwise, it is a prime number. For a prime number, display it. If the count is divisible by **10** (lines 27–30), advance to a new line. The program ends when the count reaches **50**.

The program uses the **break** statement in line 19 to exit the **for** loop as soon as the number is found to be a nonprime. You can rewrite the loop (lines 16–21) without using the **break** statement, as follows:

```
for (int divisor = 2; divisor <= number / 2 && isPrime;
     divisor++) {
    // If true, the number is not prime
    if (number % divisor == 0) {
        // Set isPrime to false, if the number is not prime
        isPrime = false;
    }
}
```

However, using the **break** statement makes the program simpler and easier to read in this case.

4.10 (GUI) Controlling a Loop with a Confirmation Dialog

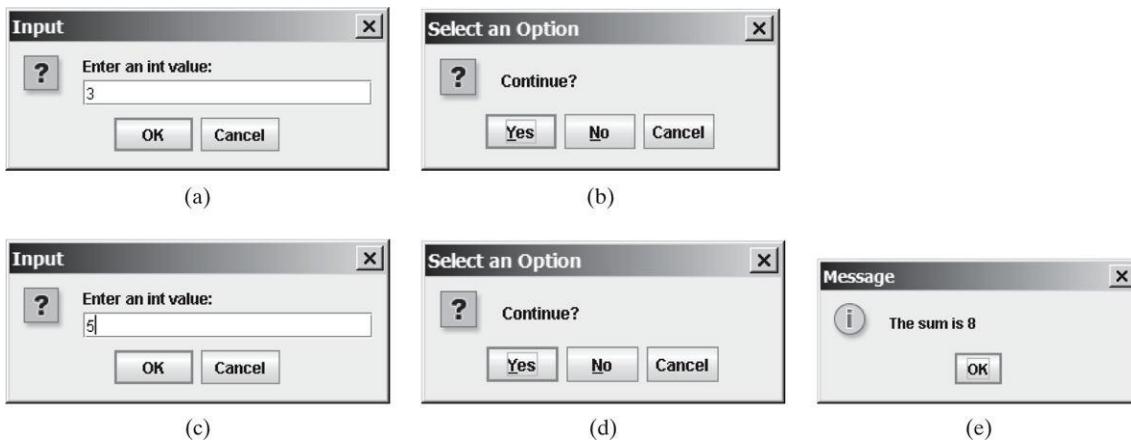
A sentinel-controlled loop can be implemented using a confirmation dialog. The answers *Yes* or *No* continue or terminate the loop. The template of the loop may look as follows:

confirmation dialog

```
int option = JOptionPane.YES_OPTION;
while (option == JOptionPane.YES_OPTION) {
    System.out.println("continue loop");
    option = JOptionPane.showConfirmDialog(null,
    "Continue?");
```

[Listing 4.15](#) rewrites [Listing 4.4](#), `SentinelValue.java`, using a confirmation dialog box. A sample run is shown in [Figure 4.4](#).

FIGURE 4.4 The user enters 3 in (a), clicks Yes in (b), enters 5 in (c), clicks No in (d), and the result is shown in (e).



LISTING 4.15 SentinelValueUsingConfirmationDialog.java

```
1 import javax.swing.JOptionPane;
2
3 public class SentinelValueUsingConfirmationDialog {
4     public static void main(String[] args) {
5         int sum = 0;
6
7         // Keep reading data until the user answers No
8         int option = JOptionPane.YES_OPTION;
9         while (option == JOptionPane.YES_OPTION) {
10             // Read the next data
11             String dataString = JOptionPane.showInputDialog(
12                 "Enter an int value: ");
13             int data = Integer.parseInt(dataString);
14
15             sum += data;
16
17             option = JOptionPane.showConfirmDialog(null, "Continue?");
18         }
19
20         JOptionPane.showMessageDialog(null, "The sum is " + sum);
21     }
22 }
```

A program displays an input dialog to prompt the user to enter an integer (line 11) and adds it to `sum` (line 15). Line 17 displays a confirmation dialog to let the user decide whether to continue the input. If the user clicks Yes, the loop continues; otherwise the loop exits. Finally the program displays the result in a message dialog box (line 20).

KEY TERMS

break statement [136](#)

continue statement [136](#)

do -while loop [124](#)

for loop [126](#)

loop control structure [127](#)

infinite loop [117](#)

input redirection [124](#)

iteration [116](#)

labeled continue statement [136](#)

loop [116](#)

loop-continuation-condition [116](#)

loop body [116](#)

nested loop [129](#)

off-by-one error [124](#)

output redirection [124](#)

sentinel value [122](#)

while loop [116](#)

CHAPTER SUMMARY

1. There are three types of repetition statements: the **while** loop, the **do-while** loop, and the **for** loop.

2. The part of the loop that contains the statements to be repeated is called the *loop body*.

3. A one-time execution of a loop body is referred to as an *iteration of the loop*.
4. An infinite loop is a loop statement that executes infinitely.
5. In designing loops, you need to consider both the loop control structure and the loop body.
6. The **while** loop checks the **loop-continuation-condition** first. If the condition is **true**, the loop body is executed; if it is **false**, the loop terminates.
7. The **do-while** loop is similar to the **while** loop, except that the **do-while** loop executes the loop body first and then checks the **loop-continuation-condition** to decide whether to continue or to terminate.
8. Since the **while** loop and the **do-while** loop contain the **loop-continuation-condition**, which is dependent on the loop body, the number of repetitions is determined by the loop body. The **while** loop and the **do-while** loop often are used when the number of repetitions is unspecified.
9. A *sentinel value* is a special value that signifies the end of the loop.
10. The **for** loop generally is used to execute a loop body a predictable number of times; this number is not determined by the loop body.
11. The **for** loop control has three parts. The first part is an initial action that often initializes a control variable. The second part, the loop-continuation-condition, determines whether the loop body is to be executed. The third part is executed after each iteration and is often used to adjust the control variable. Usually, the loop control variables are initialized and changed in the control structure.
12. The **while** loop and **for** loop are called *pretest loops* because the continuation condition is checked before the loop body is executed.
13. The **do-while** loop is called *posttest loop* because the condition is checked after the loop body is executed.
14. Two keywords, **break** and **continue**, can be used in a loop.
15. The **break** keyword immediately ends the innermost loop, which contains the break.
16. The **continue** keyword only ends the current iteration.

REVIEW QUESTIONS

Sections 4.2–4.4

4.1 Analyze the following code. Is `count < 100` always `true`, always `false`, or sometimes `true` or sometimes `false` at Point A, Point B, and Point C?

```
int count = 0;
while (count < 100) {
    // Point A
    System.out.println("Welcome to Java!\\n");
    count++;
    // Point B
}
// Point C
```

4.2 What is wrong if `guess` is initialized to `0` in line 11 in [Listing 4.2](#)?

4.3 How many times is the following loop body repeated? What is the printout of the loop?

```
int i = 1;
while (i < 10)
    if (i % 2 == 0)
        System.out.println(i);
```

(a)

```
int i = 1;
while (i < 10)
    if (i % 2 == 0)
        System.out.println(i++);
```

(b)

```
int i = 1;
while (i < 10)
    if ((i++) % 2 == 0)
        System.out.println(i);
```

(c)

4.4 What are the differences between a `while` loop and a `do-while` loop? Convert the following `while` loop into a `do-while` loop.

```
int sum = 0;
int number = input.nextInt();
while (number != 0) {
    sum += number;
    number = input.nextInt();
}
```

4.5 Do the following two loops result in the same value in `sum`?

```
for (int i = 0; i < 10; ++i) {
    sum += i;
}
```

(a)

```
for (int i = 0; i < 10; i++) {
    sum += i;
}
```

(b)

4.6 What are the three parts of a `for` loop control? Write a `for` loop that prints the numbers from `1` to `100`.

4.7 Suppose the input is **23450**. What is the output of the following code?

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int number, max;
        number = input.nextInt();
        max = number;
        while (number != 0) {
            number = input.nextInt();
            if (number > max)
                max = number;
        }
        System.out.println("max is " + max);
        System.out.println("number " + number);
    }
}
```

4.8 Suppose the input is **23450**. What is the output of the following code?

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int number, sum = 0, count;
        for (count = 0; count < 5; count++) {
            number = input.nextInt();
            sum += number;
        }
        System.out.println("sum is " + sum);
        System.out.println("count is " + count);
    }
}
```

4.9 Suppose the input is **2 3 4 5 0**. What is the output of the following code?

```
import java.util.Scanner;
public class Test {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int number, max;
        number = input.nextInt();
        max = number;
        do {
            number = input.nextInt();
            if (number > max)
                max = number;
        } while (number != 0);
        System.out.println("max is " + max);
    }
}
```

```

        System.out.println("number " + number);
    }
}

```

4.10 What does the following statement do?

```

for ( ; ; ) {
    do something;
}

```

4.11 If a variable is declared in the **for** loop control, can it be used after the loop exits?

4.12 Can you convert a **for** loop to a **while** loop? List the advantages of using **for** loops.

4.13 Convert the following **for** loop statement to a **while** loop and to a **do-while** loop:

```

long sum = 0;
for (int i = 0; i <= 1000; i++)
    sum = sum + i;

```

4.14 Will the program work if **n1** and **n2** are replaced by **n1 / 2** and **n2 / 2** in line 17 in [Listing 4.8](#)?

Section 4.9

4.15 What is the keyword **break** for? What is the keyword **continue** for? Will the following program terminate? If so, give the output.

```

int balance = 1000;
while (true) {
    if (balance < 9)
        break;
    balance = balance - 9;
}

System.out.println("Balance is "
    + balance);

```

(a)

```

int balance = 1000;
while (true) {
    if (balance < 9)
        continue;
    balance = balance - 9;
}

System.out.println("Balance is "
    + balance);

```

(b)

4.16 Can you always convert a **while** loop into a **for** loop? Convert the following **while** loop into a **for** loop.

```

int i = 1;
int sum = 0;
while (sum < 10000) {

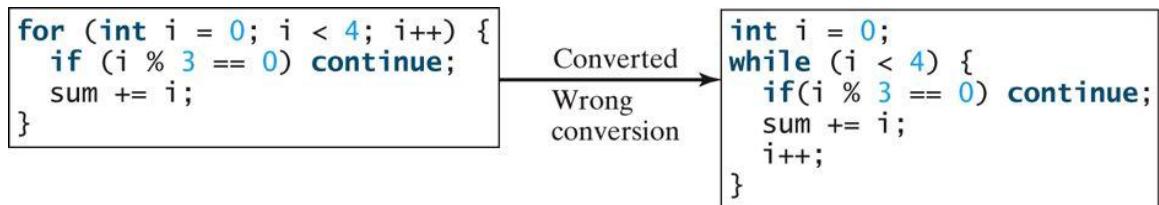
```

```

    sum = sum + i;
    i++;
}

```

- 4.17** The **for** loop on the left is converted into the **while** loop on the right. What is wrong? Correct it.



- 4.18** Rewrite the programs **TestBreak** and **TestContinue** in [Listings 4.11](#) and [4.12](#) without using **break** and **continue**.

- 4.19** After the **break** statement is executed in the following loop, which statement is executed? Show the output.

```

for (int i = 1; i < 4; i++) {
    for (int j = 1; j < 4; j++) {
        if (i * j > 2)
            break;
        System.out.println(i * j);
    }
    System.out.println(i);
}

```

- 4.20** After the **continue** statement is executed in the following loop, which statement is executed? Show the output.

```

for (int i = 1; i < 4; i++) {
    for (int j = 1; j < 4; j++) {
        if (i * j > 2)
            continue;
        System.out.println(i * j);
    }
    System.out.println(i);
}

```

Comprehensive

- 4.21** Identify and fix the errors in the following code:

```

1 public class Test {
2     public void main(String[] args) {
3         for (int i = 0; i < 10; i++);
4             sum += i;

```

```

5
6     if ( i < j );
7         System.out.println(i)
8     else
9         System.out.println(j);
10
11    while (j < 10 );
12    {
13        j++;
14    }
15
16    do {
17        j++;
18    } while (j < 10)
19 }
20 }
```

4.22 What is wrong with the following programs?

```

1 public class ShowErrors {
2     public static void main(String[] args) {
3         int i;
4         int j = 5;
5
6         if (j > 3)
7             System.out.println(i + 4);
8     }
9 }
```

(a)

```

1 public class ShowErrors {
2     public static void main(String[] args) {
3         for (int i = 0; i < 10; i++)
4             System.out.println(i + 4);
5     }
6 }
```

(b)

4.23 Show the output of the following programs. (*Tip:* Draw a table and list the variables in the columns to trace these programs.)

```

public class Test {
    /** Main method */
    public static void main(String[] args) {
        for (int i = 1; i < 5; i++) {
            int j = 0;
            while (j < i) {
                System.out.print(j + " ");
                j++;
            }
        }
    }
}

```

(a)

```

public class Test {
    /** Main method */
    public static void main(String[] args) {
        int i = 0;
        while (i < 5) {
            for (int j = i; j > 1; j--)
                System.out.print(j + " ");
            System.out.println("****");
            i++;
        }
    }
}

```

(b)

```

public class Test {
    public static void main(String[] args) {
        int i = 5;
        while (i >= 1) {
            int num = 1;
            for (int j = 1; j <= i; j++) {
                System.out.print(num + "xxx");
                num *= 2;
            }
            System.out.println();
            i--;
        }
    }
}

```

(c)

```

public class Test {
    public static void main(String[] args) {
        int i = 1;
        do {
            int num = 1;
            for (int j = 1; j <= i; j++) {
                System.out.print(num + "G");
                num += 2;
            }
            System.out.println();
            i++;
        } while (i <= 5);
    }
}

```

(d)

4.24 What is the output of the following program? Explain the reason.

```

int x = 80000000;
while (x > 0)
    x++;
System.out.println("x is " + x);

```

4.25 Count the number of iterations in the following loops.

```

int count = 0;
while (count < n) {
    count++;
}

```

(a)

```

for (int count = 0;
     count <= n; count++) {
}

```

(b)

```

int count = 5;
while (count < n) {
    count++;
}

```

(c)

```

int count = 5;
while (count < n) {
    count = count + 3;
}

```

(d)

PROGRAMMING EXERCISES



Pedagogical Note

For each problem, read it several times until you understand the problem. Think how to solve the problem before coding. Translate your logic into a program.

A problem often can be solved in many different ways. Students are encouraged to explore various solutions.

read and think before coding

explore solutions

Sections 4.2–4.7

4.1* (*Counting positive and negative numbers and computing the average of numbers*) Write a program that reads an unspecified number of integers, determines how many positive and negative values have been read, and computes the total and average of the input values (not counting zeros). Your program ends with the input **0**. Display the average as a floating-point number. Here is a sample run:



Enter an int value, the program exits if the input is 0:

1 2 -1 3 0

The number of positives is 3

The number of negatives is 1

The total is 5 The average is 1.25

4.2 (*Repeating additions*) [Listing 4.3](#), SubtractionQuizLoop.java, generates five random subtraction questions. Revise the program to generate ten random addition questions for two integers between **1** and **15**. Display the correct count and test time.

4.3 (*Conversion from kilograms to pounds*) Write a program that displays the following table (note that **1** kilogram is **2.2** pounds):

Kilograms

Pounds

1

2.2

3

6.6

...

197

433.4

199

437.8

4.4 (*Conversion from miles to kilometers*) Write a program that displays the following table (note that 1 mile is 1.609 kilometers):

Miles

Kilometers

1

1.609

2

3.218

...

9

14.481

10

16.090

4.5 (*Conversion from kilograms to pounds*) Write a program that displays the following two tables side by side (note that 1 kilogram is 2.2 pounds):

Kilograms

Pounds

Pounds

Kilograms

1

2.2

20

9.09

3

6.6

25

11.36

...

197

433.4

510

231.82

199

437.8

515

234.09

4.6 (*Conversion from miles to kilometers*) Write a program that displays the following two tables side by side (note that 1 mile is 1.609 kilometers):

Miles

Kilometers

Kilometers

Miles

1

1.609

20

12.430

2

3.218

25

15.538

...

9

14.481

60

37.290

10

16.090

65

40.398

4.7** (*Financial application: computing future tuition*) Suppose that the tuition for a university is **\$10,000** this year and increases **5%** every year. Write a program that computes the tuition in ten years and the total cost of four years' worth of tuition starting ten years from now.

4.8 (*Finding the highest score*) Write a program that prompts the user to enter the number of students and each student's name and score, and finally displays the name of the student with the highest score.

4.9* (*Finding the two highest scores*) Write a program that prompts the user to enter the number of students and each student's name and score, and finally displays the student with the highest score and the student with the second-highest score.

4.10 (*Finding numbers divisible by 5 and 6*) Write a program that displays all the numbers from **100** to **1000**, ten per line, that are divisible by **5** and **6**.

4.11 (*Finding numbers divisible by 5 or 6, but not both*) Write a program that displays all the numbers from **100** to **200**, ten per line, that are divisible by **5** or **6**, but not both.

4.12 (*Finding the smallest n such that $n^2 > 12,000$*) Use a **while** loop to find the smallest integer **n** such that **n^2** is greater than 12,000.

4.13 (*Finding the largest n such that $n^3 < 12,000$*) Use a **while** loop to find the largest integer **n** such that **n^3** is less than 12,000.

4.14* (*Displaying the ASCII character table*) Write a program that prints the characters in the ASCII character table from **&!&** to **"~&**. Print ten characters per line. The ASCII table is shown in [Appendix B](#).

Section 4.8

4.15* (*Computing the greatest common divisor*) Another solution for [Listing 4.8](#) to find the greatest common divisor of two integers **n1** and **n2** is as follows: First find **d** to be the minimum of **n1** and **n2**, then check whether **d**, **d-1**, **d-2**, **2**, or **1** is a divisor for both **n1** and **n2** in this order. The first such common divisor is the greatest

common divisor for `n1` and `n2`. Write a program that prompts the user to enter two positive integers and displays the gcd.

4.16** (*Finding the factors of an integer*) Write a program that reads an integer and displays all its smallest factors in increasing order. For example, if the input integer is `120`, the output should be as follows:`2, 2, 2, 3, 5`.

4.17** (*Displaying pyramid*) Write a program that prompts the user to enter an integer from `1` to `15` and displays a pyramid, as shown in the following sample run:



← Enter

Enter the number of lines: 7

					1							
				2	1	2						
			3	2	1	2	3					
		4	3	2	1	2	3	4				
	5	4	3	2	1	2	3	4	5			
6	5	4	3	2	1	2	3	4	5	6		
7	6	5	4	3	2	1	2	3	4	5	6	7

4.18* (*Printing four patterns using loops*) Use nested loops that print the following patterns in four separate programs:

Pattern I

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6

Pattern II

1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

Pattern III

1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1

Pattern IV

1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1

4.19** (*Printing numbers in a pyramid pattern*) Write a nested `for` loop that prints the following output:

			1															
				1	2	1												
				1	2	4	2	1										
				1	2	4	8	4	2	1								
				1	2	4	8	16	8	4	2	1						
				1	2	4	8	16	32	16	8	4	2	1				
				1	2	4	8	16	32	64	32	16	8	4	2	1		
				1	2	4	8	16	32	64	128	64	32	16	8	4	2	1
				1	2	4	8	16	32	64	128	64	32	16	8	4	2	1

4.20* (*Printing prime numbers between 2 and 1000*) Modify [Listing 4.14](#) to print all the prime numbers between 2 and 1000, inclusive. Display eight prime numbers per line.

Comprehensive

4.21** (*Financial application: comparing loans with various interest rates*) Write a program that lets the user enter the loan amount and loan period in number of years and displays the monthly and total payments for each interest rate starting from 5% to 8%, with an increment of 1/8. Here is a sample run:



Loan Amount: 

Number of Years: 

Interest Rate

Monthly Payment

Total Payment

5%

188.71

11322.74

5.125%

189.28

11357.13

5.25%

189.85

11391.59

...

7.875%

202.17

12129.97

8.0%

202.76

12165.83

For the formula to compute monthly payment, see [Listing 2.8](#), ComputeLoan.java.

4.22** (*Financial application: loan amortization schedule*) The monthly payment for a given loan pays the principal and the interest. The monthly interest is computed by multiplying the monthly interest rate and the balance (the remaining principal). The principal paid for the month is therefore the monthly payment minus the monthly interest. Write a program that lets the user enter the loan amount, number of years, and interest rate and displays the amortization schedule for the loan. Here is a sample run:



Video Note

Display loan schedule



Loan Amount: 10000

Number of Years: 1

Annual Interest Rate: 7%

Monthly Payment: 865.26

Total Payment: 10383.21

Payment

Interest

Principal

Balance

1

58.33

806.93

9193.07

2

53.62

811.64

8381.43

...

11

10.0

855.26

860.27

12

5.01

860.25

0.01



Note

The balance after the last payment may not be zero. If so, the last payment should be the normal monthly payment plus the final balance.

Hint: Write a loop to print the table. Since the monthly payment is the same for each month, it should be computed before the loop. The balance is initially the loan amount. For each iteration in the loop, compute the interest and principal, and update the balance. The loop may look like this:

```
for (i = 1; i <= numberOfYears * 12; i++) {  
    interest = monthlyInterestRate * balance;  
    principal = monthlyPayment - interest;  
    balance = balance - principal;  
    System.out.println(i + "\t\t" + interest  
        + "\t\t" + principal + "\t\t" + balance);  
}
```

4.23* (*Obtaining more accurate results*) In computing the following series, you will obtain more accurate results by computing from right to left rather than from left to right:

$$1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

Write a program that compares the results of the summation of the preceding series, computing from left to right and from right to left with $n = 50000$.

4.24* (*Summing a series*) Write a program to sum the following series:

$$\frac{1}{3} + \frac{3}{5} + \frac{5}{7} + \frac{7}{9} + \frac{9}{11} + \frac{11}{13} + \dots + \frac{95}{97} + \frac{97}{99}$$

4.25** (*Computing*) You can approximate by using the following series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots + \frac{1}{i!}$$

Write a program that displays the value for `i = 10000, 20000`, and `100000`.

4.26** (*Computing e*) You can approximate `e` using the following series:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots + \frac{1}{i!}$$

Write a program that displays the `e` value for `i = 10000, 20000`, ..., and `100000`.

$$\frac{1}{i!}$$

(Hint: Since $i! = i \times (i - 1) \times \dots \times 2 \times 1$, then, $\frac{1}{i!}$ is $\frac{1}{i(i - 1)!}$

Initialize `e` and `item` to be `1` and keep adding a new `item` to `e`. The new item is the previous item divided by `i` for `i = 2, 3, 4`,

4.27** (*Displaying leap years*) Write a program that displays all the leap years, ten per line, in the twenty-first century (from 2001 to 2100).

4.28** (*Displaying the first days of each month*) Write a program that prompts the user to enter the year and first day of the year, and displays the first day of each month in the year on the console. For example, if the user entered the year `2005`, and `6` for Saturday, January 1, 2005, your program should display the following output (note that Sunday is `0`):

January 1, 2005 is Saturday

...

December 1, 2005 is Thursday

4.29** (*Displaying calendars*) Write a program that prompts the user to enter the year and first day of the year and displays the calendar table for the year on the console. For example, if the user entered the year `2005`, and `6` for Saturday, January 1, 2005, your program should display the calendar for each month in the year, as follows:

January 2005

Sun

Mon

Tue

Wed

Thu

Fri

Sat

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

...

December 2005

Sun

Mon

Tue

Wed

Thu

Fri

Sat

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

4.30* (*Financial application: compound value*) Suppose you save \$100 each month into a savings account with the annual interest rate 5%. So, the monthly interest rate is $0.05 / 12 = 0.00417$. After the first month, the value in the account becomes

$$100 * (1 + 0.00417) = 100.417$$

After the second month, the value in the account becomes

$$(100 + 100.417) * (1 + 0.00417) = 201.252$$

After the third month, the value in the account becomes

$$(100 + 201.252) * (1 + 0.00417) = 302.507$$

and so on.

Write a program that prompts the user to enter an amount (e.g., 100), the annual interest rate (e.g., 5), and the number of months (e.g., 6) and displays the amount in the savings account after the given month.

4.31* (*Financial application: computing CD value*) Suppose you put \$10,000 into a CD with an annual percentage yield of 5.75%. After one month, the CD is worth

$$10000 + 10000 * 5.75 / 1200 = 10047.91$$

After two months, the CD is worth

$$10047.91 + 10047.91 * 5.75 / 1200 = 10096.06$$

After three months, the CD is worth

$$10096.06 + 10096.06 * 5.75 / 1200 = 10144.43$$

and so on.

Write a program that prompts the user to enter an amount (e.g., **10000**), the annual percentage yield (e.g., **5.75**), and the number of months (e.g., **18**) and displays a table as shown in the sample run.



Enter the initial deposit amount: **10000**

 Enter

Enter annual percentage yield: **5.75**

 Enter

Enter maturity period (number of months): **18**

 Enter

Month

CD Value

1

10047.91

2

10096.06

...

17

10846.56

18

10898.54

4.32** (*Game: lottery*) Revise [Listing 3.9](#), Lottery.java, to generate a lottery of a two-digit number. The two digits in the number are distinct.

(*Hint:* Generate the first digit. Use a loop to continuously generate the second digit until it is different from the first digit.)

4.33** (*Perfect number*) A positive integer is called a *perfect number* if it is equal to the sum of all of its positive divisors, excluding itself. For example, **6** is the first perfectnumber because **6=3+2+1**. The next is **28=14+7+4+2+1**. There are four perfect numbers less than **10000**. Write a program to find all these four numbers.

4.34*** (*Game: scissor, rock, paper*) Exercise 3.17 gives a program that plays the scissor-rock-paper game. Revise the program to let the user continuously play until either the user or the computer wins more than two times.

4.35* (*Summation*) Write a program that computes the following summation.

$$\frac{1}{1 + \sqrt{2}} + \frac{1}{\sqrt{2} + \sqrt{3}} + \frac{1}{\sqrt{3} + \sqrt{4}} + \dots + \frac{1}{\sqrt{624} + \sqrt{625}}$$

4.36** (*Business application: checking ISBN*) Use loops to simplify Exercise 3.19.

4.37** (*Decimal to binary*) Write a program that prompts the user to enter a decimal integer and displays its corresponding binary value. Don't use Java's `Integer.toBinaryString(int)` in this program.

4.38** (*Decimal to hex*) Write a program that prompts the user to enter a decimal integer and displays its corresponding hexadecimal value. Don't use Java's `Integer.toHexString(int)` in this program.

4.39* (*Financial application: finding the sales amount*) You have just started a sales job in a department store. Your pay consists of a base salary and a commission. The base salary is \$5,000. The scheme shown below is used to determine the commission rate.

Sales Amount

Commission Rate

\$0.01–\$5,000

8 percent

\$5,000.01–\$10,000

10 percent

\$10,000.01 and above

12 percent

Your goal is to earn \$30,000 a year. Write a program that finds out the minimum amount of sales you have to generate in order to make \$30,000.

4.40 (*Simulation: head or tail*) Write a program that simulates flipping a coin one million times and displays the number of heads and tails.

4.41** (*Occurrence of max numbers*) Write a program that reads integers, finds the largest of them, and counts its occurrences. Assume that the input ends with number 0. Suppose that you entered 352555 0; the program finds that the largest is 5 and the occurrence count for 5 is 4.

(Hint: Maintain two variables, `max` and `count`. `max` stores the current max number, and `count` stores its occurrences. Initially, assign the first number to `max` and 1 to `count`. Compare each subsequent number with `max`. If the number is greater than `max`, assign it to `max` and reset `count` to 1. If the number is equal to `max`, increment `count` by 1.)



Enter numbers: 3 5 2 5 5 5 0

← Enter

The largest number is 5

The occurrence count of the largest number is 4

4.42* (*Financial application: finding the sales amount*) Rewrite Exercise 4.39 as follows:

- Use a `for` loop instead of a `do-while` loop.
- Let the user enter `COMMISSION_SOUGHT` instead of fixing it as a constant.

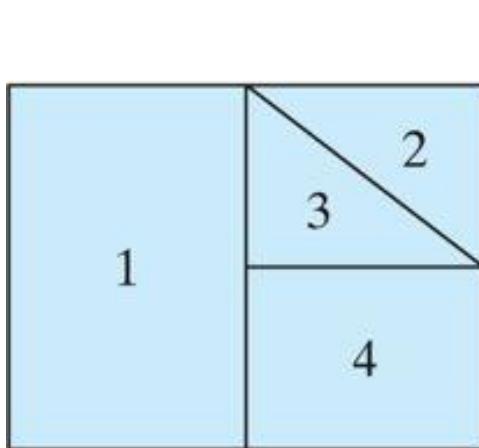
4.43* (*Simulation: clock countdown*) Write a program that prompts the user to enter the number of seconds, displays a message at every second, and terminates when the time expires. Here is a sample run:



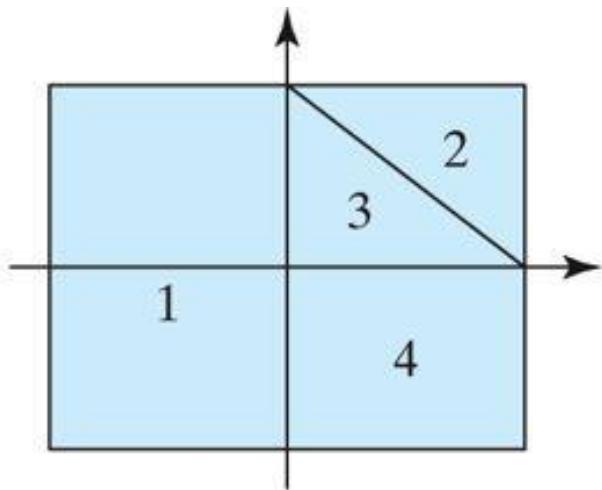
```
Enter the number of second: 3
    ↵ Enter
2 seconds remaining
1 second remaining
Stopped
```

4.44** (*Monte Carlo simulation*) A square is divided into four smaller regions as shown below in (a). If you throw a dart into the square 1000000 times, what is the probability for a dart to fall into an odd-numbered region? Write a program to simulate the process and display the result.

(Hint: Place the center of the square in the center of a coordinate system, as shown in (b). Randomly generate a point in the square and count the number of times a point falls into an odd-numbered region.)



(a)



(b)

4.45* (*Math: combinations*) Write a program that displays all possible combinations for picking two numbers from integers **1** to **7**. Also display the total number of all combinations.



1 2

1 3

...

4.46* (*Computer architecture: bit-level operations*) A **short** value is stored in **16** bits. Write a program that prompts the user to enter a short integer and displays the **16** bits for the integer. Here are sample runs:



Enter an integer: 5

The bits are 000000000000101



Enter an integer: -5

The bits are 111111111111011

(*Hint:* You need to use the bitwise right shift operator (**>>**) and the bitwise AND operator (**&**), which are covered in Supplement III.D on the Companion Website.)

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 115).
<vbk:9781256335153#outline(8)>

CHAPTER 5 METHODS

Objectives

- To define methods ([§5.2](#)).
- To invoke methods with a return value ([§5.3](#)).
- To invoke methods without a return value ([§5.4](#)).
- To pass arguments by value ([§5.5](#)).
- To develop reusable code that is modular, easy to read, easy to debug, and easy to maintain ([§5.6](#)).
- To write a method that converts decimals to hexadecimals ([§5.7](#)).
- To use method overloading and understand ambiguous overloading ([§5.8](#)).
- To determine the scope of variables ([§5.9](#)).
- To solve mathematics problems using the methods in the **Math** class ([§§5.10–5.11](#)).
- To apply the concept of method abstraction in software development ([§5.12](#)).
- To design and implement methods using stepwise refinement ([§5.12](#)).

5.1 Introduction

Suppose that you need to find the sum of integers from **1** to **10**, from **20** to **30**, and from **35** to **45**, respectively. You may write the code as follows:

problem

```
int sum = 0;
for (int i = 1; i <= 10; i++)
    sum += i;
System.out.println("Sum from 1 to 10 is " + sum);
sum = 0;
for (int i = 20; i <= 30; i++)
    sum += i;
System.out.println("Sum from 20 to 30 is " + sum);
sum = 0;
for (int i = 35; i <= 45; i++)
    sum += i;
```

```
System.out.println("Sum from 35 to 45 is " + sum);
```

You may have observed that computing sum from **1** to **10**, from **20** to **30**, and from **35** to **45** are very similar except that the starting and ending integers are different. Wouldn't it be nice if we could write the common code once and reuse it without rewriting it? We can do so by defining a method. The method is for creating reusable code.

why methods?

The preceding code can be simplified as follows:

```
define sum method
1 public static int sum(int i1, int i2) {
2     int sum = 0;
3     for (int i = i1; i <= i2; i++)
4         sum += i;
5
6     return sum;
7 }
8

main method
invoke sum
9 public static void main(String[] args) {
10    System.out.println("Sum from 1 to 10 is " + sum(1, 10));
11    System.out.println("Sum from 20 to 30 is " + sum(20, 30));
12    System.out.println("Sum from 35 to 45 is " + sum(35, 45));
13 }
```

Lines 1–7 define the method named **sum** with two parameters **i** and **j**. The statements in the **main** method invoke **sum(1, 10)** to compute the sum from **1** to **10**, **sum(20, 30)** to compute the sum from **20** to **30**, and **sum(35, 45)** to compute the sum from **35** to **45**.

A method is a collection of statements grouped together to perform an operation. In earlier chapters you have used predefined methods such as **System.out.println**, **JOptionPane.showMessageDialog**, **JOptionPane.showInputDialog**, **Integer.parseInt**, **Double.parseDouble**, **System.exit**, **Math.pow**, and **Math.random**. These methods are defined in the Java library. In this chapter, you will learn how to define your own methods and apply method abstraction to solve complex problems.

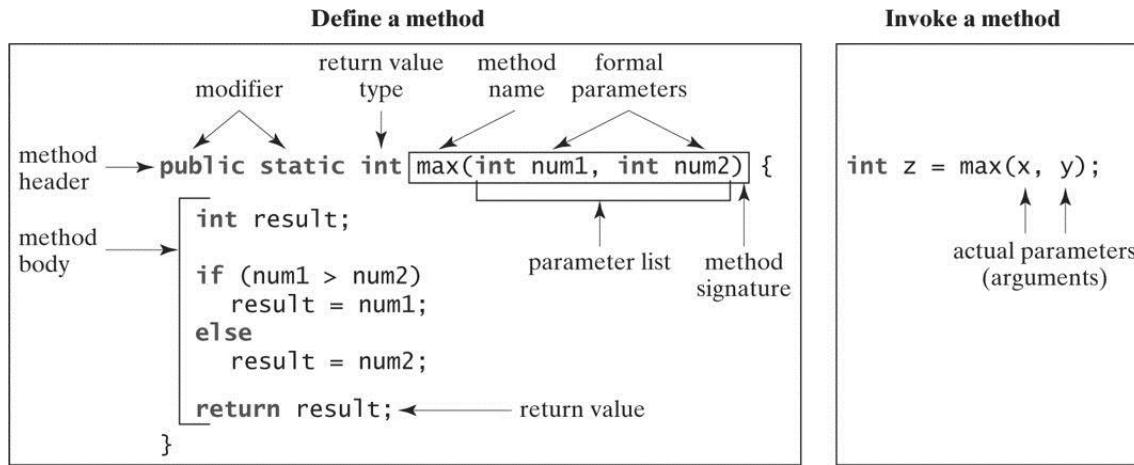
5.2 Defining a Method

The syntax for defining a method is as follows:

```
modifier returnType methodName(list of parameters) {
    // Method body;
}
```

Let's look at a method created to find which of two integers is bigger. This method, named **max**, has two **int** parameters, **num1** and **num2**, the larger of which is returned by the method. [Figure 5.1](#) illustrates the components of this method.

FIGURE 5.1 A method definition consists of a method header and a method body.



The *method header* specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the method. The static modifier is used for all the methods in this chapter. The reason for using it will be discussed in [Chapter 8](#), “Objects and Classes.”

method header

A method may return a value. The `returnValueType` is the data type of the value the method returns. Some methods perform desired operations without returning a value. In this case, the `returnValueType` is the keyword `void`. For example, the `returnValueType` is `void` in the `main` method, as well as in `System.exit`, `System.out.println`, and `JOptionPane.showMessageDialog`. If a method returns a value, it is called a *value-returning method*, otherwise it is a *void method*.

value-returning method

void method

The variables defined in the method header are known as *formal parameters* or simply *parameters*. A parameter is like a placeholder. When a method is invoked, you pass a value to the parameter. This value is referred to as an *actual parameter or argument*. The *parameter list* refers to the type, order, and number of the parameters of a method. The method name and the parameter list together constitute the *method signature*. Parameters are optional; that is, a method may contain no parameters. For example, the `Math.random()` method has no parameters.

parameter

argument

parameter list

method signature

The method body contains a collection of statements that define what the method does. The method body of the `max` method uses an `if` statement to determine which number is larger and return the value of that number. In order for a value-returning method to return a result, a return statement using the keyword `return` is *required*. The method terminates when a return statement is executed.



Note

In certain other languages, methods are referred to as *procedures* and *functions*. A value-returning method is called a *function*; a void method is called a *procedure*.



Caution

In the method header, you need to declare a separate data type for each parameter. For instance, `max(int num1, int num2)` is correct, but `max(int num1, num2)` is wrong.



Note

We say “*define* a method” and “*declare* a variable.” We are making a subtle distinction here. A definition defines what the defined item is, but a declaration usually involves allocating memory to store data for the declared item.

define vs. declare

5.3 Calling a Method

In creating a method, you define what the method is to do. To use a method, you have to *call* or *invoke* it. There are two ways to call a method, depending on whether the method returns a value or not.

If the method returns a value, a call to the method is usually treated as a value. For example,

```
int larger = max(3, 4);
```

calls `max(3, 4)` and assigns the result of the method to the variable `larger`. Another example of a call that is treated as a value is

```
System.out.println(max(3, 4));
```

which prints the return value of the method call `max(3, 4)`.

If the method returns `void`, a call to the method must be a statement. For example, the method `println` returns `void`. The following call is a statement:

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



Note

A value-returning method can also be invoked as a statement in Java. In this case, the caller simply ignores the return value. This is not often done but is permissible if the caller is not interested in the return value.

When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

[Listing 5.1](#) shows a complete program that is used to test the `max` method.



Video Note

Define/invoke max method

LISTING 5.1 TestMax.java

```

1 public class TestMax {
2     /** Main method */
3     public static void main(String[] args) {
4         int i = 5;
5         int j = 2;
6         int k = max(i, j);
7         System.out.println("The maximum between " + i +
8             " and " + j + " is " + k);
9     }
10
11    /** Return the max between two numbers */
12    public static int max(int num1, int num2) {
13        int result;
14
15        if (num1 > num2)
16            result = num1;
17        else
18            result = num2;
19
20        return result;
21    }
22 }
```



The maximum between 5 and 2 is 5

	line#	i	j	k	num1	num2	result
	4	5					
	5		2				
Invoking max	{ 12 13 16				5	2	undefined
	6			5			5



This program contains the **main** method and the **max** method. The **main** method is just like any other method except that it is invoked by the JVM.

max method

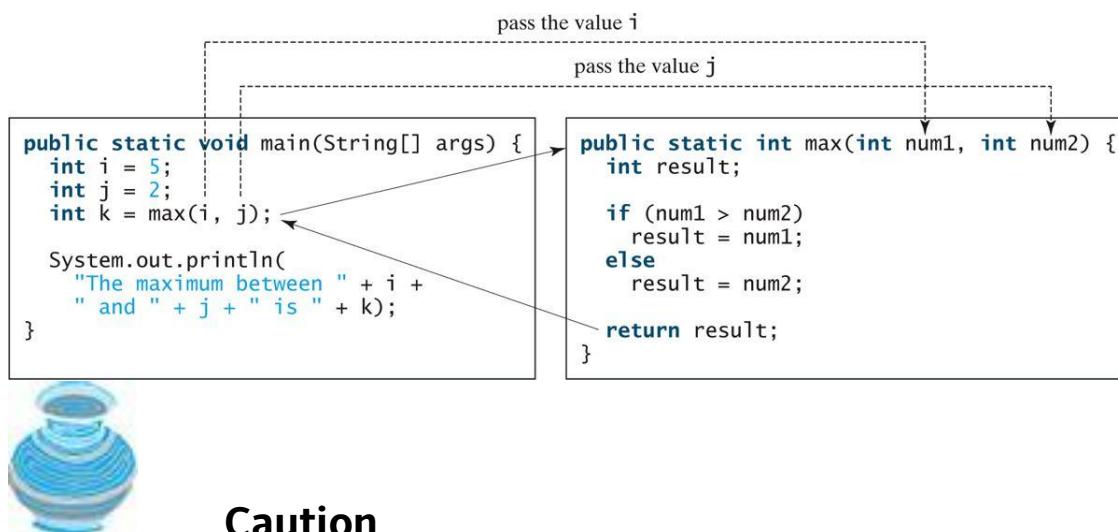
The **main** method's header is always the same. Like the one in this example, it includes the modifiers **public** and **static**, return value type **void**, method name **main**, and a parameter of the **String[]** type. **String[]** indicates that the parameter is an array of **String**, a subject addressed in Chapter 6.

The statements in `main` may invoke other methods that are defined in the class that contains the `main` method or in other classes. In this example, the `main` method invokes `max(i, j)`, which is defined in the same class with the `main` method.

When the `max` method is invoked (line 6), variable `i`'s value `5` is passed to `num1`, and variable `j`'s value `2` is passed to `num2` in the `max` method. The flow of control transfers to the `max` method. The `max` method is executed. When the `return` statement in the `max` method is executed, the `max` method returns the control to its caller (in this case the caller is the `main` method). This process is illustrated in [Figure 5.2](#), main method

max method

FIGURE 5.2 When the max method is invoked, the flow of control transfers to it. Once the max method is finished, it returns control back to the caller.



Caution

A `return` statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compile error because the Java compiler thinks it possible that this method returns no value.

To fix this problem, delete `if (n < 0)` in (a), so that the compiler will see a `return` statement to be reached regardless of how the `if` statement is evaluated.

```
public static int sign(int n) {  
    if (n > 0)  
        return 1;  
    else if (n == 0)  
        return 0;  
    else if (n < 0)  
        return -1;  
}
```

(a)



Should be

```
public static int sign(int n) {  
    if (n > 0)  
        return 1;  
    else if (n == 0)  
        return 0;  
    else  
        return -1;  
}
```

(b)

Note

Methods enable code sharing and reuse. The `max` method can be invoked from any class besides `TestMax`. If you create a new class, you can invoke the `max` method using `Class-Name.methodName` (i.e., `TestMax.max`).

reusing method

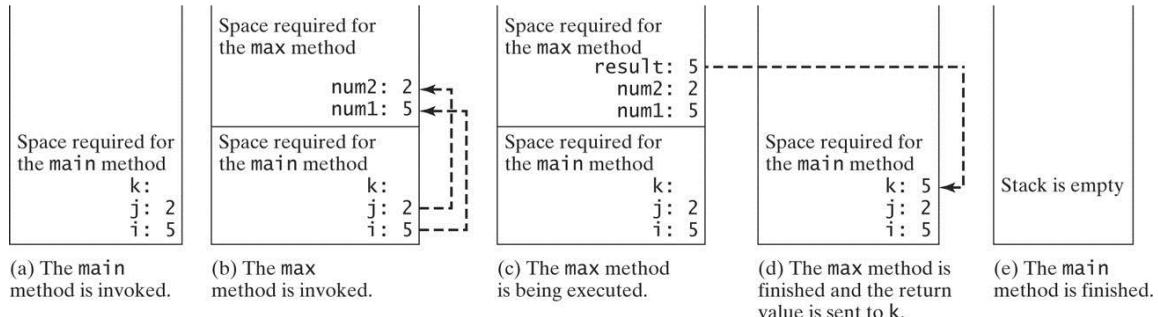
5.3.1 Call Stacks

Each time a method is invoked, the system stores parameters and variables in an area of memory known as a *stack*, which stores elements in last-in, first-out fashion. When a method calls another method, the caller's stack space is kept intact, and new space is created to handle the new method call. When a method finishes its work and returns to its caller, its associated space is released.

stack

Understanding call stacks helps you to comprehend how methods are invoked. The variables defined in the `main` method are `i`, `j`, and `k`. The variables defined in the `max` method are `num1`, `num2`, and `result`. The variables `num1` and `num2` are defined in the method signature and are parameters of the method. Their values are passed through method invocation. [Figure 5.3](#) illustrates the variables in the stack.

FIGURE 5.3 When the `max` method is invoked, the flow of control transfers to the `max` method. Once the `max` method is finished, it returns control back to the caller.



5.4 void Method Example

The preceding section gives an example of a value-returning method. This section shows how to define and invoke a `void` method. [Listing 5.2](#) gives a program that defines a method named `printGrade` and invokes it to print the grade for a given score.



Video Note

Use void method

LISTING 5.2 TestVoidMethod.java

```

1 public class TestVoidMethod {
main method
2     public static void main(String[] args) {
3         System.out.print("The grade is ");
4         printGrade(78.5);                                invoke printGrade
5
6         System.out.print("The grade is ");
7         printGrade(59.5);                                printGrade method
8     }
9
10    public static void printGrade(double score) {
11        if (score >= 90.0) {
12            System.out.println('A');
13        }
14        else if (score >= 80.0) {
15            System.out.println('B');
16        }
17        else if (score >= 70.0) {
18            System.out.println('C');
19        }
20        else if (score >= 60.0) {
21            System.out.println('D');
22        }
23        else {
24            System.out.println('F');
25        }
26    }
27 }
```



The grade is C

The grade is F

The `printGrade` method is a `void` method. It does not return any value. A call to a `void` method must be a statement. So, it is invoked as a statement in line 4 in the `main` method. Like any Java statement, it is terminated with a semicolon.

invoke void method

To see the differences between a void and a value-returning method, let us redesign the `printGrade` method to return a value. The new method, which we call `getGrade`, returns the grade as shown in [Listing 5.3](#).

void vs. valueReturned

LISTING 5.3 TestReturnGradeMethod.java

```
1 public class TestReturnGradeMethod {  
2     public static void main(String[] args) {  
3         System.out.print("The grade is " + getGrade(78.5));  
4         System.out.print("\nThe grade is " + getGrade(59.5));  
5     }  
6  
7     public static char getGrade(double score) {  
8         if (score >= 90.0)  
9             return 'A';  
10        else if (score >= 80.0)  
11            return 'B';  
12        else if (score >= 70.0)  
13            return 'C';  
14        else if (score >= 60.0)  
15            return 'D';  
16        else  
17            return 'F';  
18    }  
19 }
```

main method

invoke `printGrade`

printGrade method



The grade is C

The grade is F

The `getGrade` method defined in lines 7–18 returns a character grade based on the numeric score value. The caller invokes this method in lines 3–4.

The `getGrade` method can be invoked by a caller wherever a character may appear. The `printGrade` method does not return any value. It must be also invoked as a statement.



Note

A `return` statement is not needed for a `void` method, but it can be used for terminating the method and returning to the method's caller. The syntax is simply

```
return
```

This is not often done, but sometimes it is useful for circumventing the normal flow of control in a `void` method. For example, the following code has a return statement to terminate the method when the score is invalid.

```
return in void method
```

```

public static void printGrade(double score) {
    if (score < 0 || score > 100) {
        System.out.println("Invalid score");
        return;
    }

    if (score >= 90.0) {
        System.out.println('A');
    }
    else if (score >= 80.0) {
        System.out.println('B');
    }
    else if (score >= 70.0) {
        System.out.println('C');
    }
    else if (score >= 60.0) {
        System.out.println('D');
    }
    else {
        System.out.println('F');
    }
}

```

5.5 Passing Parameters by Values

The power of a method is its ability to work with parameters. You can use `println` to print any string and `max` to find the maximum between any two `int` values. When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method signature. This is known as *parameter order association*. For example, the following method prints a message `n` times:

parameter order association

```

public static void nPrintln(String message, int n) {
    for (int i = 0; i < n; i++)
        System.out.println(message);
}

```

You can use `nPrintln("Hello", 3)` to print "Hello" three times. The `nPrintln("Hello", 3)` statement passes the actual string parameter "Hello" to the parameter `message`; passes 3 to `n`; and prints "Hello" three times. However, the statement `nPrintln(3, "Hello")` would be wrong. The data type of 3 does not match the data type for the first parameter, `message`, nor does the second parameter, "Hello", match the second parameter, `n`.



Caution

The arguments must match the parameters in *order*, *number*, and *compatible type*, as defined in the method signature. Compatible type means that you can pass an argument to a parameter without explicit casting, such as passing an `int` value argument to a `double` value parameter.

When you invoke a method with a parameter, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method. As shown in [Listing 5.4](#), the value of `x` (1) is passed to the parameter `n` to invoke the `increment` method (line 5). `n` is incremented by 1 in the method (line 10), but `x` is not changed no matter what the method does.

pass-by-value

LISTING 5.4 Increment.java

```
1 public class Increment {  
2     public static void main(String[] args) {  
3         int x = 1;  
4         System.out.println("Before the call, x is " + x);  
5         increment(x);  
6         System.out.println("after the call, x is " + x);  
7     }  
8  
9     public static void increment(int n) {  
10        n++;  
11        System.out.println("n inside the method is " + n);  
12    }  
13 }
```

invoke increment

increment n



Before the call, x is 1
n inside the method is 2
after the call, x is 1

[Listing 5.5](#) gives another program that demonstrates the effect of passing by value. The program creates a method for swapping two variables. The `swap` method is invoked by passing two arguments. Interestingly, the values of the arguments are not changed after the method is invoked.

LISTING 5.5 TestPassByValue.java

```
1 public class TestPassByValue {  
2     /** Main method */  
3     public static void main(String[] args) {  
4         // Declare and initialize variables  
5         int num1 = 1;  
6         int num2 = 2;  
7  
8         System.out.println("Before invoking the swap method, num1 is " +  
9             num1 + " and num2 is " + num2);  
10  
11        // Invoke the swap method to attempt to swap two variables  
12        swap(num1, num2);  
13  
14        System.out.println("After invoking the swap method, num1 is " +  
15            num1 + " and num2 is " + num2);  
16    }  
17  
18    /** Swap two variables */  
19    public static void swap(int n1, int n2) {  
20        System.out.println("\tInside the swap method");  
21        System.out.println("\t\tBefore swapping n1 is " + n1  
22            + " n2 is " + n2);  
23  
24        // Swap n1 with n2  
25        int temp = n1;  
26        n1 = n2;  
27        n2 = temp;  
28  
29        System.out.println("\t\tAfter swapping n1 is " + n1  
30            + " n2 is " + n2);  
31    }  
32 }
```



Before invoking the swap method, num1 is 1 and num2 is 2

Inside the swap method

Before swapping n1 is 1 n2 is 2

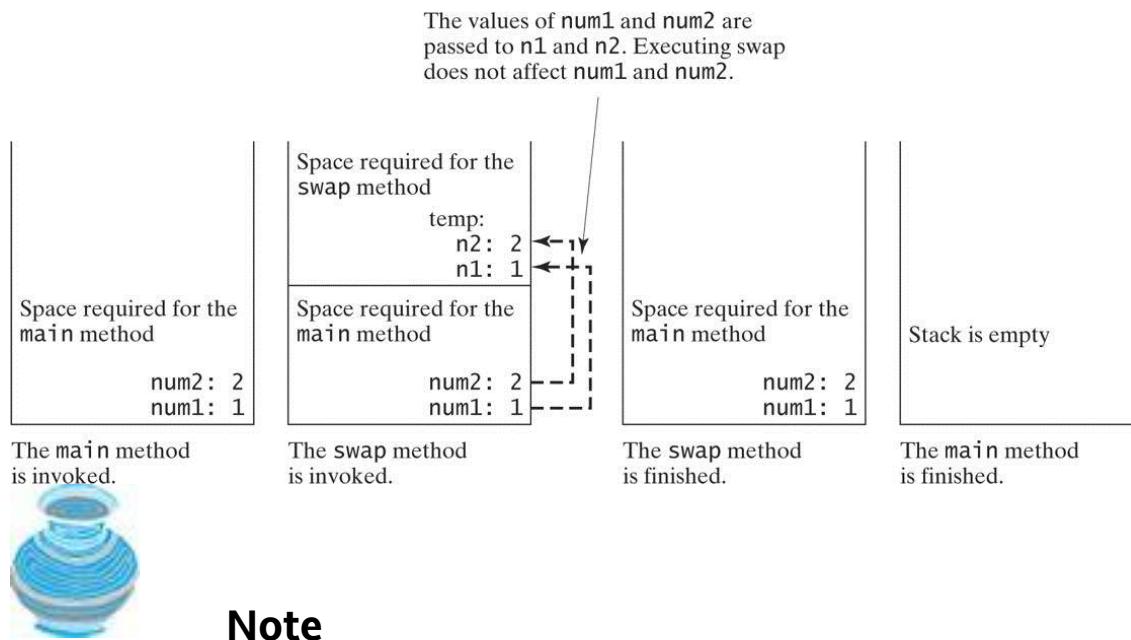
After swapping n1 is 2 n2 is 1

After invoking the swap method, num1 is 1 and num2 is 2

Before the `swap` method is invoked (line 12), `num1` is 1 and `num2` is 2. After the `swap` method is invoked, `num1` is still 1 and `num2` is still 2. Their values have not been swapped. As shown in [Figure 5.4](#), the values of the arguments `num1` and `num2` are passed to `n1` and `n2`, but `n1` and `n2` have their own memory locations independent of `num1` and `num2`. Therefore, changes in `n1` and `n2` do not affect the contents of `num1` and `num2`.

Another twist is to change the parameter name `n1` in `swap` to `num1`. What effect does this have? No change occurs, because it makes no difference whether the parameter and the argument have the same name. The parameter is a variable in the method with its own memory space. The variable is allocated when the method is invoked, and it disappears when the method is returned to its caller.

FIGURE 5.4 The values of the variables are passed to the parameters of the method.



Note

For simplicity, Java programmers often say *passing an argument x to a parameter y* , which actually means *passing the value of x to y* .

5.6 Modularizing Code

Methods can be used to reduce redundant code and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.



Video Note

Modularize code

[Listing 4.8](#) gives a program that prompts the user to enter two integers and displays their greatest common divisor. You can rewrite the program using a method, as shown in [Listing 5.6](#).

LISTING 5.6 GreatestCommonDivisorMethod.java

```
1 import java.util.Scanner;
2
3 public class GreatestCommonDivisorMethod {
4     /** Main method */
5     public static void main(String[] args) {
6         // Create a Scanner
7         Scanner input = new Scanner(System.in);
8
9         // Prompt the user to enter two integers
10        System.out.print("Enter first integer: ");
11        int n1 = input.nextInt();
12        System.out.print("Enter second integer: ");
13        int n2 = input.nextInt();
14
15        System.out.println("The greatest common divisor for " + n1 +
16                           " and " + n2 + " is " + gcd(n1, n2));           invoke gcd
17    }
18
19    /** Return the gcd of two integers */
20    public static int gcd(int n1, int n2) {                      compute gcd
21        int gcd = 1; // Initial gcd is 1
22        int k = 2; // Possible gcd
23
24        while (k <= n1 && k <= n2) {
25            if (n1 % k == 0 && n2 % k == 0)
26                gcd = k; // Update gcd
27            k++;
28        }
29
30        return gcd; // Return gcd                                return gcd
31    }
32 }
```



```
Enter first integer: 45 ↵ Enter
```

```
Enter second integer: 75 ↵ Enter
```

The greatest common divisor for 45 and 75 is 15

By encapsulating the code for obtaining the gcd in a method, this program has several advantages:

1. It isolates the problem for computing the gcd from the rest of the code in the main method. Thus, the logic becomes clear and the program is easier to read.
2. The errors on computing gcd are confined in the `gcd` method, which narrows the scope of debugging.
3. The `gcd` method now can be reused by other programs.

[Listing 5.7](#) applies the concept of code modularization to improve Listing [4.14](#), PrimeNumber.java.

LISTING 5.7 PrimeNumberMethod.java

```

1 public class PrimeNumberMethod {
2     public static void main(String[] args) {
3         System.out.println("The first 50 prime numbers are \n");
4         printPrimeNumbers(50);
5     }
6
7     public static void printPrimeNumbers(int numberOfPrimes) {
8         final int NUMBER_OF_PRIMES_PER_LINE = 10; // Display 10 per line
9         int count = 0; // Count the number of prime numbers
10        int number = 2; // A number to be tested for primeness
11
12        // Repeatedly find prime numbers
13        while (count < numberOfPrimes) {
14            // Print the prime number and increase the count
15            if (isPrime(number)) {
16                count++; // Increase the count
17
18                if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
19                    // Print the number and advance to the new line
20                    System.out.printf("%-5s\n", number);
21                }
22                else
23                    System.out.printf("%-5s", number);
24            }
25
26            // Check whether the next number is prime
27            number++;
28        }
29    }
30
31    /** Check whether number is prime */
32    public static boolean isPrime(int number) {
33        for (int divisor = 2; divisor <= number / 2; divisor++) {
34            if (number % divisor == 0) { // If true, number is not prime
35                return false; // number is not a prime
36            }
37        }
38
39        return true; // number is prime
40    }
41 }

```



The first 50 prime numbers are

2	3	5	7	11	13	17	19	23	29
31	37	41	43	47	53	59	61	67	71
73	79	83	89	97	101	103	107	109	113
127	131	137	139	149	151	157	163	167	173
179	181	191	193	197	199	211	223	227	229

We divided a large problem into two subproblems. As a result, the new program is easier to read and easier to debug. Moreover, the methods `printPrimeNumbers` and `isPrime` can be reused by other programs.

5.7 Problem: Converting Decimals to Hexadecimals

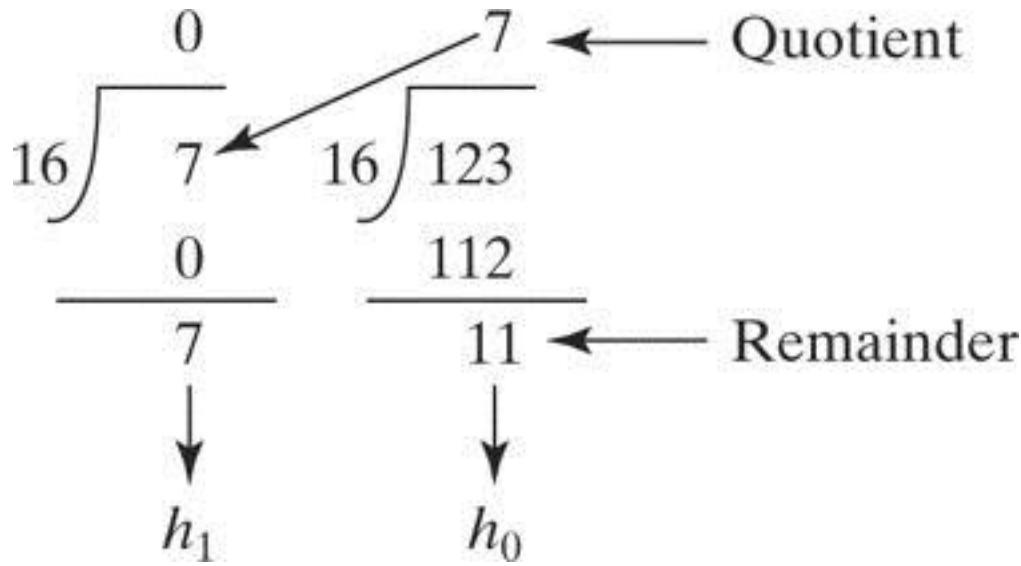
Hexadecimals are often used in computer systems programming. [Appendix F](#) introduces number systems. This section presents a program that converts a decimal to a hexadecimal.

To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits h_n , h_{n-1} , h_{n-2} , ..., h_2 , h_1 , and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots \\ + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These numbers can be found by successively dividing d by 16 until the quotient is 0. The remainders are h_0 , h_1 , h_2 , ..., h_{n-2} , h_{n-1} , and h_n .

For example, the decimal number `123` is `7B` in hexadecimal. The conversion is done as follows:



[Listing 5.8](#) gives a program that prompts the user to enter a decimal number and converts it into a hex number as a string.

LISTING 5.8 Decimal2HexConversion.java

```

1 import java.util.Scanner;
2
3 public class Decimal2HexConversion {
4     /** Main method */
5     public static void main(String[] args) {
6         // Create a Scanner
7         Scanner input = new Scanner(System.in);
8
9         // Prompt the user to enter a decimal integer
10        System.out.print("Enter a decimal number: ");
11        int decimal = input.nextInt();                                input string
12
13        System.out.println("The hex number for decimal " +           hex to decimal
14            decimal + " is " + decimalToHex(decimal));
15    }
16
17    /** Convert a decimal to a hex as a string */
18    public static String decimalToHex(int decimal) {
19        String hex = "";
20
21        while (decimal != 0) {
22            int hexValue = decimal % 16;
23            hex = toHexChar(hexValue) + hex;
24            decimal = decimal / 16;
25
26        }
27        return hex;
28    }
29
30    /** Convert an integer to a single hex digit in a character */
31    public static char toHexChar(int hexValue) {
32        if (hexValue <= 9 && hexValue >= 0)
33            return (char)(hexValue + '0');
34        else // hexValue <= 15 && hexValue >= 10
35            return (char)(hexValue - 10 + 'A');
36    }
37 }

```

hex char to decimal
to uppercase



Enter a decimal number:

The hex number for decimal 1234 is 4D2



line#	decimal	hex	hexValue	toHexChar(hexValue)
19	1234	“”		
iteration 1 22 23 24			2	
		“2”		2
	77			
iteration 2 22 23 24			13	
		“D2”		D
	4			
iteration 3 22 23 24			4	
		“4D2”		4
	0			

The program uses the `decimalToHex` method (lines 18–28) to convert a decimal integer to a hex number as a string. The method gets the remainder of the division of the decimal integer by `16` (line 22). The remainder is converted into a character by invoking the `toHexChar` method (line 23). The character is then appended to the hex string (line 23). The hex string is initially empty (line 19). Divide the decimal number by `16` to remove a hex digit from the number (line 24). The method repeatedly performs these operations in a loop until quotient becomes `0` (lines 21–25).

The `toHexChar` method (lines 31–36) converts a `hexValue` between `0` and `15` into a hex character. If `hexValue` is between `0` and `9`, it is converted to `(char)(hexValue + & 0 &)` (line 33). Recall when adding a character with an integer, the character's Unicode is used in the evaluation. For example, if `hexValue` is `5`, `(char)(hexValue + & 0 &)` returns `& 5 &`. Similarly, if `hexValue` is between `10` and `15`, it is converted to `(char)(hexValue + & A &)` (line 35). For example, if `hexValue` is `11`, `(char)(hexValue + & A &)` returns `& B &`.

5.8 Overloading Methods

The `max` method that was used earlier works only with the `int` data type. But what if you need to determine which of two floating-point numbers has the maximum value? The solution is to create another method with the same name but different parameters, as shown in the following code:

```
public static double max(double num1, double num2) {  
    if (num1 > num2)  
        return num1;  
    else  
        return num2;  
}
```

If you call `max` with `int` parameters, the `max` method that expects `int` parameters will be invoked; if you call `max` with `double` parameters, the `max` method that expects `double` parameters will be invoked. This is referred to as *method overloading*; that is, two methods have the same name but different parameter lists within one class. The Java compiler determines which method is used based on the method signature.

method overloading

[Listing 5.9](#) is a program that creates three methods. The first finds the maximum integer, the second finds the maximum double, and the third finds the maximum among three double values. All three methods are named `max`.

LISTING 5.9 TestMethodOverloading.java

```
1 public class TestMethodOverloading {
2     /* Main method */
3     public static void main(String[] args) {
4         // Invoke the max method with int parameters
5         System.out.println("The maximum between 3 and 4 is "
6             + max(3, 4));
7
8         // Invoke the max method with the double parameters
9         System.out.println("The maximum between 3.0 and 5.4 is "
10            + max(3.0, 5.4));
11
12        // Invoke the max method with three double parameters
13        System.out.println("The maximum between 3.0, 5.4, and 10.14 is "
14            + max(3.0, 5.4, 10.14));
15    }
16
17    /* Return the max between two int values */
18    public static int max(int num1, int num2) {                                overloaded max
19        if (num1 > num2)
20            return num1;
21        else
22            return num2;
23    }
24
25    /* Find the max between two double values */
26    public static double max(double num1, double num2) {                         overloaded max
27        if (num1 > num2)
28            return num1;
29        else
30            return num2;
31    }
32
33    /* Return the max among three double values */
34    public static double max(double num1, double num2, double num3) {           overloaded max
35        return max(max(num1, num2), num3);
36    }
37 }
```



The maximum between 3 and 4 is 4

The maximum between 3.0 and 5.4 is 5.4

The maximum between 3.0, 5.4, and 10.14 is 10.14

When calling `max(3, 4)` (line 6), the `max` method for finding the maximum of two integers is invoked. When calling `max(3.0, 5.4)` (line 10), the `max` method for finding the maximum of two doubles is invoked. When calling `max(3.0, 5.4, 10.14)` (line 14), the `max` method for finding the maximum of three double values is invoked.

Can you invoke the `max` method with an `int` value and a `double` value, such as `max(2, 2.5)`? If so, which of the `max` methods is invoked? The answer to the first question is yes. The answer to the second is that the `max` method for finding the maximum of two `double` values is invoked. The argument value 2 is automatically converted into a `double` value and passed to this method.

You may be wondering why the method `max(double, double)` is not invoked for the call `max(3, 4)`. Both `max(double, double)` and `max(int, int)` are possible matches for `max(3, 4)`. The Java compiler finds the most specific method for a method invocation. Since the method `max(int, int)` is more specific than `max(double, double)`, `max(int, int)` is used to invoke `max(3, 4)`.



Note

Overloaded methods must have different parameter lists. You cannot overload methods based on different modifiers or return types.



Note

Sometimes there are two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*. Ambiguous invocation causes a compile error. Consider the following code:

ambiguous invocation

```

public class AmbiguousOverloading {
    public static void main(String[] args) {
        System.out.println(max(1, 2));
    }

    public static double max(int num1, double num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }

    public static double max(double num1, int num2) {
        if (num1 > num2)
            return num1;
        else
            return num2;
    }
}

```

Both `max(int, double)` and `max(double, int)` are possible candidates to match `max(1, 2)`. Since neither is more specific than the other, the invocation is ambiguous, resulting in a compile error.

5.9 The Scope of Variables

The *scope of a variable* is the part of the program where the variable can be referenced. A variable defined inside a method is referred to as a *local variable*.

local variable

The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared and assigned a value before it can be used.

A parameter is actually a local variable. The scope of a method parameter covers the entire method.

A variable declared in the initial-action part of a `for`-loop header has its scope in the entire loop. But a variable declared inside a `for`-loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable, as shown in [Figure 5.5](#).

FIGURE 5.5 A variable declared in the initial action part of a for-loop header has its scope in the entire loop.

```
public static void method1() {  
    .  
    .  
    .  
    for (int i = 1; i < 10; i++) {  
        .  
        .  
        int j;  
        .  
    }  
    .  
}
```

The scope of i →

The scope of j →

You can declare a local variable with the same name in different blocks in a method, but you cannot declare a local variable twice in the same block or in nested blocks, as shown in [Figure 5.6](#).

FIGURE 5.6 A variable can be declared multiple times in nonnested blocks but only once in nested blocks.

<p>It is fine to declare i in two nonnested blocks</p> <pre>public static void method1() { int x = 1; int y = 1; for (int i = 1; i < 10; i++) { x += i; } for (int i = 1; i < 10; i++) { y += i; } }</pre>	<p>It is wrong to declare i in two nested blocks</p> <pre>public static void method2() { int i = 1; int sum = 0; for (int i = 1; i < 10; i++) sum += i; }</pre>
---	--



Caution

Do not declare a variable inside a block and then attempt to use it outside the block. Here is an example of a common mistake:

```
for (int i = 0; i < 10; i++) {  
}  
System.out.println(i);
```

The last statement would cause a syntax error, because variable `i` is not defined outside of the `for` loop.

5.10 The Math Class

The `Math` class contains the methods needed to perform basic mathematical functions. You have already used the `pow(a, b)` method to compute in [Listing 2.8](#), `ComputeLoan.java`, and the `Math.random()` method in [Listing 3.4](#), `SubtractionQuiz.java`. This section introduces other useful methods in the `Math` class. They can be categorized as *trigonometric methods*, *exponent methods*, and *service methods*. Besides methods, the `Math` class provides two useful `double` constants, `PI` and `E` (the base of natural logarithms). You can use these constants as `Math.PI` and `Math.E` in any program.

5.10.1 Trigonometric Methods

The `Math` class contains the following trigonometric methods:

```
/** Return the trigonometric sine of an angle in radians */  
  
public static double sin(double radians)  
/** Return the trigonometric cosine of an angle in radian s */  
public static double cos(double radians)  
/** Return the trigonometric tangent of an angle in radians */  
public static double tan(double radians)  
/** Convert the angle in degrees to an angle in radians */  
  
public static double toRadians(double degree)  
/** Convert the angle in radians to an angle in degrees */  
  
public static double toDegrees(double radians)  
/** Return the angle in radians for the inverse of sin */  
public static double asin(double a)  
/** Return the angle in radians for the inverse of cos */  
public static double acos(double a)  
/** Return the angle in radians for the inverse of tan */  
public static double atan(double a)
```

The parameter for `sin`, `cos`, and `tan` is an angle in radians. The return value for `asin`, `acos`, and `atan` is a degree in radians in the range between and One degree is equal to in radians, 90 degrees is equal to in radians, and 30 degrees is equal to in radians.

For example,

```
Math.toDegrees(Math.PI / 2) returns 90.0
Math.toRadians(30) returns returns π/6
Math.sin(0) returns 0.0
Math.sin(Math.toRadians(270)) returns -1.0
Math.sin(Math.PI / 6) returns 0.5
Math.sin(Math.PI / 2) returns 1.0
Math.cos(0) returns 1.0
Math.cos(Math.PI / 6) returns 0.866
Math.cos(Math.PI / 2) returns 0
Math.asin(0.5) returns π/6
```

5.10.2 Exponent Methods

There are five methods related to exponents in the `Math` class:

```
/** Return e raised to the power of x (ex) */
public static double exp(double x)
/** Return the natural logarithm of x (ln(x) = loge(x)) */
public static double log(double x)
/** Return the base 10 logarithm of x (log10 (x)) */
public static double log10(double x)
/** Return a raised to the power of b (ab) */
public static double pow(double a, double b)
/** Return the square root of x (✓x) for x >= 0 */
public static double sqrt(double x)
```

For example,

```
Math.exp(1) returns 2.71828
Math.log(Math.E) returns 1.0
Math.log10(10) returns 1.0
Math.pow(2, 3) returns 8.0
Math.pow(3, 2) returns 9.0
Math.pow(3.5, 2.5) returns 22.91765
Math.sqrt(4) returns 2.0
Math.sqrt(10.5) returns 3.24
```

5.10.3 The Rounding Methods

The `Math` class contains five rounding methods:

```

/** x is rounded up to its nearest integer. This integer
 * is
 *      * returned as a double value. */
public static double ceil(double x)
/** x is rounded down to its nearest integer. This integer
 * is
 *      * returned as a double value. */
public static double floor (double x)
/** x is rounded to its nearest integer. If x is equally
 * close
 *      * to two integers, the even one is returned as a double
 * . */
public static double rint(double x)
/** Return (int)Math.floor(x + 0.5). */
public static int round(float x)
/** Return (long)Math.floor(x + 0.5). */
public static long round(double x)

```

For example,

```

Math.ceil(2.1) returns 3.0
Math.ceil(2.0) returns 2.0
Math.ceil(-2.0) returns -2.0
Math.ceil(-2.1) returns -2.0
Math.floor(2.1) returns 2.0
Math.floor(2.0) returns 2.0
Math.floor(-2.0) returns -2.0
Math.floor(-2.1) returns -3.0
Math.rint(2.1) returns 2.0
Math.rint(-2.0) returns -2.0
Math.rint(-2.1) returns -2.0
Math.rint(2.5) returns 2.0
Math.rint(3.5) returns 4.0
Math.rint(-2.5) returns -2.0
Math.round(2.6f) returns 3 // Returns int
Math.round(2.0) returns 2 // Returns long
Math.round(-2.0f) returns -2
Math.round(-2.6) returns -3

```

5.10.4 The min, max, and abs Methods

The **min** and **max** methods are overloaded to return the minimum and maximum numbers between two numbers (**int**, **long**, **float**, or **double**). For example, **max(3.4, 5.0)** returns **5.0**, and **min(3, 2)** returns **2**.

The **abs** method is overloaded to return the absolute value of the number (**int**, **long**, **float**, and **double**). For example,

```

Math.max(2, 3) returns 3
Math.max(2.5, 3) returns 3.0
Math.min(2.5, 3.6) returns 2.5
Math.abs(-2) returns 2
Math.abs(-2.1) returns 2.1

```

5.10.5 The random Method

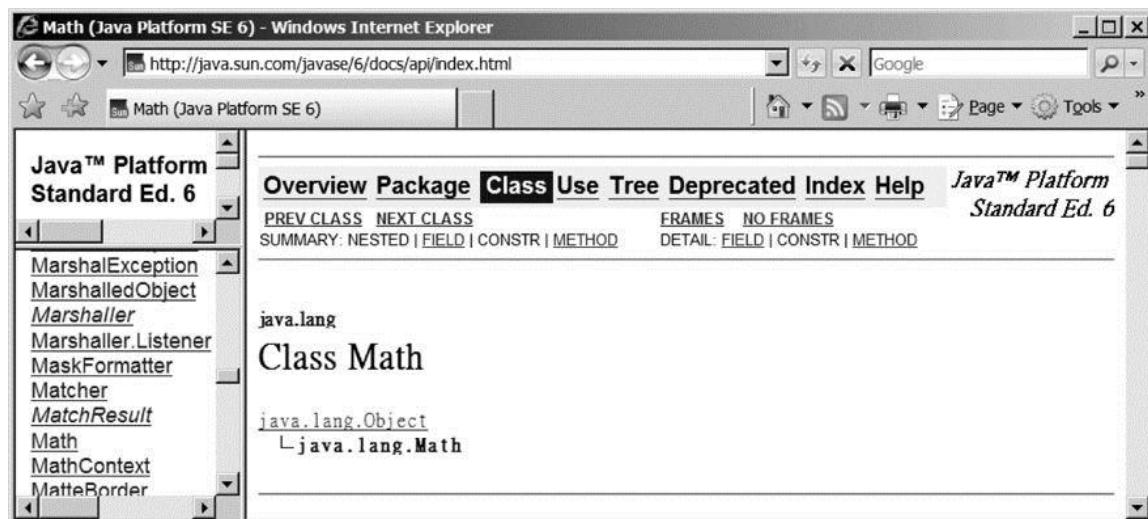
You have used the `random()` method to generate a random `double` value greater than or equal to 0.0 and less than 1.0 (`0 <= Math.random() < 1.0`). This method is very useful. You can use it to write a simple expression to generate random numbers in any range. For example,

`(int) (Math.random() * 10)` → Returns a random integer between 0 and 9
`50 + (int) (Math.random() * 50)` → Returns a random integer between 50 and 99

In general,

`a + Math.random() * b` → Returns a random number between `a` and `a + b` excluding `a + b`

FIGURE 5.7 You can view the documentation for Java API online.





Tip

You can view the complete documentation for the `Math` class online at <http://java.sun.com/javase/6/docs/api/index.html>, as shown in [Figure 5.7](#).



Note

Not all classes need a `main` method. The `Math` class and `JOptionPane` class do not have `main` methods. These classes contain methods for other classes to use.

5.11 Case Study: Generating Random Characters

Computer programs process numerical data and characters. You have seen many examples that involve numerical data. It is also important to understand characters and how to process them. This section presents an example for generating random characters.

As introduced in §2.13, every character has a unique Unicode between 0 and FFFF in hexadecimal (65535 in decimal). To generate a random character is to generate a random integer between 0 and 65535 using the following expression (note that since `0 <= Math.random() < 1.0`, you have to add 1 to 65535):

```
(int) (Math.random() * (65535 + 1))
```

Now let us consider how to generate a random lowercase letter. The Unicodes for lowercase letters are consecutive integers starting from the Unicode for `&a&`, then that for `&b&`, `&c&`, and `&z&`. The Unicode for `&a&` is

```
(int) &a&
```

So a random integer between `(int) &a&` and `(int) &z&` is

```
(int) ((int) &a& + Math.random() * ((int) &z& - (int) &a& + 1))
```

As discussed in §2.13.3, all numeric operators can be applied to the `char` operands. The `char` operand is cast into a number if the other operand is a number or a character. Thus the preceding expression can be simplified as follows:

```
&a& + Math.random() * (&z& - &a& + 1)
```

and a random lowercase letter is

```
(char) (&a& + Math.random() * (&z& - &a& + 1))
```

To generalize the foregoing discussion, a random character between any two characters `ch1` and `ch2` with `ch1 < ch2` can be generated as follows:

```
(char) (ch1 + Math.random() * (ch2 - ch1 + 1))
```

This is a simple but useful discovery. Let us create a class named `RandomCharacter` in Listing 5.10 with five overloaded methods to get a certain type of character randomly. You can use these methods in your future projects.

LISTING 5.10 RandomCharacter.java

```
1 public class RandomCharacter {  
2     /** Generate a random character between ch1 and ch2 */  
3     public static char getRandomCharacter(char ch1, char ch2) {           getRandomCharacter  
4         return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));  
5     }  
  
6  
7     /** Generate a random lowercase letter */  
8     public static char getRandomLowerCaseLetter() {  
9         return getRandomCharacter('a', 'z');  
10    }  
11  
12    /** Generate a random uppercase letter */  
13    public static char getRandomUpperCaseLetter() {  
14        return getRandomCharacter('A', 'Z');  
15    }  
16  
17    /** Generate a random digit character */  
18    public static char getRandomDigitCharacter() {  
19        return getRandomCharacter('0', '9');  
20    }  
21  
22    /** Generate a random character */  
23    public static char getRandomCharacter() {  
24        return getRandomCharacter('\u0000', '\uFFFF');  
25    }  
26 }
```

[Listing 5.11](#) gives a test program that displays 175 random lowercase letters.

LISTING 5.11 TestRandomCharacter.java

```

1 public class TestRandomCharacter {
2     /** Main method */
3     public static void main(String[] args) {
4         final int NUMBER_OF_CHARS = 175;
5         final int CHARS_PER_LINE = 25;
6
7         // Print random characters between 'a' and 'z', 25 chars per line
8         for (int i = 0; i < NUMBER_OF_CHARS; i++) {
9             char ch = RandomCharacter.getRandomLowerCaseLetter();
10            if ((i + 1) % CHARS_PER_LINE == 0)
11                System.out.println(ch);
12            else
13                System.out.print(ch);
14        }
15    }
16 }
```



gmjsuhezfkgtazqgmswfclrao

pnrunulnwmaztlfjedmpchcif

lalqdgivvxkxpzbzulrmqbhikr

lbnrjlsopfxahssqhwuuljvbe

xbhdotzhpehbqmuwsfktwsoli

cbuwkzgxpmtdzihgatdsvbwbz

bfesoklwbhnooygiigzdxuqni

Line 9 invokes `getRandomLowerCaseLetter()` defined in the `RandomCharacter` class. Note that `getRandomLowerCaseLetter()` does not have any parameters, but you still have to use the parentheses when defining and invoking the method.

parentheses required

5.12 Method Abstraction and Stepwise Refinement

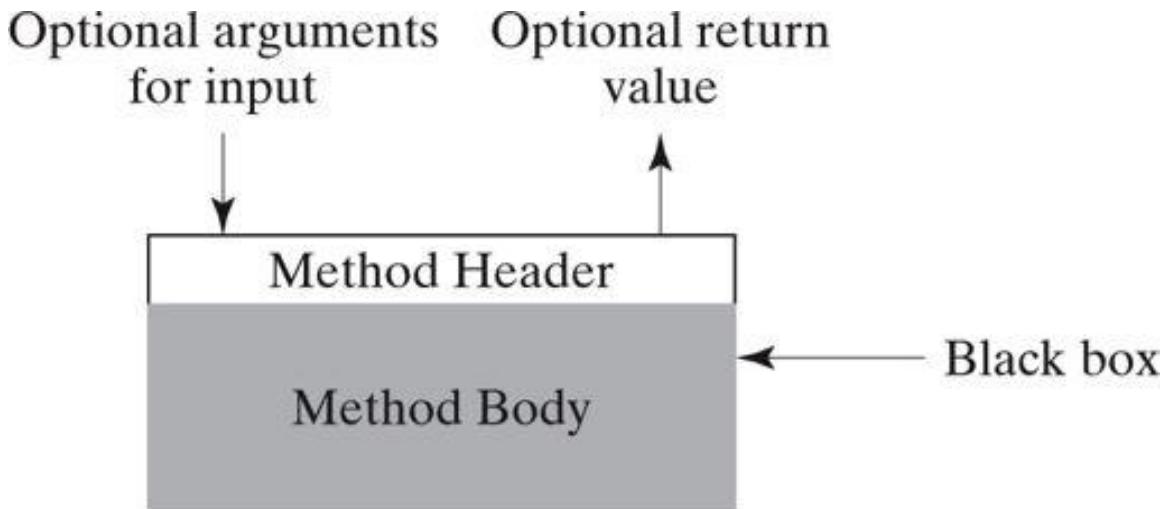
method abstraction

The key to developing software is to apply the concept of abstraction. You will learn many levels of abstraction from this book. *Method abstraction* is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is known as *information hiding* or

encapsulation. If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature. The implementation of the method is hidden from the client in a “black box,” as shown in [Figure 5.8](#).

information hiding

FIGURE 5.8 The method body can be thought of as a black box that contains the detailed implementation for the method.



You have already used the `System.out.print` method to display a string, the `JOptionPane.showInputDialog` method to read a string from a dialog box, and the `max` method to find the maximum number. You know how to write the code to invoke these methods in your program, but as a user of these methods, you are not required to know how they are implemented.

The concept of method abstraction can be applied to the process of developing programs. When writing a large program, you can use the *divide-and-conquer* strategy, also known as *stepwise refinement*, to decompose it into subproblems. The subproblems can be further decomposed into smaller, more manageable problems.

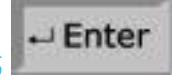
divide and conquer

stepwise refinement

Suppose you write a program that displays the calendar for a given month of the year. The program prompts the user to enter the year and the month, then displays the entire calendar

for the month, as shown in the following sample run: Let us use this example to demonstrate the divide-and-conquer approach.



Enter full year (e.g., 2001): **2006** 

Enter month in number between 1 and 12: **6** 

June 2006

Sun

Mon

Tue

Wed

Thu

Fri

Sat

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

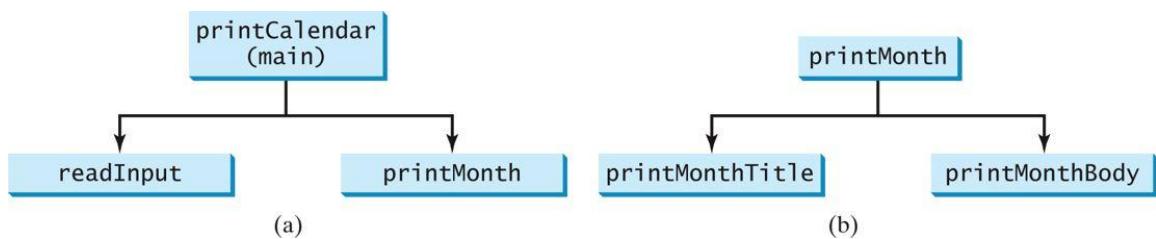
Let us use this example to demonstrate the divide-and-conquer approach.

5.12.1 Top-Down Design

How would you get started on such a program? Would you immediately start coding? Beginning programmers often start by trying to work out the solution to every detail. Although details are important in the final program, concern for detail in the early stages may block the problem-solving process. To make problem solving flow as smoothly as possible, this example begins by using method abstraction to isolate details from design and only later implements the details.

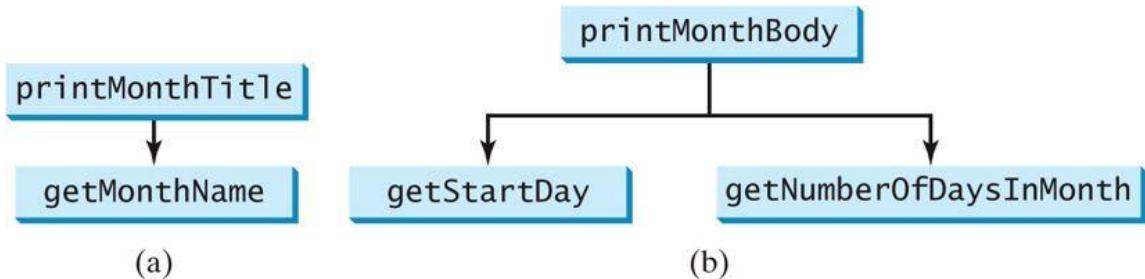
For this example, the problem is first broken into two subproblems: get input from the user, and print the calendar for the month. At this stage, you should be concerned with what the subproblems will achieve, not with how to get input and print the calendar for the month. You can draw a structure chart to help visualize the decomposition of the problem (see [Figure 5.9\(a\)\)](#).

FIGURE 5.9 The structure chart shows that the `printCalendar` problem is divided into two subproblems, `readInput` and `printMonth`, and that `printMonth` is divided into two smaller subproblems, `printMonthTitle` and `printMonthBody`.



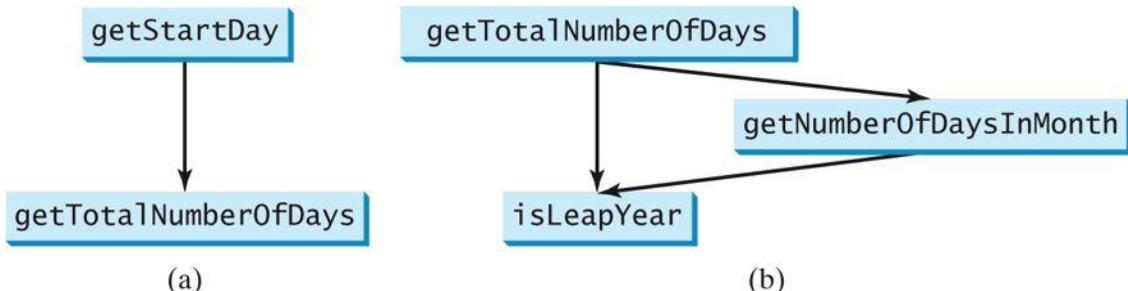
The problem of printing the calendar for a given month can be broken into two subproblems: print the month title, and print the month body, as shown in [Figure 5.9\(b\)](#). The month title consists of three lines: month and year, a dash line, and the names of the seven days of the week. You need to get the month name (e.g., January) from the numeric month (e.g., 1). This is accomplished in `getMonthName` (see [Figure 5.10\(a\)\)](#).

FIGURE 5.10 (a) To `printMonthTitle`, you need `getMonthName`. (b) The `printMonthBody` problem is refined into several smaller problems.



In order to print the month body, you need to know which day of the week is the first day of the month (`getStartDay`) and how many days the month has (`getNumberOfDaysInMonth`), as shown in [Figure 5.10\(b\)](#). For example, December 2005 has 31 days, and December 1, 2005, is Thursday.

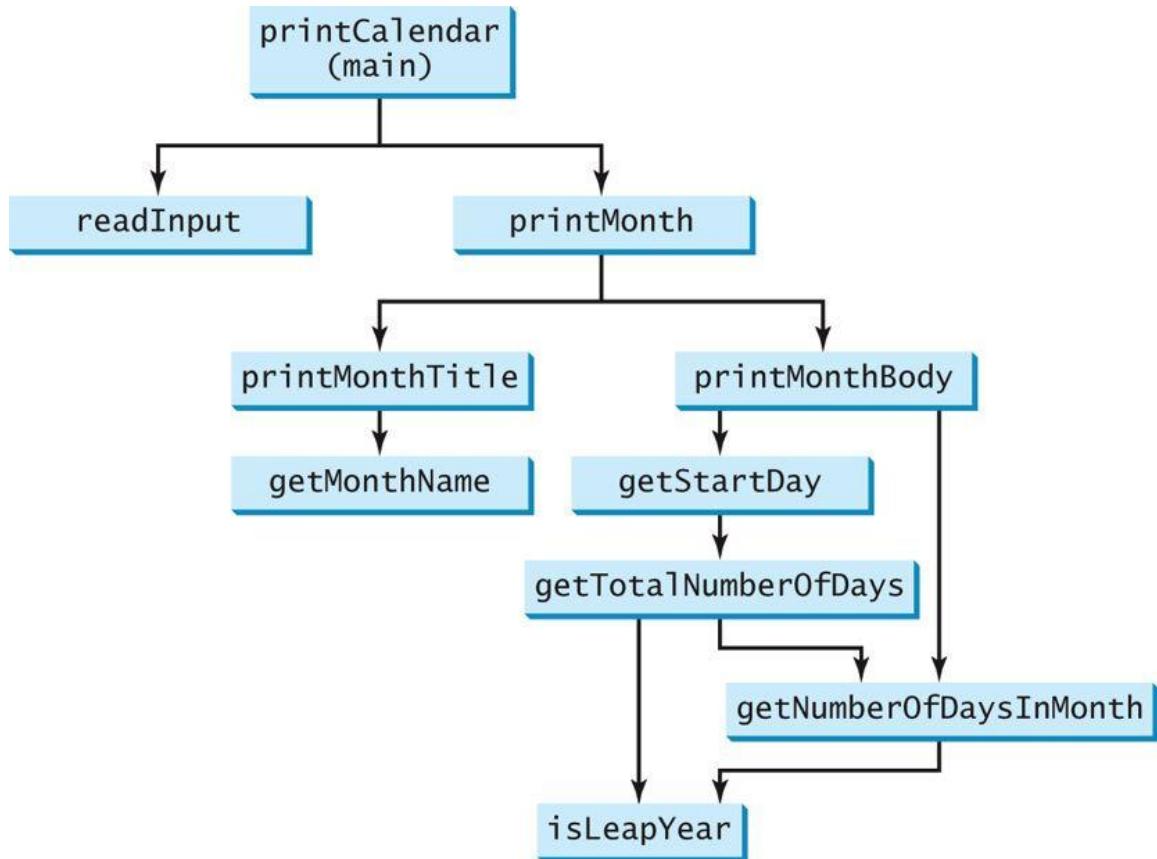
FIGURE 5.11 (a) To `getStartDay`, you need `getTotalNumberOfDays`. (b) The `getTotalNumberOfDays` problem is refined into two smaller problems.



How would you get the start day for the first date in a month? There are several ways to do so. For now, we'll use the following approach. Assume you know that the start day (`startDay1800 = 3`) for Jan 1, 1800, was Wednesday. You could compute the total number of days (`totalNumberOfDays`) between Jan 1, 1800, and the first date of the calendar month. The start day for the calendar month is `(totalNumberOfDays + startDay1800) % 7`, since every week has seven days. So the `getStartDay` problem can be further refined as `getTotalNumberOfDays`, as shown in [Figure 5.11\(a\)](#).

To get the total number of days, you need to know whether the year is a leap year and the number of days in each month. So `getTotalNumberOfDays` is further refined into two subproblems: `isLeapYear` and `getNumberOfDaysInMonth`, as shown in [Figure 5.11\(b\)](#). The complete structure chart is shown in [Figure 5.12](#).

FIGURE 5.12 The structure chart shows the hierarchical relationship of the subproblems in the program.



5.12.2 Top-Down or Bottom-Up Implementation

Now we turn our attention to implementation. In general, a subproblem corresponds to a method in the implementation, although some are so simple that this is unnecessary. You would need to decide which modules to implement as methods and which to combine in other methods. Decisions of this kind should be based on whether the overall program will be easier to read as a result of your choice. In this example, the subproblem `readInput` can be simply implemented in the `main` method.

You can use either a “top-down” or a “bottom-up” approach. The top-down approach implements one method in the structure chart at a time from the top to the bottom. Stubs

can be used for the methods waiting to be implemented. A *stub* is a simple but incomplete version of a method. The use of stubs enables you to quickly build the framework of the program. Implement the `main` method first, then use a stub for the `printMonth` method. For example, let `printMonth` display the year and the month in the stub. Thus, your program may begin like this:

top-down approach

stub

```
public class PrintCalendar {  
    /** Main method */  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
  
        // Prompt the user to enter year  
        System.out.print("Enter full year (e.g., 2001): ");  
        int year = input.nextInt();  
  
        // Prompt the user to enter month  
        System.out.print("Enter month as number between 1 and 12: ");  
        int month = input.nextInt();  
  
        // Print calendar for the month of the year  
        printMonth(year, month);  
    }  
  
    /** A stub for printMonth may look like this */  
    public static void printMonth(int year, int month) {  
        System.out.print(month + " " + year);  
    }  
  
    /** A stub for printMonthTitle may look like this */  
    public static void printMonthTitle(int year, int month) {  
    }  
  
    /** A stub for getMonthBody may look like this */  
    public static void printMonthBody(int year, int month) {  
    }  
  
    /** A stub for getMonthName may look like this */  
    public static String getMonthName(int month) {  
        return "January"; // A dummy value  
    }  
  
    /** A stub for getMonthName may look like this */  
    public static int getStartDay(int year, int month) {  
        return 1; // A dummy value  
    }  
  
    /** A stub for getTotalNumberOfDays may look like this */  
    public static int getTotalNumberOfDays(int year, int month) {  
        return 10000; // A dummy value  
    }  
  
    /** A stub for getNumberOfDaysInMonth may look like this */  
    public static int getNumberOfDaysInMonth(int year, int month) {  
        return 31; // A dummy value  
    }  
  
    /** A stub for getTotalNumberOfDays may look like this */  
    public static boolean isLeapYear(int year) {  
        return true; // A dummy value  
    }
```

Compile and test the program, and fix any errors. You can now implement the `printMonth` method. For methods invoked from the `printMonth` method, you can again use stubs.

The bottom-up approach implements one method in the structure chart at a time from the bottom to the top. For each method implemented, write a test program to test it. The top-down and bottom-up approaches are both fine. Both approaches implement methods incrementally, help to isolate programming errors, and make debugging easy. Sometimes they can be used together.

bottom-up approach

5.12.3 Implementation Details

The `isLeapYear(int year)` method can be implemented using the following code:

```
return (year % 400 == 0 || (year % 4 == 0 && year % 100  
!= 0));
```

Use the following facts to implement `getTotalNumberOfDaysInMonth(int year, int month)`:

- January, March, May, July, August, October, and December have 31 days.
- April, June, September, and November have 30 days.
- February has 28 days during a regular year and 29 days during a leap year. A regular year, therefore, has 365 days, a leap year 366 days.

To implement `getTotalNumberOfDays(int year, int month)`, you need to compute the total number of days (`totalNumberOfDays`) between January 1, 1800, and the first day of the calendar month. You could find the total number of days between the year 1800 and the calendar year and then figure out the total number of days prior to the calendar month in the calendar year. The sum of these two totals is `totalNumberOfDays`.

To print a body, first pad some space before the start day and then print the lines for every week.

The complete program is given in [Listing 5.12](#).

LISTING 5.12 PrintCalendar.java

```

1 import java.util.Scanner;
2
3 public class PrintCalendar {
4     /** Main method */
5     public static void main(String[] args) {
6         Scanner input = new Scanner(System.in);
7
8         // Prompt the user to enter year
9         System.out.print("Enter full year (e.g., 2001): ");
10        int year = input.nextInt();
11
12        // Prompt the user to enter month
13        System.out.print("Enter month in number between 1 and 12: ");
14        int month = input.nextInt();
15
16        // Print calendar for the month of the year
17        printMonth(year, month);
18    }
19
20    /** Print the calendar for a month in a year */
21    public static void printMonth(int year, int month) {           printMonth
22        // Print the headings of the calendar
23        printMonthTitle(year, month);
24
25        // Print the body of the calendar
26        printMonthBody(year, month);
27    }
28
29    /** Print the month title, e.g., May, 1999 */
30    public static void printMonthTitle(int year, int month) {       printMonthTitle
31        System.out.println(" " + getMonthName(month)
32            + " " + year);
33        System.out.println("-----");
34        System.out.println(" Sun Mon Tue Wed Thu Fri Sat");
35    }
36
37    /** Get the English name for the month */
38    public static String getMonthName(int month) {                  getMonthName
39        String monthName = " ";

```

```

40     switch (month) {
41         case 1: monthName = "January"; break;
42         case 2: monthName = "February"; break;
43         case 3: monthName = "March"; break;
44         case 4: monthName = "April"; break;
45         case 5: monthName = "May"; break;
46         case 6: monthName = "June"; break;
47         case 7: monthName = "July"; break;
48         case 8: monthName = "August"; break;
49         case 9: monthName = "September"; break;
50         case 10: monthName = "October"; break;
51         case 11: monthName = "November"; break;
52         case 12: monthName = "December";
53     }
54
55     return monthName;
56 }
57
58 /** Print month body */
59 public static void printMonthBody(int year, int month) {
60     // Get start day of the week for the first date in the month
61     int startDay = getStartDay(year, month);
62
63     // Get number of days in the month
64     int numberOfDaysInMonth = getNumberOfDaysInMonth(year, month);
65
66     // Pad space before the first day of the month
67     int i = 0;
68     for (i = 0; i < startDay; i++)
69         System.out.print(" ");
70
71     for (i = 1; i <= numberOfDaysInMonth; i++) {
72         System.out.printf("%d", i);
73
74         if ((i + startDay) % 7 == 0)
75             System.out.println();
76     }
77
78     System.out.println();
79 }
80
81 /** Get the start day of month/1/year */
82 public static int getStartDay(int year, int month) {
83     final int START_DAY_FOR_JAN_1_1800 = 3;
84     // Get total number of days from 1/1/1800 to month/1/year
85     int totalNumberOfDays = getTotalNumberOfDays(year, month);
86
87     // Return the start day for month/1/year
88     return (totalNumberOfDays + START_DAY_FOR_JAN_1_1800) % 7;
89 }
90
91 /** Get the total number of days since January 1, 1800 */
92 public static int getTotalNumberOfDays(int year, int month) {
93     int total = 0;
94
95     // Get the total days from 1800 to 1/1/year
96     for (int i = 1800; i < year; i++)
97         if (isLeapYear(i))
98
99             total = total + 366;
100        else
101            total = total + 365;
102
103        // Add days from Jan to the month prior to the calendar month
104        for (int i = 1; i < month; i++)
105            total = total + getNumberOfDaysInMonth(year, i);
106
107    return total;
108 }
109
110 /** Get the number of days in a month */
111 public static int getNumberOfDaysInMonth(int year, int month) {      getNumberOfDaysInMonth
112     if (month == 1 || month == 3 || month == 5 || month == 7 ||
113     month == 8 || month == 10 || month == 12)
114         return 31;
115
116     if (month == 4 || month == 6 || month == 9 || month == 11)
117         return 30;
118
119     if (month == 2) return isLeapYear(year) ? 29 : 28;
120
121     return 0; // If month is incorrect
122 }
123
124 /** Determine if it is a leap year */
125 public static boolean isLeapYear(int year) {                         isLeapYear
126     return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);
127 }

```

The program does not validate user input. For instance, if the user enters either a month not in the range between 1 and 12 or a year before 1800, the program displays an erroneous calendar. To avoid this error, add an `if` statement to check the input before printing the calendar.

This program prints calendars for a month but could easily be modified to print calendars for a whole year. Although it can print months only after January [1800](#), it could be modified to trace the day of a month before [1800](#).



Note

Method abstraction modularizes programs in a neat, hierarchical manner. Programs written as collections of concise methods are easier to write, debug, maintain, and modify. This writing style also promotes method reusability.

incremental development and testing



Tip

When implementing a large program, use the top-down or bottom-up approach. Do not write the entire program at once. Using these approaches seems to take more development time (because you repeatedly compile and run the program), but it actually saves time and makes debugging easier.

KEY TERMS

actual parameter [157](#)

argument [157](#)

ambiguous invocation [170](#)

divide and conquer [177](#)

formal parameter (i.e., parameter) [157](#)

information hiding [177](#)

method [156](#)

method abstraction [176](#)

method overloading [169](#)

method signature [157](#)

modifier [157](#)

pass-by-value [163](#)

parameter [157](#)

return type [170](#)

return value [157](#)

scope of variable [171](#)

stepwise refinement [177](#)

stub [179](#)

CHAPTER SUMMARY

1. Making programs modular and reusable is one of the central goals in software engineering. Java provides many powerful constructs that help to achieve this goal. Methods are one such construct.

2. The method header specifies the *modifiers*, *return value type*, *method name*, and *parameters* of the method. The static modifier is used for all the methods in this chapter.

3. A method may return a value. The `returnValueType` is the data type of the value the method returns. If the method does not return a value, the `returnValueType` is the keyword `void`.

4. The *parameter list* refers to the type, order, and number of the parameters of a method. The method name and the parameter list together constitute the *method signature*. Parameters are optional; that is, a method may contain no parameters.

5. A return statement can also be used in a `void` method for terminating the method and returning to the method's caller. This is useful occasionally for circumventing the normal flow of control in a method.

6. The arguments that are passed to a method should have the same number, type, and order as the parameters in the method signature.

7. When a program calls a method, program control is transferred to the called method. A called method returns control to the caller when its return statement is executed or when its method-ending closing brace is reached.

8. A value-returning method can also be invoked as a statement in Java. In this case, the caller simply ignores the return value.

9. Each time a method is invoked, the system stores parameters and local variables in a space known as a *stack*. When a method calls another method, the caller's stack space is kept intact, and new space is created to handle the new method call. When a method finishes its work and returns to its caller, its associated space is released.

10. A method can be overloaded. This means that two methods can have the same name, as long as their method parameter lists differ.

11. A variable declared in a method is called a local variable. The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable. A local variable must be declared and initialized before it is used.

12. *Method abstraction* is achieved by separating the use of a method from its implementation. The client can use a method without knowing how it is implemented. The details of the implementation are encapsulated in the method and hidden from the client who invokes the method. This is known as *information hiding* or *encapsulation*.

13. Method abstraction modularizes programs in a neat, hierarchical manner. Programs written as collections of concise methods are easier to write, debug, maintain, and modify than would otherwise be the case. This writing style also promotes method reusability.

14. When implementing a large program, use the top-down or bottom-up coding approach. Do not write the entire program at once. This approach seems to take more time for coding (because you are repeatedly compiling and running the program), but it actually saves time and makes debugging easier.

REVIEW QUESTIONS

Sections 5.2–5.4

5.1 What are the benefits of using a method? How do you define a method? How do you invoke a method?

5.2 What is the `return` type of a `main` method?

5.3 Can you simplify the `max` method in [Listing 5.1](#) using the conditional operator?

5.4 True or false? A call to a method with a `void` return type is always a statement itself, but a call to a value-returning method is always a component of an expression.

5.5 What would be wrong with not writing a `return` statement in a value-returning method? Can you have a `return` statement in a `void` method? Does the `return` statement in the following method cause syntax errors?

```
public static void xMethod(double x, double y) {  
    System.out.println(x + y);  
    return x + y;  
}
```

5.6 Define the terms parameter, argument, and method signature.

5.7 Write method headers for the following methods:

- Computing a sales commission, given the sales amount and the commission rate.
- Printing the calendar for a month, given the month and year.
- Computing a square root.
- Testing whether a number is even, and returning `true` if it is.
- Printing a message a specified number of times.
- Computing the monthly payment, given the loan amount, number of years, and annual interest rate.
- Finding the corresponding uppercase letter, given a lowercase letter.

5.8 Identify and correct the errors in the following program:

```
1 public class Test {  
2     public static method1(int n, m) {  
3         n += m;  
4         method2(3.4);  
5     }  
6  
7     public static int method2(int n) {  
8         if (n > 0) return 1;
```

```

9      else if (n == 0) return 0 ;
10     else if (n < 0) return -1 ;
11 }
12 }
```

5.9 Reformat the following program according to the programming style and documentation guidelines proposed in §2.16, “Programming Style and Documentation.” Use the next-line brace style.

```

public class Test {
    public static double method1(double i, double j)
    {
        while (i>j) {
            j--;
        }

        return j;
    }
}
```

Sections 5.5–5.7

5.10 How is an argument passed to a method? Can the argument have the same name as its parameter?

5.11 What is pass-by-value? Show the result of the following programs:

```

public class Test {
    public static void main(String[] args) {
        int max = 0;
        max(1, 2, max);
        System.out.println(max);
    }

    public static void max(
        int value1, int value2, int max) {
        if (value1 > value2) {
            max = value1;
        } else
            max = value2;
    }
}
```

```

public class Test {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 6) {
            method1(i, 2);
            i++;
        }
    }

    public static void method1(
        int i, int num) {
        for (int j = 1; j <= i; j++) {
            System.out.print(num + " ");
            num *= 2;
        }
        System.out.println();
    }
}
```

(a)

(b)

```

public class Test {
    public static void main(String[] args) {
        // Initialize times
        int times = 3;
        System.out.println("Before the call,"
            + " variable times is " + times);

        // Invoke nprintln and display times
        nprintln("Welcome to Java!", times);
        System.out.println("After the call,"
            + " variable times is " + times);
    }

    // Print the message n times
    public static void nprintln(
        String message, int n) {
        while (n > 0) {
            System.out.println("n = " + n);
            System.out.println(message);
            n--;
        }
    }
}

```

(c)

```

public class Test {
    public static void main(String[] args) {
        int i = 0;
        while (i <= 4) {
            method1(i);
            i++;
        }

        System.out.println("i is " + i);
    }

    public static void method1(int i) {
        do {
            if (i % 3 != 0)
                System.out.print(i + " ");
            i--;
        } while (i >= 1);
        System.out.println();
    }
}

```

(d)

- 5.12** For (a) in the preceding question, show the contents of the stack just before the method **max** is invoked, just as **max** is entered, just before **max** is returned, and right after **max** is returned.

Section 5.8

- 5.13** What is method overloading? Is it permissible to define two methods that have the same name but different parameter types? Is it permissible to define two methods in a class that have identical method names and parameter lists but different return value types or different modifiers?

- 5.14** What is wrong in the following program?

```

public class Test {
    public static void method(int x) {
    }
    public static int method(int y) {
        return y;
    }
}

```

Section 5.9

- 5.15** Identify and correct the errors in the following program:

```

1 public class Test {
2     public static void main(String[] args) {
3         nprintln("Welcome to Java!", 5);

```

```

4      }
5
6  public static void nPrintln(String message, int
n) {
7      int n = 1;

8      for (int i = 0; i < n; i++)
9          System.out.println(message);
10     }
11 }
```

Section 5.10

5.16 True or false? The argument for trigonometric methods represents an angle in radians.

5.17 Write an expression that returns a random integer between **34** and **55**. Write an expression that returns a random integer between **0** and **999**. Write an expression that returns a random number between **5.5** and **55.5**. Write an expression that returns a random lowercase letter.

5.18 Evaluate the following method calls:

- (a) Math.sqrt(**4**)
- (b) Math.sin(**2** * Math.PI)
- (c) Math.cos(**2** * Math.PI)
- (d) Math.pow(**2**, **2**)
- (e) Math.log(Math.E)
- (f) Math.exp(**1**)
- (g) Math.max(**2**, Math.min(**3**, **4**))
- (h) Math.rint(**-2.5**)
- (i) Math.ceil(**-2.5**)
- (j) Math.floor(**-2.5**)
- (k) Math.round(**-2.5F**)

- (l) Math.round(-2.5)
- (m) Math.rint(2.5)
- (n) Math.ceil(2.5)
- (o) Math.floor(2.5)
- (p) Math.round(2.5F)
- (q) Math.round(2.5)
- (r) Math.round(Math.abs(-2.5))

PROGRAMMING EXERCISES

Sections 5.2–5.9

5.1 (*Math: pentagonal numbers*) A pentagonal number is defined as $n(3n-1)/2$ for $n = 1, 2, \dots$, and so on. So, the first few numbers are 1, 5, 12, 22, Write the following method that returns a pentagonal number:

```
public static int getPentagonalNumber (int n)
```

Write a test program that displays the first 100 pentagonal numbers with 10 numbers on each line.

5.2* (*Summing the digits in an integer*) Write a method that computes the sum of the digits in an integer. Use the following method header:

```
public static int sumDigits (long n)
```

For example, `sumDigits (234)` returns 9 (2 + 3 + 4).

(*Hint:* Use the `%` operator to extract digits, and the `/` operator to remove the extracted digit. For instance, to extract 4 from 234, use `234 % 10` (=4). To remove 4 from 234, use `234 / 10` (=23). Use a loop to repeatedly extract and remove the digit until all the digits are extracted. Write a test program that prompts the user to enter an integer and displays the sum of all its digits.)

5.3** (*Palindrome integer*) Write the following two methods

```
// Return the reversal of an integer, i.e. reverse (45
6) returns 654
public static int reverse (int number)
// Return true if number is palindrome
```

```
public static boolean isPalindrome(int number)
```

Use the `reverse` method to implement `isPalindrome`. A number is a palindrome if its reversal is the same as itself. Write a test program that prompts the user to enter an integer and reports whether the integer is a palindrome.

5.4*

(*Displaying an integer reversed*) Write the following method to display an integer in reverse order:

```
public static void reverse(int number)
```

For example, `reverse(3456)` displays `6543`. Write a test program that prompts the user to enter an integer and displays its reversal.

5.5* (*Sorting three numbers*) Write the following method to display three numbers in increasing order:

```
public static void displaySortedNumbers(  
    double num1, double num2, double num3)
```

5.6* (*Displaying patterns*) Write a method to display a pattern as follows:

```
    1  
    2 1  
    3 2 1  
...  
n n-1 ... 3 2 1
```

The method header is

```
public static void displayPattern(int n)
```

5.7* (*Financial application: computing the future investment value*) Write a method that computes future investment value at a given interest rate for a specified number of years. The future investment is determined using the formula in Exercise 2.13. Use the following method header:

```
public static double futureInvestmentValue(  
    double investmentAmount, double  
    monthlyInterestRate, int years)
```

For example, `futureInvestmentValue(10000, 0.05/12, 5)` returns **12833.59**.

Write a test program that prompts the user to enter the investment amount (e.g., 1000) and the interest rate (e.g., 9%) and prints a table that displays future value for the years from 1 to 30, as shown below:

The amount invested: 1000

Annual interest rate: 9%	
Years	Future Value
1	1093.80
2	1196.41
...	
29	13467.25
30	14730.57

5.8 (Conversions between Celsius and Fahrenheit) Write a class that contains the following two methods:

```
/** Converts from Celsius to Fahrenheit */
public static double celsiusToFahrenheit(double celsius)
/** Converts from Fahrenheit to Celsius */
public static double fahrenheitToCelsius(double fahrenheit)
```

The formula for the conversion is:

$$\text{fahrenheit} = (9.0 / 5) * \text{celsius} + 32$$

Write a test program that invokes these methods to display the following tables:

Celsius

Fahrenheit

Fahrenheit

Celsius

40.0

104.0

120.0

48.89

39.0

102.2

110.0

43.33

...

32.0
89.6
40.0
4.44
31.0
87.8
30.0
-1.11

5.9 (*Conversions between feet and meters*) Write a class that contains the following two methods:

```
/** Converts from feet to meters */
public static double footToMeter(double foot)
/** Converts from meters to feet */
public static double meterToFoot(double meter)
```

The formula for the conversion is:

$\text{meter} = 0.305 * \text{foot}$

Write a test program that invokes these methods to display the following tables:

Feet

Meters

Meters

Feet

1.0

0.305

20.0

65.574

2.0

0.61

25.0

81.967

...

9.0

2.745

60.0

196.721

10.0

3.05

65.0

213.115

5.10 (*Using the `isPrime` method*) Listing 5.7, PrimeNumberMethod.java, provides the `isPrime(int number)` method for testing whether a number is prime. Use this method to find the number of prime numbers less than 10000.

5.11 (*Financial application: computing commissions*) Write a method that computes the commission, using the scheme in Exercise 4.39. The header of the method is as follows:

```
public static double computeCommission(double  
salesAmount)
```

Write a test program that displays the following table:

Sales Amount

Commission

10000

900.0

15000

1500.0

...

95000

11100.0

100000

11700.0

- 5.12** (*Displaying characters*) Write a method that prints characters using the following header:

```
public static void printChars(char ch1, char ch2, int  
    numberPerLine)
```

This method prints the characters between `ch1` and `ch2` with the specified numbers per line. Write a test program that prints ten characters per line from `&1&` to `&z&`.

- 5.13*** (*Summing series*) Write a method to compute the following series:

$$m(i) = \frac{1}{2} + \frac{2}{3} + \dots + \frac{i}{i+1}$$

Write a test program that displays the following table:

i

m(i)

1

0.5000

2

1.1667

...

19

16.4023

20

17.3546

5.14* (*Computing series*) Write a method to compute the following series:

$$m(i) = \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots + \frac{1}{2i-1} - \frac{1}{2i+1} \right)$$



Video Note

Compute π

Write a test program that displays the following table:

i

m(i)

10

3.04184

20

3.09162

...

90

3.13048

100

3.13159

5.15* (*Financial application: printing a tax table*) [Listing 3.6](#) gives a program to compute tax. Write a method for computing tax using the following header:

```
public static double computetax(int status, double  
taxableIncome)
```

Use this method to write a program that prints a tax table for taxable income from \$50,000 to \$60,000 with intervals of \$50 for all four statuses, as follows:

Taxable Income

Single

Married Joint

Married Separate

Head of a House

50000

8688

6665

8688

7353

50050

8700

6673

8700

7365

. . .

59950

11175

8158

11175

9840

60000

11188

8165

11188

9853

- 5.16*** (*Number of days in a year*) Write a method that returns the number of days in a year using the following header:

```
public static int numberOfDaysInAYear(int year)
```

Write a test program that displays the number of days in year from **2000** to **2010**.

Sections 5.10–5.11

- 5.17*** (*Displaying matrix of 0s and 1s*) Write a method that displays an **n**-by-**n** matrix using the following header:

```
public static void printMatrix(int n)
```

Each element is 0 or 1, which is generated randomly. Write a test program that prints a 3-by-3 matrix that may look like this:

0 1 0

0 0 0

1 1 1

- 5.18** (*Using the**Math.sqrt** method*) Write a program that prints the following table using the **sqrt** method in the **Math** class.

Number

SquareRoot

0

0.0000

2

1.4142

...

18

4.2426

20

4.4721

- 5.19*** (*TheMyTriangle class*) Create a class named **MyTriangle** that contains the following two methods:

```
/** Returns true if the sum of any two sides is
 * greater than the third side.*/
public static boolean isValid(
    double side1, double side2, double side3)
/** Returns the area of the triangle. */
public static double area(
    double side1, double side2, double side3)
```

Write a test program that reads three sides for a triangle and computes the area if the input is valid. Otherwise, it displays that the input is invalid. The formula for computing the area of a triangle is given in Exercise 2.21.

- 5.20** (*Using trigonometric methods*) Print the following table to display the **sin** value and **cos** value of degrees from 0 to 360 with increments of 10 degrees. Round the value to keep four digits after the decimal point.

Degree

Sin

Cos

0

0.0000

1.0000

10

0.1736

0.9848\

...

350

-0.1736

0.9848

360

0.0000

1.0000

5.21** (*Statistics: computing mean and standard deviation*) In business applications, you are often asked to compute the mean and standard deviation of data. The mean is simply the average of the numbers. The standard deviation is a statistic that tells you how tightly all the various data are clustered around the mean in a set of data. For example, what is the average age of the students in a class? How close are the ages? If all the students are the same age, the deviation is 0. Write a program that prompts the user to enter ten numbers, and displays the mean and standard deviations of these numbers using the following formula:

$$\text{mean} = \frac{\sum_{i=1}^n x_i}{n} = \frac{x_1 + x_2 + \dots + x_n}{n} \quad \text{deviation} = \sqrt{\frac{\sum_{i=1}^n x_i^2 - \frac{(\sum_{i=1}^n x_i)^2}{n}}{n - 1}}$$

Here is a sample run:



Enter ten numbers: 1 2 3 4. 5. 6 6 7 8 9 10



The mean is 5.61

The standard deviation is 2.99794

- 5.22**** (*Math: approximating the square root*) Implement the `sqrt` method. The square root of a number, `num`, can be approximated by repeatedly performing a calculation using the following formula:

```
nextGuess = (lastGuess + (num / lastGuess)) / 2
```

When `nextGuess` and `lastGuess` are almost identical, `nextGuess` is the approximated square root.

The initial guess can be any positive value (e.g., `1`). This value will be the starting value for `lastGuess`. If the difference between `nextGuess` and `lastGuess` is less than a very small number, such as `0.0001`, you can claim that `nextGuess` is the approximated square root of `num`. If not, `nextGuess` becomes `lastGuess` and the approximation process continues.

Sections 5.10–5.11

- 5.23*** (*Generating random characters*) Use the methods in `RandomCharacter` in [Listing 5.10](#) to print 100 uppercase letters and then 100 single digits, printing ten per line.

- 5.24**** (*Displaying current date and time*) [Listing 2.9](#), `ShowCurrentTime.java`, displays the current time. Improve this example to display the current date and time. The calendar example in [Listing 5.12](#), `PrintCalendar.java`, should give you some ideas on how to find year, month, and day.

- 5.25**** (*Converting milliseconds to hours, minutes, and seconds*) Write a method that converts milliseconds to hours, minutes, and seconds using the following header:

```
public static String convertMillis(long millis)
```

The method returns a string as hours:minutes:seconds. For example, `convertMillis(5500)` returns a string `0:0:5`, `convertMillis(100000)` returns a string `0:1:40`, and `convertMillis(555550000)` returns a string `154:19:10`.

Comprehensive

- 5.26**** (*Palindromic prime*) A *palindromic prime* is a prime number and also palindromic. For example, `131` is a prime and also a palindromic prime. So are `313` and `757`. Write a program that displays the first `100` palindromic prime numbers. Display `10` numbers per line and align the numbers properly, as follows:

2	3	5	7	11	101	131	151	181
191								
313	353	373	383	727	757	787	797	919
929								
...								

5.27** (*Emirp*) An *emirp* (prime spelled backward) is a nonpalindromic prime number whose reversal is also a prime. For example, **17** is a prime and **71** is a prime. So, **17** and **71** are emirps. Write a program that displays the first **100** emirps.

Display **10** numbers per line and align the numbers properly, as follows:

13	17	31	37	71	73	79	97	107	113
149	157	167	179	199	311	337	347	359	389
...									

5.28** (*Mersenne prime*) A prime number is called a *Mersenne prime* if it can be written in the form for some positive integer p . Write a program that finds all Mersenne primes with \leq and displays the output as follows:

```

p
2^p - 1
2
3
3
7
5
31
...

```

5.29** (*Game: craps*) Craps is a popular dice game played in casinos. Write a program to play a variation of the game, as follows:

Roll two dice. Each die has six faces representing values **1**, **2**, and **6**, respectively. Check the sum of the two dice. If the sum is **2**, **3**, or **12** (called *craps*), you lose; if the sum is **7** or **11** (called *natural*), you win; if the sum is another value (i.e., **4**, **5**, **6**, **8**, **9**, or **10**), a *point* is established. Continue to roll the dice until either a **7** or the same point value is rolled. If **7** is rolled, you lose. Otherwise, you win. Your program acts as a single player. Here are some sample runs.



You rolled 5 + 6 = 11

You win



You rolled 1 + 2 = 3

You lose



You rolled 4 + 4 = 8

point is 8

You rolled 6 + 2 = 8

You win



You rolled 3 + 2 = 5

point is 5

You rolled 2 + 5 = 7

You lose

5.30** (*Twin primes*) Twin primes are a pair of prime numbers that differ by 2. For example, 3 and 5 are twin primes, 5 and 7 are twin primes, and 11 and 13 are twin primes. Write a program to find all twin primes less than 1000. Display the output as follows:

(3, 5)
(5, 7)

...

5.31** (*Financial: credit card number validation*) Credit card numbers follow certain patterns. A credit card number must have between 13 and 16 digits. It must start with:

- 4 for Visa cards
- 5 for Master cards
- 37 for American Express cards
- 6 for Discover cards

In 1954, Hans Luhn of IBM proposed an algorithm for validating credit card numbers. The algorithm is useful to determine whether a card number is entered correctly or whether a credit card is scanned correctly by a scanner. All credit card numbers are generated following this validity check, commonly known as the *Luhn check* or the *Mod 10 check*, which can be described as follows (for illustration, consider the card number 4388576018402626):

1. Double every second digit from right to left. If doubling of a digit results in a two-digit number, add up the two digits to get a single-digit number.

$$2 * 2 = 4$$

$$2 * 2 = 4$$

$$4 * 2 = 8$$

$$1 * 2 = 2$$

$$6 * 2 = 12 \quad (1 + 2 = 3)$$

$$5 * 2 = 10 \quad (1 + 0 = 1)$$

$$8 * 2 = 16 \quad (1 + 6 = 7)$$

$$4 * 2 = 8$$

2. Now add all single-digit numbers from Step 1.

$$4 + 4 + 8 + 2 + 3 + 1 + 7 + 8 = 37$$

3. Add all digits in the odd places from right to left in the card number.

$$6 + 6 + 0 + 8 + 0 + 7 + 8 + 3 = 38$$

4. Sum the results from Step 2 and Step 3.

$$37 + 38 = 75$$

5. If the result from Step 4 is divisible by 10, the card number is valid; otherwise, it is invalid. For example, the number 4388576018402626 is invalid, but the number 4388576018410707 is valid.

Write a program that prompts the user to enter a credit card number as a **long** integer. Display whether the number is valid or invalid. Design your program to use the following methods:

```
/** Return true if the card number is valid */
public static boolean isValid(long number)
/** Get the result from Step 2 */
public static int sumOfDoubleEvenPlace(long number)
/** Return this number if it is a single digit, otherwise,
 *  return
 *  the sum of the two digits */
public static int getDigit(int number)
/** Return sum of odd place digits in number */
public static int sumOfOddPlace(long number)
/** Return true if the digit d is a prefix for number
 */
public static boolean prefixMatched(long number, (int
    d)
/** Return the number of digits in d */
public static int getSize((long d)
/** Return the first k number of digits from number.
 * If the
 *  number of digits in number is less than k, return
 *  number. */
public static long getPrefix(long number, int k)
```

5.32** (*Game: chance of winning at craps*) Revise Exercise 5.29 to run it 10000 times and display the number of winning games.

5.33*** (*Current date and time*) Invoking **System.currentTimeMillis()** returns the elapse time in milliseconds since midnight of January 1, 1970. Write a program that displays the date and time. Here is a sample run:



Current date and time is May 16, 2009 10:34:23

5.34** (*Printing calendar*) Exercise 3.21 uses Zeller's congruence to calculate the day of the week. Simplify [Listing 5.12](#), PrintCalendar.java, using Zeller's algorithm to get the start day of the month.

5.35 (*Geometry: area of a pentagon*) The area of a pentagon can be computed using the following formula:

$$\text{Area} = \frac{5 \times s^2}{4 \times \tan\left(\frac{\pi}{5}\right)}$$

Write a program that prompts the user to enter the side of a pentagon and displays its area.

5.36* (*Geometry: area of a regular polygon*) A regular polygon is an n-sided polygon in which all sides are of the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). The formula for computing the area of a regular polygon is

$$\text{Area} = \frac{n \times s^2}{4 \times \tan\left(\frac{\pi}{n}\right)}.$$

Write a method that returns the area of a regular polygon using the following header:

```
public static double area(int n, double side)
```

Write a main method that prompts the user to enter the number of sides and the side of a regular polygon and displays its area.

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 155).
<vbk:9781256335153#outline(9)>

CHAPTER 8 OBJECTS AND CLASSES

Objectives

- To describe objects and classes, and use classes to model objects ([§8.2](#))
- To use UML graphical notations to describe classes and objects ([§8.2](#))
- To demonstrate defining classes and creating objects ([§8.3](#)).
- To create objects using constructors ([§8.4](#)).
- To access objects via object reference variables ([§8.5](#)).
- To define a reference variable using a reference type ([§8.5.1](#)).
- To access an object's data and methods using the object member access operator (.) ([§8.5.2](#)).
- To define data fields of reference types and assign default values for an object's data fields ([§8.5.3](#)).
- To distinguish between object reference variables and primitive data type variables ([§8.5.4](#)).
- To use classes `Date`, `Random`, and `JFrame` in the Java library ([§8.6](#)).
- To distinguish between instance and static variables and methods ([§8.7](#)).
- To define private data fields with appropriate `get` and `set` methods ([§8.8](#)).
- To encapsulate data fields to make classes easy to maintain ([§8.9](#)).
- To develop methods with object arguments and differentiate between primitive-type arguments and object-type arguments ([§8.10](#)).
- To store and process objects in arrays ([§8.11](#)).

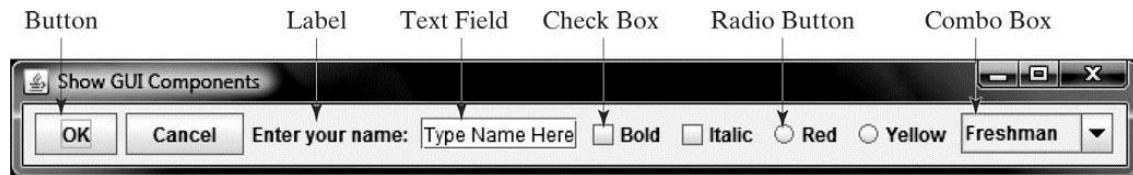
8.1 Introduction

Having learned the material in earlier chapters, you are able to solve many programming problems using selections, loops, methods, and arrays. However, these Java features are not sufficient for developing graphical user interfaces and large-scale software systems. Suppose

you want to develop a GUI (graphical user interface, pronounced *goo-ee*) as shown in [Figure 8.1](#). How do you program it?

why OOP?

FIGURE 8.1 The GUI objects are created from classes.



This chapter begins the introduction of object-oriented programming, which will enable you to develop GUI and large-scale software systems effectively.

8.2 Defining Classes for Objects

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects. An object has a unique identity, state, and behavior.

Object

- The *state* of an object (also known as its *properties* or *attributes*) is represented by *data fields* with their current values. A circle object, for example, has a data field `radius`, which is the property that characterizes a circle. A rectangle object has data fields `width` and `height`, which are the properties that characterize a rectangle.

state

- The *behavior* of an object (also known as its *actions*) is defined by methods. To invoke a method on an object is to ask the object to perform an action. For example, you may define a method named `getArea()` for circle objects. A circle object may invoke `getArea()` to return its area.

behavior

Objects of the same type are defined using a common class. A class is a template, blueprint, or *contract* that defines what an object's data fields and methods will be. An object is an instance of a class. You can create many instances of a class. Creating an instance is referred to as *instantiation*. The terms *object* and *instance* are often interchangeable. The relationship

between classes and objects is analogous to that between an apple-pie recipe and apple pies. You can make as many apple pies as you want from a single recipe. [Figure 8.2](#) shows a class named **Circle** and its three objects.

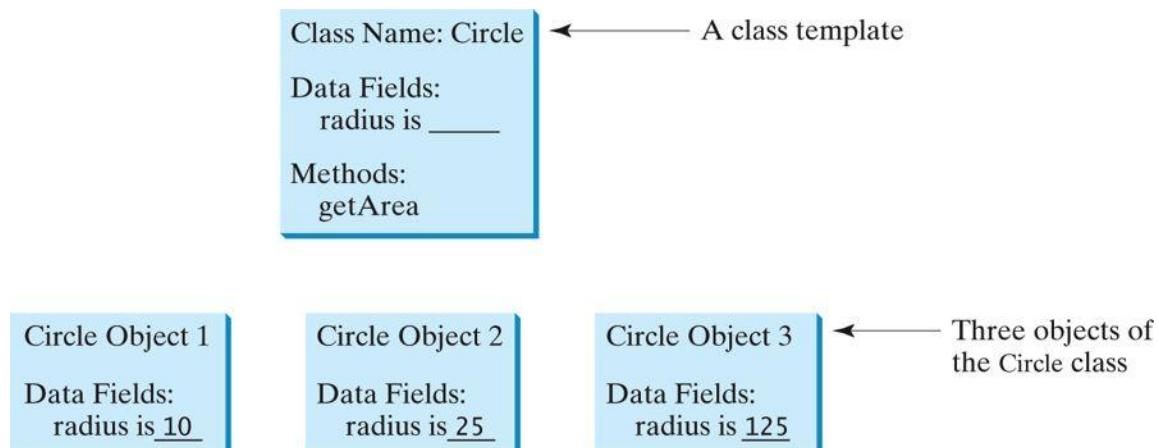
contract

instantiation

object

instance

FIGURE 8.2 A class is a template for creating objects.



A Java class uses variables to define data fields and methods to define actions. Additionally, a class provides methods of a special type, known as *constructors*, which are invoked to create a new object. A constructor can perform any action, but constructors are designed to perform initializing actions, such as initializing the data fields of objects. [Figure 8.3](#) shows an example of defining the class for circle objects.

class

data field

method

constructor

FIGURE 8.3 A class is a construct that defines objects of the same type.

```

class Circle {
    /** The radius of this circle */
    double radius = 1.0; ← Data field

    /** Construct a circle object */
    Circle() {
    }

    /** Construct a circle object */
    Circle(double newRadius) {
        radius = newRadius;
    }

    /** Return the area of this circle */
    double getArea() {
        return radius * radius * Math.PI;
    }
}

```

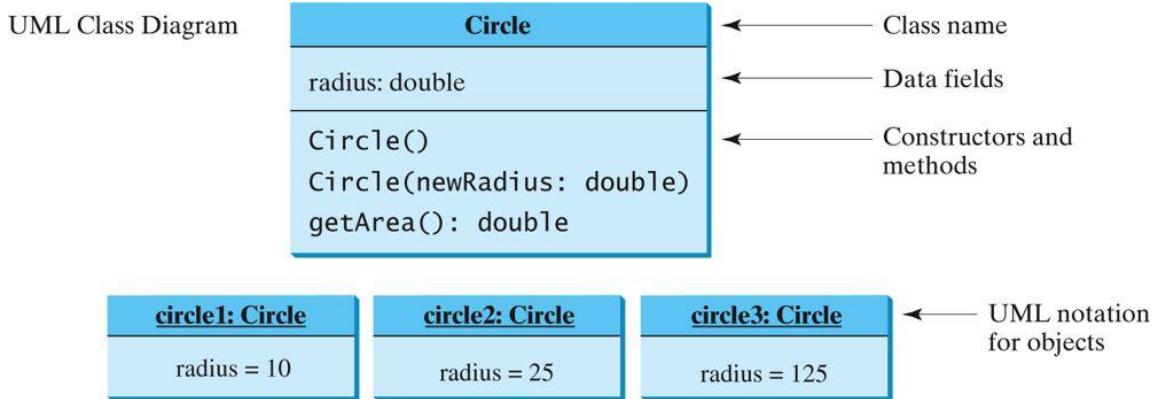
The `Circle` class is different from all of the other classes you have seen thus far. It does not have a `main` method and therefore cannot be run; it is merely a definition for circle objects. The class that contains the `main` method will be referred to in this book, for convenience, as the *main class*.

main class

The illustration of class templates and objects in [Figure 8.2](#) can be standardized using UML (Unified Modeling Language) notations. This notation, as shown in [Figure 8.4](#), is called a *UML class diagram*, or simply a *class diagram*. In the class diagram, the data field is denoted as

class diagram
dataFieldName: dataType

FIGURE 8.4 Classes and objects can be represented using UML notations.



The constructor is denoted as

```
ClassName (parameterName: parameterType)
```

The method is denoted as

```
methodName (parameterName: parameterType) : returnType
```

8.3 Example: Defining Classes and Creating Objects

This section gives two examples of defining classes and uses the classes to create objects.

[Listing 8.1](#) is a program that defines the **Circle** class and uses it to create objects. To avoid a naming conflict with several improved versions of the **Circle** class introduced later in this book, the **Circle** class in this example is named **Circle1**.

The program constructs three circle objects with radius **1.0**, **25**, and **125** and displays the radius and area of each of the three circles. Change the radius of the second object to **100** and display its new radius and area.

LISTING 8.1 TestCircle1.java

```

main class
1 public class TestCircle1 {
2     /** Main method */
3     public static void main(String[] args) {
4         // Create a circle with radius 1.0
5         Circle1 circle1 = new Circle1();
6         System.out.println("The area of the circle of radius "
7             + circle1.radius + " is " + circle1.getArea());
8
9         // Create a circle with radius 25
10        Circle1 circle2 = new Circle1(25);
11        System.out.println("The area of the circle of radius "
12            + circle2.radius + " is " + circle2.getArea());
13
14        // Create a circle with radius 125
15        Circle1 circle3 = new Circle1(125);
16        System.out.println("The area of the circle of radius "
17            + circle3.radius + " is " + circle3.getArea());
18
19        // Modify circle radius
20        circle2.radius = 100;
21        System.out.println("The area of the circle of radius "
22            + circle2.radius + " is " + circle2.getArea());
23    }
24 }
25
26 // Define the circle class with two constructors
27 class Circle1 {
28     double radius;
29
30     /** Construct a circle with radius 1 */
31     Circle1() {
32         radius = 1.0;
33     }
34
35     /** Construct a circle with a specified radius */
36     Circle1(double newRadius) {
37         radius = newRadius;
38     }
39
40     /** Return the area of this circle */
41     double getArea() {
42         return radius * radius * Math.PI;
43     }
44 }
```



The area of the circle of radius 1.0 is 3.141592653589793

The area of the circle of radius 25.0 is 1963.4954084936207

The area of the circle of radius 125.0 is 49087.385212340516

The area of the circle of radius 100.0 is 31415.926535897932

The program contains two classes. The first of these, `TestCircle1`, is the main class. Its sole purpose is to test the second class, `Circle1`. Such a program that uses the class is often referred to as a *client* of the class. When you run the program, the Java runtime system invokes the `main` method in the main class.

client

You can put the two classes into one file, but only one class in the file can be a public class. Furthermore, the public class must have the same name as the file name. Therefore, the file name is `TestCircle1.java`, since `TestCircle1` is public.

public class

The main class contains the `main` method (line 3) that creates three objects. As in creating an array, the `new` operator is used to create an object from the constructor. `new Circle1()` creates an object with radius `1.0` (line 5), `new Circle1(25)` creates an object with radius `25` (line 10), and `new Circle1(125)` creates an object with radius `125` (line 15).

These three objects (referenced by `circle1`, `circle2`, and `circle3`) have different data but the same methods. Therefore, you can compute their respective areas by using the `getArea()` method. The data fields can be accessed via the reference of the object using `circle1.radius`, `circle2.radius`, and `circle3.radius`, respectively. The object can invoke its method via the reference of the object using `circle1.getArea()`, `circle2.getArea()`, and `circle3.getArea()`, respectively.

These three objects are independent. The radius of `circle2` is changed to `100` in line 20. The object's new radius and area is displayed in lines 21–22.

There are many ways to write Java programs. For instance, you can combine the two classes in the example into one, as shown in [Listing 8.2](#).

LISTING 8.2 Circle1.java

```

1 public class Circle1 {
2     /** Main method */
3     public static void main(String[] args) {                                main method
4         // Create a circle with radius 1.0
5         Circle1 circle1 = new Circle1();
6         System.out.println("The area of the circle of radius "
7             + circle1.radius + " is " + circle1.getArea());
8
9         // Create a circle with radius 25
10        Circle1 circle2 = new Circle1(25);
11        System.out.println("The area of the circle of radius "
12            + circle2.radius + " is " + circle2.getArea());
13
14        // Create a circle with radius 125
15        Circle1 circle3 = new Circle1(125);
16        System.out.println("The area of the circle of radius "
17            + circle3.radius + " is " + circle3.getArea());
18
19        // Modify circle radius
20        circle2.radius = 100;
21        System.out.println("The area of the circle of radius "
22            + circle2.radius + " is " + circle2.getArea());
23    }
24

data field
25     double radius;
26
27     /** Construct a circle with radius 1 */
no-arg constructor
28     Circle1() {
29         radius = 1.0;
30     }
31
32     /** Construct a circle with a specified radius */
second constructor
33     Circle1(double newRadius) {
34         radius = newRadius;
35     }
36
37     /** Return the area of this circle */
method
38     double getArea() {
39         return radius * radius * Math.PI;
40     }
41 }
```

Since the combined class has a **main** method, it can be executed by the Java interpreter. The **main** method is the same as in [Listing 1.1](#). This demonstrates that you can test a class by simply adding a main method in the same class.

As another example, consider TV sets. Each TV is an object with states (current channel, current volume level, power on or off) and behaviors (change channels, adjust volume, turn on/off). You can use a class to model TV sets. The UML diagram for the class is shown in [Figure 8.5](#).

[Listing 8.3](#) gives a program that defines the **TV** class.

FIGURE 8.5 The TV class models TV sets.

TV	
channel: int	The current channel (1 to 120) of this TV.
volumeLevel: int	The current volume level (1 to 7) of this TV.
on: boolean	Indicates whether this TV is on/off.
+TV()	Constructs a default TV object.
+turnOn(): void	Turns on this TV.
+turnOff(): void	Turns off this TV.
+setChannel(newChannel: int): void	Sets a new channel for this TV.
+setVolume(newVolumeLevel: int): void	Sets a new volume level for this TV.
+channelUp(): void	Increases the channel number by 1.
+channelDown(): void	Decreases the channel number by 1.
+volumeUp(): void	Increases the volume level by 1.
+volumeDown(): void	Decreases the volume level by 1.

LISTING 8.3 TV.java

```
1 public class TV {  
2     int channel = 1; // Default channel is 1  
3     int volumeLevel = 1; // Default volume level is 1  
4     boolean on = false; // By default TV is off  
5  
6     public TV() {  
7     }  
8 }
```

```

9  public void turnOn() {                                turn on TV
10     on = true;
11 }
12
13 public void turnOff() {                               turn off TV
14     on = false;
15 }
16
17 public void setChannel(int newChannel) {             set a new channel
18     if (on && newChannel >= 1 && newChannel <= 120)
19         channel = newChannel;
20 }
21
22 public void setVolume(int newVolumeLevel) {          set a new volume
23     if (on && newVolumeLevel >= 1 && newVolumeLevel <= 7)
24         volumeLevel = newVolumeLevel;
25 }
26
27 public void channelUp() {                            increase channel
28     if (on && channel < 120)
29         channel++;
30 }
31
32 public void channelDown() {                          decrease channel
33     if (on && channel > 1)
34         channel--;
35 }
36
37 public void volumeUp() {                            increase volume
38     if (on && volumeLevel < 7)
39         volumeLevel++;
40 }
41
42 public void volumeDown() {                          decrease volume
43     if (on && volumeLevel > 1)
44         volumeLevel--;
45 }
46 }

```

Note that the channel and volume level are not changed if the TV is not on. Before either of these is changed, its current value is checked to ensure that it is within the correct range.

[Listing 8.4](#) gives a program that uses the **TV** class to create two objects.

LISTING 8.4 TestTV.java

```

1 public class TestTV {
2     public static void main(String[] args) {
3         TV tv1 = new TV();
4         tv1.turnOn();
5         tv1.setChannel(30);
6         tv1.setVolume(3);
7
8         TV tv2 = new TV();
9         tv2.turnOn();
10        tv2.channelUp();
11        tv2.channelUp();
12        tv2.volumeUp();
13
14        System.out.println("tv1's channel is " + tv1.channel
15                      + " and volume level is " + tv1.volumeLevel);
16
17        System.out.println("tv2's channel is " + tv2.channel
18                      + " and volume level is " + tv2.volumeLevel);
19    }

```



tv1's channel is 30 and volume level is 3

tv2's channel is 3 and volume level is 2

The program creates two objects in lines 3 and 8 and invokes the methods on the objects to perform actions for setting channels and volume levels and for increasing channels and volumes. The program displays the state of the objects in lines 14–17. The methods are invoked using a syntax such as `tv1.turnOn()` (line 4). The data fields are accessed using a syntax such as `tv1.channel` (line 14).

These examples have given you a glimpse of classes and objects. You may have many questions regarding constructors, objects, reference variables, and accessing data fields, and invoking object's methods. The sections that follow discuss these issues in detail.

8.4 Constructing Objects Using Constructors

Constructors are a special kind of method. They have three peculiarities:

- A constructor must have the same name as the class itself.

constructor's name

- Constructors do not have a return type—not even `void`.

no return type

- Constructors are invoked using the `new` operator when an object is created. Constructors play the role of initializing objects.

new operator

The constructor has exactly the same name as the defining class. Like regular methods, constructors can be overloaded (i.e., multiple constructors can have the same name but different signatures), making it easy to construct objects with different initial data values.

overloaded constructors

It is a common mistake to put the `void` keyword in front of a constructor. For example,

no void

```
public void Circle() {  
}
```

In this case, `Circle()` is a method, not a constructor.

Constructors are used to construct objects. To construct an object from a class, invoke a constructor of the class using the `new` operator, as follows:

constructing objects

```
new ClassName(arguments);
```

For example, `new Circle()` creates an object of the `Circle` class using the first constructor defined in the `Circle` class, and `new Circle(25)` creates an object using the second constructor defined in the `Circle` class.

A class normally provides a constructor without arguments (e.g., `Circle()`). Such a constructor is referred to as a *no-arg* or *no-argument constructor*.

no-arg constructor

A class may be defined without constructors. In this case, a no-arg constructor with an empty body is implicitly defined in the class. This constructor, called a *default constructor*, is provided automatically *only if no constructors are explicitly defined in the class*.

default constructor

8.5 Accessing Objects via Reference Variables

Newly created objects are allocated in the memory. They can be accessed via reference variables.

8.5.1 Reference Variables and Reference Types

Objects are accessed via object *reference variables*, which contain references to the objects. Such variables are declared using the following syntax:

reference variable

```
ClassName objectRefVar;
```

A class is essentially a programmer-defined type. A class is a *reference type*, which means that a variable of the class type can reference an instance of the class. The following statement declares the variable `myCircle` to be of the `Circle` type:

reference type

```
Circle myCircle;
```

The variable `myCircle` can reference a `Circle` object. The next statement creates an object and assigns its reference to `myCircle`:

```
myCircle = new Circle();
```

Using the syntax shown below, you can write a single statement that combines the declaration of an object reference variable, the creation of an object, and the assigning of an object reference to the variable.

```
ClassName objectRefVar = new ClassName();
```

Here is an example:

```
Circle myCircle = new Circle();
```

The variable `myCircle` holds a reference to a `Circle` object.



Note

An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored. So it is fine, for simplicity, to say that

`myCircle` is a `Circle` object rather than use the longer-winded description that `myCircle` is a variable that contains a reference to a `Circle` object.

object vs. object reference variable



Note

Arrays are treated as objects in Java. Arrays are created using the `new` operator. An array variable is actually a variable that contains a reference to an array.

array object

8.5.2 Accessing an Object's Data and Methods

After an object is created, its data can be accessed and its methods invoked using the dot operator (.), also known as the *object member access operator*:

- `objectRefVar.dataField` references a data field in the object.
- `objectRefVar.method(arguments)` invokes a method on the object.

dot operator

For example, `myCircle.radius` references the radius in `myCircle`, and `myCircle.getArea()` invokes the `getArea` method on `myCircle`. Methods are invoked as operations on objects.

The data field `radius` is referred to as an *instance variable*, because it is dependent on a specific instance. For the same reason, the method `getArea` is referred to as an *instance method*, because you can invoke it only on a specific instance. The object on which an instance method is invoked is called a *calling object*.

instance variable

instance method

calling object



Caution

Recall that you use `Math.methodName(arguments)` (e.g., `Math.pow(3, 2.5)`) to invoke a method in the `Math` class. Can you invoke `getArea()` using `Circle.getArea()`? The answer is no. All the methods in the `Math` class are static methods, which are defined using the `static` keyword. However, `getArea()` is an instance method, and thus nonstatic. It must be invoked from an object using `objectRefVar.methodName(arguments)` (e.g., `myCircle.getArea()`). Further explanation is given in §8.7, “Static Variables, Constants, and Methods.”

invoking methods



Note

Usually you create an object and assign it to a variable. Later you can use the variable to reference the object. Occasionally an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable, as shown below:

```
new Circle();
```

or

```
System.out.println("Area is " + new Circle(5).getArea());
```

The former statement creates a `Circle` object. The latter creates a `Circle` object and invokes its `getArea` method to return its area. An object created in this way is known as an *anonymous object*.

anonymous object

8.5.3 Reference Data Fields and the `null` Value

The data fields can be of reference types. For example, the following `Student` class contains a data field `name` of the `String` type. `String` is a predefined Java class.

reference data fields

```
class Student {  
    String name; // name has default value null  
    int age; // age has default value 0  
    boolean isScienceMajor; // isScienceMajor has default  
    value false  
    char gender; // c has default value &\u0000&
```

```
}
```

If a data field of a reference type does not reference any object, the data field holds a special Java value, `null`. `null` is a literal just like `true` and `false`. While `true` and `false` are Boolean literals, `null` is a literal for a reference type.

`null` value

The default value of a data field is `null` for a reference type, `0` for a numeric type, `false` for a `boolean` type, and `\u0000` for a `char` type. However, Java assigns no default value to a local variable inside a method. The following code displays the default values of data fields `name`, `age`, `isScienceMajor`, and `gender` for a `Student` object:

default field values

```
class Test {  
    public static void main(String[] args) {  
        Student student = new Student();  
        System.out.println("name? " + student.name);  
        System.out.println("age? " + student.age);  
        System.out.println("isScienceMajor? " + student.isScienceMajor);  
        System.out.println("gender? " + student.gender);  
    }  
}
```

The code below has a compile error, because local variables `x` and `y` are not initialized:

```
class Test {  
    public static void main(String[] args) {  
  
        int x; // x has no default value  
        String y; // y has no default value  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}
```



Caution

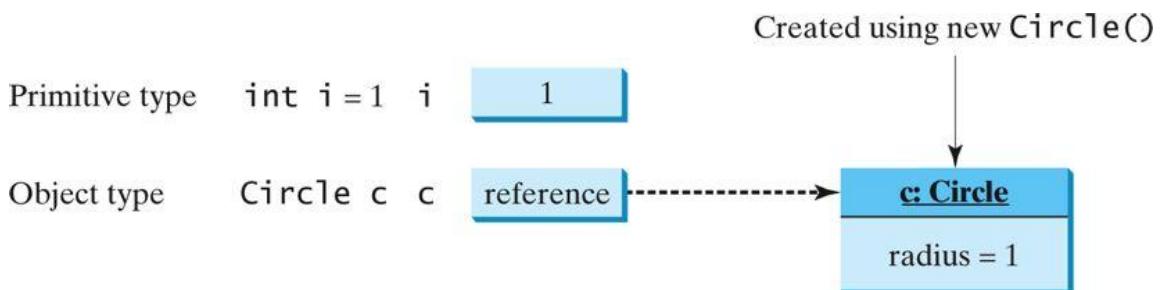
NullPointerException is a common runtime error. It occurs when you invoke a method on a reference variable with **null** value. Make sure you assign an object reference to the variable before invoking the method through the reference variable.

NullPointerException

8.5.4 Differences Between Variables of Primitive Types and Reference Types

Every variable represents a memory location that holds a value. When you declare a variable, you are telling the compiler what type of value the variable can hold. For a variable of a primitive type, the value is of the primitive type. For a variable of a reference type, the value is a reference to where an object is located. For example, as shown in [Figure 8.6](#), the value of **int** variable **i** is **int** value **1**, and the value of **Circle** object **c** holds a reference to where the contents of the **Circle** object are stored in the memory.

FIGURE 8.6 A variable of a primitive type holds a value of the primitive type, and a variable of a reference type holds a reference to where an object is stored in memory.



When you assign one variable to another, the other variable is set to the same value. For a variable of a primitive type, the real value of one variable is assigned to the other variable. For a variable of a reference type, the reference of one variable is assigned to the other variable. As shown in [Figure 8.7](#), the assignment statement **i=j** copies the contents of **j** into **i** for primitive variables. As shown in [Figure 8.8](#), the assignment statement **c1 = c2** copies the reference of **c2** into **c1** for reference variables. After the assignment, variables **c1** and **c2** refer to the same object.

FIGURE 8.7 Primitive variable **j is copied to variable **i**.**

Primitive type assignment $i = j$

Before:



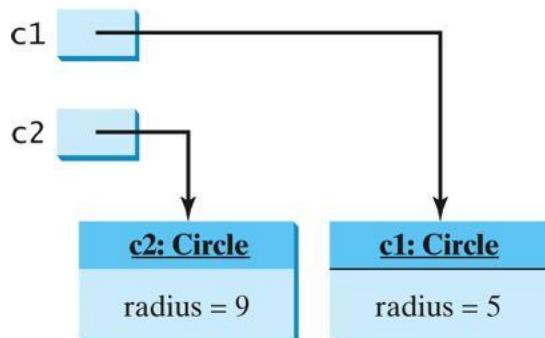
After:



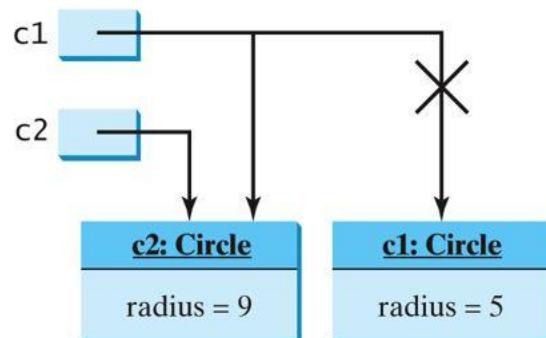
FIGURE 8.8 Reference variable $c2$ is copied to variable $c1$.

Object type assignment $c1 = c2$

Before:



After:



Note

As shown in [Figure 8.8](#), after the assignment statement `c1 = c2`, `c1` points to the same object referenced by `c2`. The object previously referenced by `c1` is no longer useful and therefore is now known as *garbage*. Garbage occupies memory space. The Java runtime system detects garbage and automatically reclaims the space it occupies. This process is called *garbage collection*.

garbage

garbage collection



Tip

If you know that an object is no longer needed, you can explicitly assign `null` to a reference variable for the object. The JVM will automatically collect the space if the object is not referenced by any reference variable.

8.6 Using Classes from the Java Library

[Listing 8.1](#) defined the `Circle1` class and created objects from the class. You will frequently use the classes in the Java library to develop programs. This section gives some examples of the classes in the Java library.

8.6.1 The Date Class

In [Listing 2.8](#), `ShowCurrentTime.java`, you learned how to obtain the current time using `System.currentTimeMillis()`. You used the division and remainder operators to extract current second, minute, and hour. Java provides a system-independent encapsulation of date and time in the `java.util.Date` class, as shown in [Figure 8.9](#).

`java.util.Date` class

FIGURE 8.9 A Date object represents a specific date and time.

java.util.Date	
The + sign indicates public modifier →	<code>+Date()</code> <code>+Date(elapseTime: long)</code> <code>+toString(): String</code> <code>+getTime(): long</code> <code>+setTime(elapseTime: long): void</code>

Constructs a `Date` object for the current time.
Constructs a `Date` object for a given time in milliseconds elapsed since January 1, 1970, GMT.
Returns a string representing the date and time.
Returns the number of milliseconds since January 1, 1970, GMT.
Sets a new elapse time in the object.

You can use the no-arg constructor in the `Date` class to create an instance for the current date and time, its `getTime()` method to return the elapsed time since January 1, 1970, GMT, and its `toString` method to return the date and time as a string. For example, the following code

```

java.util.Date date = new java.util.Date();
System.out.println("The elapsed time since Jan 1, 1970 is " +
    date.getTime() + " milliseconds");
System.out.println(date.toString());

```

create object
get elapsed time
invoke **toString**

displays the output like this:

```

The elapsed time since Jan 1, 1970 is 1100547210284
milliseconds
Mon Nov 15 14:33:30 EST 2004

```

The **Date** class has another constructor, **Date(long elapseTime)**, which can be used to construct a **Date** object for a given time in milliseconds elapsed since January 1, 1970, GMT.

8.6.2 The Random Class

You have used **Math.random()** to obtain a random **double** value between **0.0** and **1.0** (excluding **1.0**). Another way to generate random numbers is to use the **java.util.Random** class, as shown in [Figure 8.10](#), which can generate a random **int**, **long**, **double**, **float**, and **boolean** value.

FIGURE 8.10 A Random object can be used to generate random values.

java.util.Random	
+Random()	Constructs a Random object with the current time as its seed.
+Random(seed: long)	Constructs a Random object with a specified seed.
+nextInt(): int	Returns a random int value.
+nextInt(n: int): int	Returns a random int value between 0 and n (exclusive).
+nextLong(): long	Returns a random long value.
+nextDouble(): double	Returns a random double value between 0.0 and 1.0 (exclusive).
+nextFloat(): float	Returns a random float value between 0.0F and 1.0F (exclusive).
+nextBoolean(): boolean	Returns a random boolean value.

When you create a **Random** object, you have to specify a seed or use the default seed. The no-arg constructor creates a **Random** object using the current elapsed time as its seed. If two **Random** objects have the same seed, they will generate identical sequences of numbers. For example, the following code creates two **Random** objects with the same seed, **3**.

```

Random random1 = new Random(3);
System.out.print("From random1: ");
for (int i = 0; i < 10; i++)
    System.out.print(random1.nextInt(1000) + " ");
Random random2 = new Random(3);

```

```
System.out.print("\nFrom random2: ");
for (int i = 0; i < 10; i++)
    System.out.print(random2.nextInt(1000) + " ");
```

The code generates the same sequence of random `int` values:

```
From random1: 734 660 210 581 128 202 549 564 459 961
From random2: 734 660 210 581 128 202 549 564 459 961
```



Note

The ability to generate the same sequence of random values is useful in software testing and many other applications. In software testing, you can test your program using a fixed sequence of numbers before using different sequences of random numbers.

same sequence

8.6.3 Displaying GUI Components

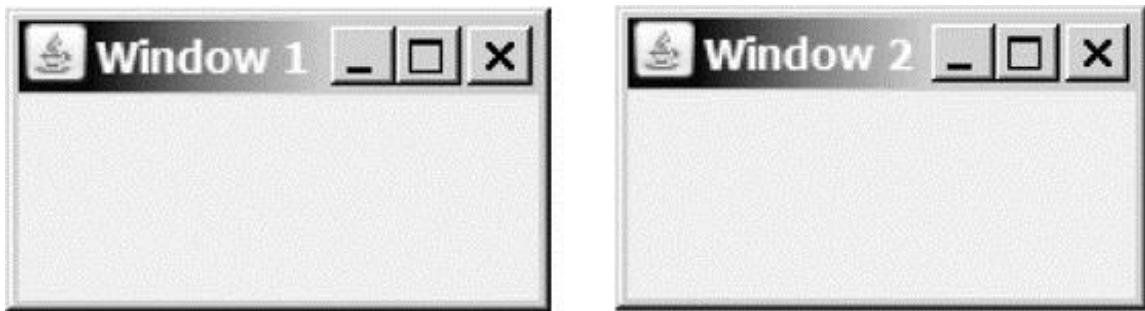


Pedagogical Note

Graphical user interface (GUI) components are good examples for teaching OOP. Simple GUI examples are introduced for this purpose. The full introduction to GUI programming begins with Chapter 12, “GUI Basics.”

When you develop programs to create graphical user interfaces, you will use Java classes such as `JFrame`, `JButton`, `JRadioButton`, `JComboBox`, and `JList` to create frames, buttons, radio buttons, combo boxes, lists, and so on. [Listing 8.5](#) is an example that creates two windows using the `JFrame` class. The output of the program is shown in [Figure 8.11](#).

FIGURE 8.11 The program creates two windows using the `JFrame` class.



LISTING 8.5 TestFrame.java

```
1 import javax.swing.JFrame;
2
3 public class TestFrame {
4     public static void main(String[] args) {
5         JFrame frame1 = new JFrame();
6         frame1.setTitle("Window 1");
7         frame1.setSize(200, 150);
8         frame1.setLocation(200, 100);
9         frame1.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
10        frame1.setVisible(true);
11
12        JFrame frame2 = new JFrame();
13        frame2.setTitle("Window 2");
14        frame2.setSize(200, 150);
15        frame2.setLocation(410, 100);
16        frame2.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17        frame2.setVisible(true);
18    }
19 }
```

This program creates two objects of the `JFrame` class (lines 5, 12) and then uses the methods `setTitle`, `setSize`, `setLocation`, `setDefaultCloseOperation`, and `setVisible` to set the properties of the objects. The `setTitle` method sets a title for the window (lines 6, 13). The `setSize` method sets the window's width and height (lines 7, 14). The `setLocation` method specifies the location of the window's upper-left corner (lines 8, 15). The `setDefaultCloseOperation` method terminates the program when the frame is closed (lines 9, 16). The `setVisible` method displays the window.

You can add graphical user interface components, such as buttons, labels, text fields, check boxes, and combo boxes to the window. The components are defined using classes. [Listing 8.6](#) gives an example of creating a graphical user interface, as shown in [Figure 8.1](#).



Video Note

Use classes

LISTING 8.6 GUIComponents.java

```
1 import javax.swing.*;
2
3 public class GUIComponents {
4     public static void main(String[] args) {
5         // Create a button with text OK
6         JButton jbtOK = new JButton("OK");
7
8         // Create a button with text Cancel
9         JButton jbtCancel = new JButton("Cancel");
10
11        // Create a label with text "Enter your name: "
12        JLabel jlblName = new JLabel("Enter your name: ");
13
14        // Create a text field with text "Type Name Here"
15        JTextField jtfName = new JTextField("Type Name Here");
16
17        // Create a check box with text bold
18        JCheckBox jchkBold = new JCheckBox("Bold");
19
20        // Create a check box with text italic
21        JCheckBox jchkItalic = new JCheckBox("Italic");
22
23        // Create a radio button with text red
24        JRadioButton jrbRed = new JRadioButton("Red");
25
26        // Create a radio button with text yellow
27        JRadioButton jrbYellow = new JRadioButton("Yellow");
28
29        // Create a combo box with several choices
30        JComboBox jcboColor = new JComboBox(new String[]{"Freshman",
31            "Sophomore", "Junior", "Senior"});
32
33        // Create a panel to group components
34        JPanel panel = new JPanel();
35        panel.add(jbtOK); // Add the OK button to the panel
36        panel.add(jbtCancel); // Add the Cancel button to the panel
37        panel.add(jlblName); // Add the label to the panel
38        panel.add(jtfName); // Add the text field to the panel
39        panel.add(jchkBold); // Add the check box to the panel
40        panel.add(jchkItalic); // Add the check box to the panel
41        panel.add(jrbRed); // Add the radio button to the panel
42        panel.add(jrbYellow); // Add the radio button to the panel
43        panel.add(jcboColor); // Add the combo box to the panel
44
45        JFrame frame = new JFrame(); // Create a frame
46        frame.add(panel); // Add the panel to the frame
47        frame.setTitle("Show GUI Components");
48        frame.setSize(450, 100);
49        frame.setLocation(200, 100);
50        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
51        frame.setVisible(true);
52    }
53 }
```

Video Note

Use classes

create a button

create a button

create a label

create a text field

create a check box

create a check box

create a radio button

create a radio button

create a combo box

create a panel
add to panel

create a frame
add panel to frame

display frame

This program creates GUI objects using the classes **JButton**, **JLabel**, **JTextField**, **JCheckBox**, **JRadioButton**, and **JComboBox** (lines 6–31). Then, using the **JPanel**

class (line 34), it then creates a panel object and adds to it the button, label, text field, check box, radio button, and combo box (lines 35–43). The program then creates a frame and adds the panel to the frame (line 45). The frame is displayed in line 51.

8.7 Static Variables, Constants, and Methods

The data field `radius` in the circle class in [Listing 8.1](#) is known as an *instance variable*. An instance variable is tied to a specific instance of the class; it is not shared among objects of the same class. For example, suppose that you create the following objects:

instance variable

```
Circle circle1 = new Circle();
Circle circle2 = new Circle(5);
```



Video Note

static vs. instance

The `radius` in `circle1` is independent of the `radius` in `circle2` and is stored in a different memory location. Changes made to `circle1`'s `radius` do not affect `circle2`'s `radius`, and vice versa.

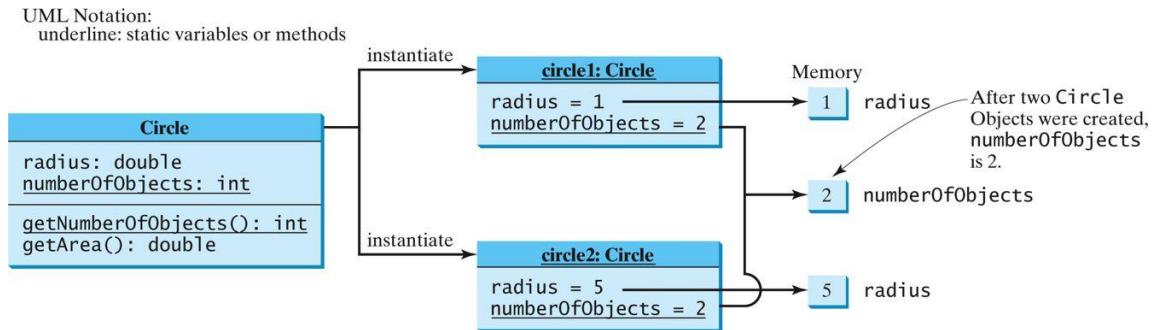
If you want all the instances of a class to share data, use *static variables*, also known as *class variables*. Static variables store values for the variables in a common memory location. Because of this common location, if one object changes the value of a static variable, all objects of the same class are affected. Java supports static methods as well as static variables. *Static methods* can be called without creating an instance of the class.

static variable

static method

Let us modify the `Circle` class by adding a static variable `numberOfObjects` to count the number of circle objects created. When the first object of this class is created, `numberOfObjects` is `1`. When the second object is created, `numberOfObjects` becomes `2`. The UML of the new circle class is shown in [Figure 8.12](#). The `Circle` class defines the instance variable `radius` and the static variable `numberOfObjects`, the instance methods `getRadius`, `setRadius`, and `getArea`, and the static method `getNumberOfObjects`. (Note that static variables and methods are underlined in the UML class diagram.)

FIGURE 8.12 Instance variables belong to the instances and have memory storage independent of one another. Static variables are shared by all the instances of the same class.



To declare a static variable or define a static method, put the modifier **static** in the variable or method declaration. The static variable **numberOfObjects** and the static method **getNumberOfObjects()** can be declared as follows:

declare static variable

```
static int numberOfObjects;
```

define static method

```
static int getNumberOfObjects() {
    return numberOfObjects;
}
```

Constants in a class are shared by all objects of the class. Thus, constants should be declared **final static**. For example, the constant **PI** in the **Math** class is defined as:

```
final static double PI = 3.14159265358979323846;
```

declare constant

The new circle class, named **Circle2**, is declared in [Listing 8.7](#):

LISTING 8.7 Circle2.java

```

1 public class Circle2 {
2     /** The radius of the circle */
3     double radius;
4
5     /** The number of objects created */
6     static int numberOfWorkObjects = 0;           static variable
7
8     /** Construct a circle with radius 1 */
9     Circle2() {
10         radius = 1.0;
11         numberOfWorkObjects++;                  increase by 1
12     }
13
14     /** Construct a circle with a specified radius */
15     Circle2(double newRadius) {
16         radius = newRadius;
17         numberOfWorkObjects++;                  increase by 1
18     }
19
20     /** Return numberOfWorkObjects */
21     static int getNumberOfWorkObjects() {        static method
22         return numberOfWorkObjects;
23     }
24
25     /** Return the area of this circle */
26     double getArea() {
27         return radius * radius * Math.PI;
28     }
29 }
```

Method `getNumberOfObjects()` in `Circle2` is a static method. Other examples of static methods are `showMessageDialog` and `showInputDialog` in the `JOptionPane` class and all the methods in the `Math` class. The `main` method is static, too.

Instance methods (e.g., `getArea()`) and instance data (e.g., `radius`) belong to instances and can be used only after the instances are created. They are accessed via a reference variable. Static methods (e.g., `getNumberOfObjects()`) and static data (e.g., `numberOfObjects`) can be accessed from a reference variable or from their class name.

The program in [Listing 8.8](#) demonstrates how to use instance and static variables and methods and illustrates the effects of using them.

LISTING 8.8 TestCircle2.java

```

1 public class TestCircle2 {
2     /** Main method */
3     public static void main(String[] args) {
4         System.out.println("Before creating objects");
5         System.out.println("The number of Circle objects is " +
6             Circle2.numberOfObjects);                                static variable
7
8         // Create c1
9         Circle2 c1 = new Circle2();
10
11        // Display c1 BEFORE c2 is created
12        System.out.println("\nAfter creating c1");
13        System.out.println("c1: radius (" + c1.radius +           instance variable
14            ") and number of Circle objects (" +
15                c1.numberOfObjects + ")");
16
17        // Create c2
18        Circle2 c2 = new Circle2(5);
19
20        // Modify c1
21        c1.radius = 9;
22
23        // Display c1 and c2 AFTER c2 was created
24        System.out.println("\nAfter creating c2 and modifying c1");
25        System.out.println("c1: radius (" + c1.radius +           static variable
26            ") and number of Circle objects (" +
27                c1.numberOfObjects + ")");
28        System.out.println("c2: radius (" + c2.radius +           static variable
29            ") and number of Circle objects (" +
30                c2.numberOfObjects + ")");
31    }
32 }
```



Before creating objects

The number of Circle objects is 0

After creating c1

c1: radius (1.0) and number of Circle objects (1)

After creating c2 and modifying c1

c1: radius (9.0) and number of Circle objects (2)

c2: radius (5.0) and number of Circle objects (2)

When you compile **TestCircle2.java**, the Java compiler automatically compiles **Circle2.java** if it has not been compiled since the last change.

Static variables and methods can be accessed without creating objects. Line 6 displays the number of objects, which is 0, since no objects have been created.

The `main` method creates two circles, `c1` and `c2` (lines 9, 18). The instance variable `radius` in `c1` is modified to become 9 (line 21). This change does not affect the instance variable `radius` in `c2`, since these two instance variables are independent. The static variable `numberOfObjects` becomes 1 after `c1` is created (line 9), and it becomes 2 after `c2` is created (line 18).

Note that `PI` is a constant defined in `Math`, and `Math.PI` references the constant. `c.numberOfObjects` could be replaced by `Circle2.numberOfObjects`. This improves readability, because the reader can easily recognize the static variable. You can also replace `Circle2.numberOfObjects` by `Circle2.getNumberOfObjects()`.



Tip

Use `ClassName.methodName(arguments)` to invoke a static method and `ClassName.-staticVariable` to access a static variable. This improves readability, because the user can easily recognize the static method and data in the class.

use class name

Static variables and methods can be used from instance or static methods in the class. However, instance variables and methods can be used only from instance methods, not from static methods, since static variables and methods don't belong to a particular object. Thus the code given below is wrong.

```
1 public class Foo {  
2     int i = 5;  
3     static int k = 2;  
4  
5     public static void main(String[] args) {  
6         int j = i; // Wrong because i is an instance variable  
7         m1(); // Wrong because m1() is an instance method
```

```

8    }
9
10   public void m1() {
11     // Correct since instance and static variables and methods
12     // can be used in an instance method
13     i = i + k + m2(i, k);
14   }
15
16   public static int m2(int i, int j) {
17     return (int)(Math.pow(i, j));
18   }
19 }
```

Note that if you replace the code in lines 5–8 with the following new code, the program is fine, because the instance data field `i` and method `m1` are now accessed from an object `foo` (lines 6–7):

```

1 public class Foo {
2   int i = 5;
3   static int k = 2;
4
5   public static void main(String[] args) {
6     Foo foo = new Foo();
7     int j = foo.i; // OK, foo.i accesses the object's instance variable
8     foo.m1(); // OK. Foo.m1() invokes object's instance method
9   }
10
11   public void m1() {
12     i = i + k + m2(i, k);
13   }
14
15   public static int m2(int i, int j) {
16     return (int)(Math.pow(i, j));
17   }
18 }
```



Design Guide

How do you decide whether a variable or method should be an instance one or a static one? A variable or method that is dependent on a specific instance of the class should be an instance variable or method. A variable or method that is not dependent on a specific instance of the class should be a static variable or method. For example, every circle has its own radius. Radius is dependent on a specific circle. Therefore, `radius` is an instance variable of the `Circle` class. Since the `getArea` method is dependent on a specific circle, it is an instance method. None of the methods in the `Math` class, such as `random`,

`pow`, `sin`, and `cos`, is dependent on a specific instance. Therefore, these methods are static methods. The `main` method is static and can be invoked directly from a class.

instance or static?



Caution

It is a common design error to define an instance method that should have been defined static. For example, the method `factorial(int n)` should be defined static, as shown below, because it is independent of any specific instance.

common design error

```
public class Test {  
    public int factorial(int n) {  
        int result = 1;  
        for (int i = 1; i <= n; i++)  
            result *= i;  
  
        return result;  
    }  
}
```

(a) Wrong design

```
public class Test {  
    public static int factorial(int n)  
    int result = 1;  
    for (int i = 1; i <= n; i++)  
        result *= i;  
  
    return result;  
}  
}
```

(b) Correct design

8.8 Visibility Modifiers

You can use the `public` visibility modifier for classes, methods, and data fields to denote that they can be accessed from any other classes. If no visibility modifier is used, then by default the classes, methods, and data fields are accessible by any class in the same package. This is known as *package-private* or *package-access*.



Note

Packages can be used to organize classes. To do so, you need to add the following line as the first noncomment and nonblank statement in the program:

```
package packageName;
```

If a class is defined without the package statement, it is said to be placed in the *default package*.

Java recommends that you place classes into packages rather than using a default package. For simplicity, however, this book uses default packages. For more information on packages, see Supplement III.G, “Packages.”

using packages

In addition to the **public** and default visibility modifiers, Java provides the **private** and **protected** modifiers for class members. This section introduces the **private** modifier. The **protected** modifier will be introduced in §11.13, “The **protected** Data and Methods.”

The **private** modifier makes methods and data fields accessible only from within its own class. [Figure 8.13](#) illustrates how a public, default, and private data field or method in class **C1** can be accessed from a class **C2** in the same package and from a class **C3** in a different package.

FIGURE 8.13 The private modifier restricts access to its defining class, the default modifier restricts access to a package, and the public modifier enables unrestricted access.

```
package p1;  
public class C1 {  
    public int x;  
    int y;  
    private int z;  
  
    public void m1() {  
    }  
    void m2() {  
    }  
    private void m3() {  
    }  
}
```

```
package p1;  
public class C2 {  
    void aMethod() {  
        C1 o = new C1();  
        can access o.x;  
        can access o.y;  
        cannot access o.z;  
  
        can invoke o.m1();  
        can invoke o.m2();  
        cannot invoke o.m3();  
    }  
}
```

```
package p2;  
public class C3 {  
    void aMethod() {  
        C1 o = new C1();  
        can access o.x;  
        cannot access o.y;  
        cannot access o.z;  
  
        can invoke o.m1();  
        cannot invoke o.m2();  
        cannot invoke o.m3();  
    }  
}
```

If a class is not defined **public**, it can be accessed only within the same package. As shown in [Figure 8.14](#), **C1** can be accessed from **C2** but not from **C3**.

FIGURE 8.14 A nonpublic class has package-access.

```
package p1;  
class C1 {  
    ...  
}
```

```
package p1;  
public class C2 {  
    can access C1  
}
```

```
package p2;  
public class C3 {  
    cannot access C1;  
    can access C2;  
}
```

A visibility modifier specifies how data fields and methods in a class can be accessed from outside the class. There is no restriction on accessing data fields and methods from inside the class. As shown in [Figure 8.15\(b\)](#), an object `foo` of the `Foo` class cannot access its private members, because `foo` is in the `Test` class. As shown in [Figure 8.15\(a\)](#), an object `foo` of the `Foo` class can access its private members, because `foo` is defined inside its own class.

inside access

FIGURE 8.15 An object can access its private members if it is defined in its own class.

```
public class Foo {  
    private boolean x;  
  
    public static void main(String[] args) {  
        Foo foo = new Foo();  
        System.out.println(foo.x);  
        System.out.println(foo.convert());  
    }  
  
    private int convert() {  
        return x ? 1 : -1;  
    }  
}
```

(a) This is OK because object `foo` is used inside the `Foo` class



```
public class Test {  
    public static void main(String[] args) {  
        Foo foo = new Foo();  
        System.out.println(foo.x);  
        System.out.println(foo.convert());  
    }  
}
```

(b) This is wrong because `x` and `convert` are private in `Foo`.



Caution

The `private` modifier applies only to the members of a class. The `public` modifier can apply to a class or members of a class. Using modifiers `public` and `private` on local variables would cause a compile error.



Note

In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, use a private constructor. For example, there is no reason to create an instance from the `Math` class, because all of its data fields and methods are static. To prevent the user from creating objects from the `Math` class, the constructor in `java.lang.Math` is defined as follows:

```
private Math() {  
}
```

private constructor

8.9 Data Field Encapsulation

The data fields `radius` and `numberOfObjects` in the `Circle2` class in [Listing 8.7](#) can be modified directly (e.g., `myCircle.radius = 5` or `Circle2.numberOfObjects = 10`). This is not a good practice—for two reasons:



Video Note

Data field encapsulation

- First, data may be tampered with. For example, `numberOfObjects` is to count the number of objects created, but it may be mistakenly set to an arbitrary value (e.g., `Circle2.numberOfObjects = 10`).
- Second, the class becomes difficult to maintain and vulnerable to bugs. Suppose you want to modify the `Circle2` class to ensure that the radius is nonnegative after other programs have already used the class. You have to change not only the `Circle2` class but also the programs that use it, because the clients may have modified the radius directly (e.g., `myCircle.radius = -5`).

To prevent direct modifications of data fields, you should declare the data fields private, using the `private` modifier. This is known as *data field encapsulation*.

data field encapsulation

A private data field cannot be accessed by an object from outside the class that defines the private field. But often a client needs to retrieve and modify a data field. To make a private private constructor data field accessible, provide a *get* method to return its value. To enable a private data field to be updated, provide a *set* method to set a new value.



Note

Colloquially, a `get` method is referred to as a *getter* (or *accessor*), and a `set` method is referred to as a *setter* (or *mutator*).

accessor

mutator

A **get** method has the following signature:

```
public returnType getPropertyname()
```

boolean accessor

If the **returnType** is **boolean**, the **get** method should be defined as follows by convention:

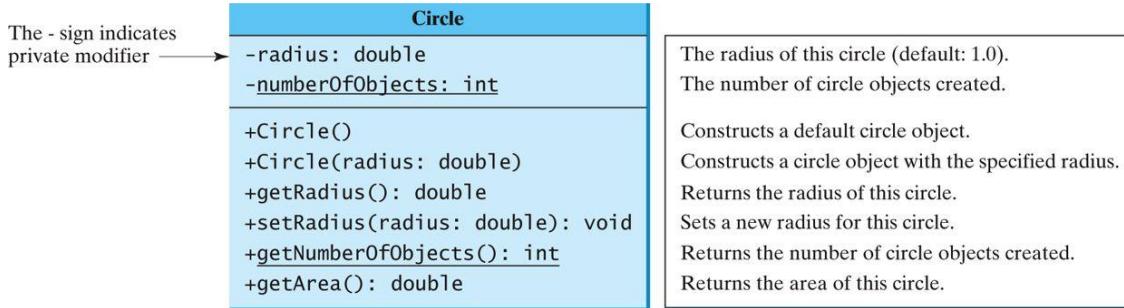
```
public boolean isPropertyName()
```

A **set** method has the following signature:

```
public void setPropertyName(dataType propertyName)
```

Let us create a new circle class with a private data-field radius and its associated accessor and mutator methods. The class diagram is shown in [Figure 8.16](#). The new circle class, named **Circle3**, is defined in [Listing 8.9](#):

FIGURE 8.16 The Circle class encapsulates circle properties and provides get/set and other methods.



LISTING 8.9 Circle3.java

```
1 public class Circle3 {  
2     /** The radius of the circle */  
3     private double radius = 1;  
4  
5     /** The number of the objects created */  
6     private static int numberofObjects = 0;  
7  
8     /** Construct a circle with radius 1 */  
9     public Circle3() {  
10         numberofObjects++;  
11     }  
12  
13    /** Construct a circle with a specified radius */  
14    public Circle3(double newRadius) {  
15        radius = newRadius;  
16        numberofObjects++;  
17    }  
18}
```

```

19  /** Return radius */
20  public double getRadius() {
21      return radius;
22  }
23
24  /** Set a new radius */
25  public void setRadius(double newRadius) {
26      radius = (newRadius >= 0) ? newRadius : 0;
27  }
28
29  /** Return numberOfObjects */
30  public static int getNumberOfObjects() {
31      return numberofObjects;
32  }
33
34  /** Return the area of this circle */
35  public double getArea() {
36      return radius * radius * Math.PI;
37  }
38 }

```

The `getRadius()` method (lines 20–22) returns the radius, and the `setRadius(newRadius)` method (line 25–27) sets a new radius into the object. If the new radius is negative, `0` is set to the radius in the object. Since these methods are the only ways to read and modify radius, you have total control over how the `radius` property is accessed. If you have to change the implementation of these methods, you need not change the client programs. This makes the class easy to maintain.

[Listing 8.10](#) gives a client program that uses the `Circle` class to create a `Circle` object and modifies the radius using the `setRadius` method.

LISTING 8.10 TestCircle3.java

```

1 public class TestCircle3 {
2     /** Main method */
3     public static void main(String[] args) {
4         // Create a Circle with radius 5.0
5         Circle3 myCircle = new Circle3(5.0);
6         System.out.println("The area of the circle of radius "
7             + myCircle.getRadius() + " is " + myCircle.getArea());           invoke public method
8
9         // Increase myCircle's radius by 10%
10        myCircle.setRadius(myCircle.getRadius() * 1.1);
11        System.out.println("The area of the circle of radius "
12            + myCircle.getRadius() + " is " + myCircle.getArea());           invoke public method
13
14        System.out.println("The number of objects created is "
15            + Circle3.getNumberOfObjects());                                invoke public method
16    }
17 }

```

The data field `radius` is declared private. Private data can be accessed only within their defining class. You cannot use `myCircle.radius` in the client program. A compile error would occur if you attempted to access private data from a client.

Since `numberOfObjects` is private, it cannot be modified. This prevents tampering. For example, the user cannot set `numberOfObjects` to `100`. The only way to make it `100` is to create `100` objects of the `Circle` class.

Suppose you combined `TestCircle` and `Circle` into one class by moving the `main` method in `TestCircle` into `Circle`. Could you use `myCircle.radius` in the `main` method? See Review Question 8.15 for the answer.



Design Guide

To prevent data from being tampered with and to make the class easy to maintain, declare data fields private.

8.10 Passing Objects to Methods

You can pass objects to methods. Like passing an array, passing an object is actually passing the reference of the object. The following code passes the `myCircle` object as an argument to the `printCircle` method:

```
1 public class Test {  
2     public static void main(String[] args) {  
3         // Circle3 is defined in Listing 8.9  
4         Circle3 myCircle = new Circle3(5.0);  
5         printCircle(myCircle);  
6     }  
7     public static void printCircle(Circle3 c) {  
8         System.out.println("The area of the circle of radius "  
9             + c.getRadius() + " is " + c.getArea());  
10    }  
11 }  
12 }
```

Java uses exactly one mode of passing arguments: pass-by-value. In the preceding code, the value of `myCircle` is passed to the `printCircle` method. This value is a reference to a `Circle` object.

pass-by-value

Let us demonstrate the difference between passing a primitive type value and passing a reference value with the program in [Listing 8.11](#):

LISTING 8.11 TestPassObject.java

```

1 public class TestPassObject {
2     /** Main method */
3     public static void main(String[] args) {
4         // Create a Circle object with radius 1
5         Circle3 myCircle = new Circle3(1);
6
7         // Print areas for radius 1, 2, 3, 4, and 5.
8         int n = 5;
9         printAreas(myCircle, n);
10
11        // See myCircle.radius and times
12        System.out.println("\n" + "Radius is " + myCircle.getRadius());
13        System.out.println("n is " + n);
14    }
15
16    /** Print a table of areas for radius */
17    public static void printAreas(Circle3 c, int times) {
18        System.out.println("Radius \t\tArea");
19        while (times >= 1) {
20            System.out.println(c.getRadius() + "\t\t" + c.getArea());
21            c.setRadius(c.getRadius() + 1);
22            times--;
23        }
24    }
25 }
```



Radius

1.0

2.0

3.0

4.0

5.0

Area

3.141592653589793

12.566370614359172

29.274333882308138

50.26548245743669

79.53981633974483

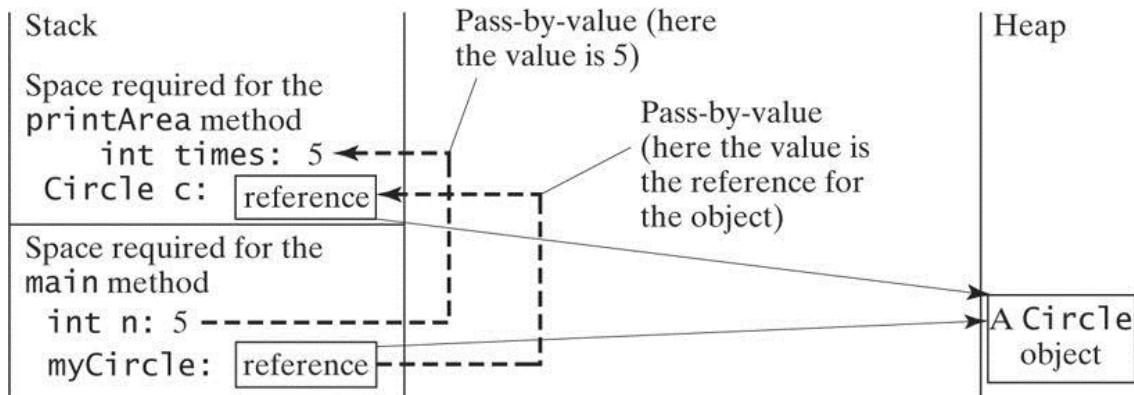
Radius is 6.0

n is 5

The `Circle3` class is defined in [Listing 8.9](#). The program passes a `Circle3` object `myCircle` and an integer value from `n` to invoke `printAreas (myCircle, n)` (line 9), which prints a table of areas for radii `1, 2, 3, 4, 5`, as shown in the sample output.

[Figure 8.17](#) shows the call stack for executing the methods in the program. Note that the objects are stored in a heap.

FIGURE 8.17 The value of `n` is passed to `times`, and the reference of `myCircle` is passed to `c` in the `printAreas` method.



When passing an argument of a primitive data type, the value of the argument is passed. In this case, the value of `n` (5) is passed to `times`. Inside the `printAreas` method, the content of `times` is changed; this does not affect the content of `n`.

When passing an argument of a reference type, the reference of the object is passed. In this case, `c` contains a reference for the object that is also referenced via `myCircle`. Therefore, changing the properties of the object through `c` inside the `printAreas` method has the same effect as doing so outside the method through the variable `myCircle`. Pass-by-value on references can be best described semantically as *pass-by-sharing*; i.e., the object referenced in the method is the same as the object being passed.

pass-by-sharing

8.11 Array of Objects

In Chapter 6, “Single-Dimensional Arrays,” arrays of primitive type elements were created. You can also create arrays of objects. For example, the following statement declares and creates an array of ten `Circle` objects:

```
Circle[] circleArray = new Circle[ 10 ];
```

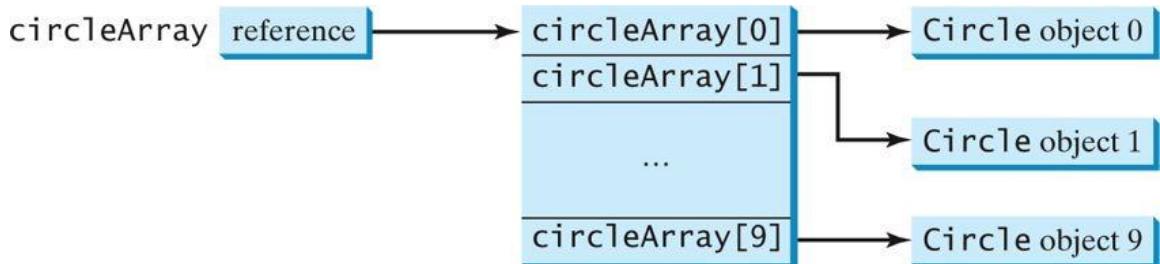
To initialize the `circleArray`, you can use a `for` loop like this one:

```
for (int i = 0; i < circleArray.length; i++) {
    circleArray[i] = new Circle(); }
```

An array of objects is actually an *array of reference variables*. So, invoking `circleArray[1].getArea()` involves two levels of referencing, as shown in [Figure 8.17](#).

[8.18.](#) `circleArray` references the entire array. `circleArray[1]` references a `Circle` object.

FIGURE 8.18 In an array of objects, an element of the array contains a reference to an object.



Note

When an array of objects is created using the `new` operator, each element in the array is a reference variable with a default value of `null`.

[Listing 8.12](#) gives an example that demonstrates how to use an array of objects. The program summarizes the areas of an array of circles. The program creates `circleArray`, an array composed of five `Circle` objects; it then initializes circle radii with random values and displays the total area of the circles in the array.

LISTING 8.12 TotalArea.java

```

1 public class TotalArea {
2     /** Main method */
3     public static void main(String[] args) {
4         // Declare circleArray
5         Circle3[] circleArray;
6
7         // Create circleArray
8         circleArray = createCircleArray();
9
10        // Print circleArray and total areas of the circles
11        printCircleArray(circleArray);
12    }
13
14    /** Create an array of Circle objects */
15    public static Circle3[] createCircleArray() {
16        Circle3[] circleArray = new Circle3[5];
17
18        for (int i = 0; i < circleArray.length; i++) {
19            circleArray[i] = new Circle3(Math.random() * 100);
20        }
21
22        // Return Circle array
23        return circleArray;
24    }
25
26    /** Print an array of circles and their total area */
27    public static void printCircleArray(Circle3[] circleArray) {
28        System.out.printf("%-30s%-15s\n", "Radius", "Area");
29        for (int i = 0; i < circleArray.length; i++) {
30            System.out.printf("%-30f%-15f\n", circleArray[i].getRadius(),
31                circleArray[i].getArea());
32        }
33
34        System.out.println("-----");
35
36        // Compute and display the result
37        System.out.printf("%-30s%-15f\n", "The total area of circles is",
38            sum(circleArray));
39    }
40
41    /** Add circle areas */
42    public static double sum(Circle3[] circleArray) {                                pass array of objects
43        // Initialize sum
44        double sum = 0;
45
46        // Add areas to sum
47        for (int i = 0; i < circleArray.length; i++)
48            sum += circleArray[i].getArea();
49
50        return sum;
51    }
52 }

```



Radius

70.577708

44.152266

Area

15648.941866

6124.291736

24.867853	1942.792644
5.680718	101.380949
36.734246	4239.280350

The total area of circles is	28056.687544

The program invokes `createCircleArray()` (line 8) to create an array of five `Circle` objects. Several `Circle` classes were introduced in this chapter. This example uses the `Circle` class introduced in §8.9, “Data Field Encapsulation.”

The circle radii are randomly generated using the `Math.random()` method (line 19). The `createCircleArray` method returns an array of `Circle` objects (line 23). The array is passed to the `printCircleArray` method, which displays the radius and area of each circle and the total area of the circles.

The sum of the circle areas is computed using the `sum` method (line 38), which takes the array of `Circle` objects as the argument and returns a `double` value for the total area.

KEY TERMS

accessor method (getter) [284](#)

action [264](#)

attribute [264](#)

behavior [264](#)

class [265](#)

client [267](#)

constructor [268](#)

data field [268](#)

data-field encapsulation [283](#)

default constructor [270](#)

dot operator (.) [271](#)

instance [271](#)

instance method [271](#)

instance variable [271](#)

instantiation [264](#)

mutator method (setter) [285](#)

null [272](#)

no-arg constructor [266](#)

object-oriented programming (OOP) [264](#)

Unified Modeling Language (UML) [265](#)

package-private (or package-access) [282](#)

private [283](#)

property [264](#)

public [282](#)

reference variable [271](#)

reference type [271](#)

state [264](#)

static method [278](#)

static variable [278](#)

CHAPTER SUMMARY

1. A class is a template for objects. It defines the properties of objects and provides constructors for creating objects and methods for manipulating them.

2. A class is also a data type. You can use it to declare object reference variables. An object reference variable that appears to hold an object actually contains a reference to that

object. Strictly speaking, an object reference variable and an object are different, but most of the time the distinction can be ignored.

3. An object is an instance of a class. You use the `new` operator to create an object, and the dot (`.`) operator to access members of that object through its reference variable.

4. An instance variable or method belongs to an instance of a class. Its use is associated with individual instances. A static variable is a variable shared by all instances of the same class. A static method is a method that can be invoked without using instances.

5. Every instance of a class can access the class's static variables and methods. For clarity, however, it is better to invoke static variables and methods using `ClassName.variable` and `ClassName.method`.

6. Modifiers specify how the class, method, and data are accessed. A `public` class, method, or data is accessible to all clients. A `private` method or data is accessible only inside the class.

7. You can provide a `get` method or a `set` method to enable clients to see or modify the data. Colloquially, a `get` method is referred to as a *getter* (or *accessor*), and a `set` method as a *setter* (or *mutator*).

8. A `get` method has the signature `public returnType
getPropertyName()`. If the `returnType` is `boolean`, the `get` method should be defined as `public boolean isPropertyName()`. A `set` method has the signature `public void setPropertyName(dataType propertyName)`.

9. All parameters are passed to methods using pass-by-value. For a parameter of a primitive type, the actual value is passed; for a parameter of a reference type, the reference for the object is passed.

10. A Java array is an object that can contain primitive type values or object type values. When an array of objects is created, its elements are assigned the default value of `null`.

REVIEW QUESTIONS

Sections 8.2–8.5

8.1 Describe the relationship between an object and its defining class. How do you define a class? How do you declare an object reference variable? How do you create an object? How do you declare and create an object in one statement?

8.2 What are the differences between constructors and methods?

8.3 Is an array an object or a primitive type value? Can an array contain elements of an object type as well as a primitive type? Describe the default value for the elements of an array.

8.4 What is wrong with the following program?

```
1 public class ShowErrors {  
2     public static void main(String[] args) {  
3         ShowErrors t = new ShowErrors(5);  
4     }  
5 }
```

(a)

```
1 public class ShowErrors {  
2     public static void main(String[] args) {  
3         ShowErrors t = new ShowErrors();  
4         t.x();  
5     }  
6 }
```

(b)

```
1 public class ShowErrors {  
2     public void method1() {  
3         Circle c;  
4         System.out.println("What is radius "  
5             + c.getRadius());  
6         c = new Circle();  
7     }  
8 }
```

(c)

```
1 public class ShowErrors {  
2     public static void main(String[] args) {  
3         C c = new C(5.0);  
4         System.out.println(c.value);  
5     }  
6 }  
7  
8 class C {  
9     int value = 2;  
10 }
```

(d)

8.5 What is wrong in the following code?

```
1 class Test {  
2     public static void main(String[] args) {  
3         A a = new A();  
4         a.print();  
5     }  
6 }  
7  
8 class A {  
9     String s;  
10  
11    A(String s) {  
12        this.s = s;  
13    }  
14  
15    public void print() {  
16        System.out.print(s);  
17    }  
18 }
```

8.6 What is the printout of the following code?

```
public class Foo {  
    private boolean x;  
    public static void main(String[] args) {  
        Foo foo = new Foo();  
        System.out.println(foo.x);  
    }  
}
```

Section 8.6

8.7 How do you create a **Date** for the current time? How do you display the current time?

8.8 How do you create a **JFrame**, set a title in a frame, and display a frame?

8.9 Which packages contain the classes **Date**, **JFrame**, **JOptionPane**, **System**, and **Math**?

Section 8.7

8.10 Suppose that the class **Foo** is defined in (a). Let **f** be an instance of **Foo**. Which of the statements in (b) are correct?

```
public class Foo {  
    int i;  
    static String s;  
    void imethod() {  
    }  
    static void smethod() {  
    }  
}
```

(a)

```
System.out.println(f.i);  
System.out.println(f.s);  
f.imethod();  
f.smethod();  
System.out.println(Foo.i);  
System.out.println(Foo.s);  
Foo.imethod();  
Foo.smethod();
```

(b)

8.11 Add the **static** keyword in the place of **?** if appropriate.

```

public class Test {
    private int count;

    public ? void main(String[] args) {
        ...
    }

    public ? int getCount() {
        return count;
    }

    public ? int factorial(int n) {
        int result = 1;
        for (int i = 1; i <= n; i++)
            result *= i;

        return result;
    }
}

```

8.12 Can you invoke an instance method or reference an instance variable from a static method? Can you invoke a static method or reference a static variable from an instance method? What is wrong in the following code?

```

1 public class Foo {
2     public static void main(String[] args) {
3         method1();
4     }
5
6     public void method1() {
7         method2();
8     }
9
10    public static void method2() {
11        System.out.println("What is radius " + c.get
Radius());
12    }

```

```
13  
14     Circle c = new      Circle();  
15 }
```

Sections 8.8–8.9

8.13 What is an accessor method? What is a mutator method? What are the naming conventions for accessor methods and mutator methods?

8.14 What are the benefits of data-field encapsulation?

8.15 In the following code, `radius` is private in the `Circle` class, and `myCircle` is an object of the `Circle` class. Does the highlighted code below cause any problems? Explain why.

```
public class Circle {  
    private double radius = 1.0;  
  
    /** Find the area of this circle */  
    public double getArea() {  
        return radius * radius * Math.PI;  
    }  
  
    public static void main(String[] args) {  
        Circle myCircle = new Circle();  
        System.out.println("Radius is " + myCircle.radius);  
    }  
}
```

Section 8.10

8.16 Describe the difference between passing a parameter of a primitive type and passing a parameter of a reference type. Show the output of the following program:

```

public class Test {
    public static void main(String[] args) {
        Count myCount = new Count();
        int times = 0;
        for (int i = 0; i < 100; i++)
            increment(myCount, times);
        System.out.println("count is " + myCount.count);
        System.out.println("times is " + times);
    }
    public static void increment(Count c, int times) {
        c.count++;
        times++;
    }
}

```

```

public class Count {
    public int count;
    public Count(int c) {
        count = c;
    }
    public Count() {
        count = 1;
    }
}

```

8.17 Show the output of the following program:

```

public class Test {
    public static void main(String[] args) {
        Circle circle1 =new Circle(1);
        Circle circle2 = new Circle(2);
        swap1(circle1, circle2);
        System.out.println("After swap1: circle1 = " +
                           circle1.radius + " circle2 = " +
                           circle2.radius);
        swap2(circle1, circle2);
        System.out.println("After swap2: circle1 = " +
                           circle1.radius + " circle2 = " +
                           circle2.radius);
    }
    public static void swap1(Circle x, Circle y) {
        Circle temp = x;
        x = y;
        y = temp;
    }
    public static void swap2(Circle x, Circle y) {
        double temp = x.radius;
        x.radius = y.radius;
        y.radius = temp;
    }
}
class Circle {
    double radius;
    Circle(double newRadius) {
        radius = newRadius;
    }
}

```

8.18 Show the printout of the following code:

```

public class Test {
    public static void main(String[] args) {
        int[] a = {1, 2};
        swap(a[0], a[1]);
        System.out.println("a[0] = " + a[0]
            + " a[1] = " + a[1]);
    }

    public static void swap(int n1, int n2) {
        int temp = n1;
        n1 = n2;
        n2 = temp;
    }
}

```

(a)

```

public class Test {
    public static void main(String[] args) {
        int[] a = {1, 2};
        swap(a);
        System.out.println("a[0] = " + a[0]
            + " a[1] = " + a[1]);
    }

    public static void swap(int[] a) {
        int temp = a[0];
        a[0] = a[1];
        a[1] = temp;
    }
}

```

(b)

```

public class Test {
    public static void main(String[] args) {
        T t = new T();
        swap(t);
        System.out.println("e1 = " + t.e1
            + " e2 = " + t.e2);
    }

    public static void swap(T t) {
        int temp = t.e1;
        t.e1 = t.e2;
        t.e2 = temp;
    }

    class T {
        int e1 = 1;
        int e2 = 2;
    }
}

```

(c)

```

public class Test {
    public static void main(String[] args) {
        T t1 = new T();
        T t2 = new T();
        System.out.println("t1's i = " +
            t1.i + " and j = " + t1.j);
        System.out.println("t2's i = " +
            t2.i + " and j = " + t2.j);
    }

    class T {
        static int i = 0;
        int j = 0;

        T() {
            i++;
            j = 1;
        }
    }
}

```

(d)

8.19 What is the output of the following program?

```

import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = null;
        m1(date);
        System.out.println(date);
    }

    public static void m1(Date date) {
        date = new Date();
    }
}

```

(a)

```

import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date(1234567);
        m1(date);
        System.out.println(date.getTime());
    }

    public static void m1(Date date) {
        date = new Date(7654321);
    }
}

```

(b)

```

import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date(1234567);
        m1(date);
        System.out.println(date.getTime());
    }

    public static void m1(Date date) {
        date.setTime(7654321);
    }
}

```

(c)

```

import java.util.Date;

public class Test {
    public static void main(String[] args) {
        Date date = new Date(1234567);
        m1(date);
        System.out.println(date.getTime());
    }

    public static void m1(Date date) {
        date = null;
    }
}

```

(d)

Section 8.11

8.20 What is wrong in the following code?

```

1 public class Test {
2     public static void main(String[] args) {
3         java.util.Date[] dates = new java.util.Date[10];
4         System.out.println(dates[0]);
5         System.out.println(dates[0].toString());
6     }
7 }

```

PROGRAMMING EXERCISES



Pedagogical Note

The exercises in [Chapters 8–14](#) achieve three objectives:

- Design classes and draw UML class diagrams;

- Implement classes from the UML;
- Use classes to develop applications.

Solutions for the UML diagrams for the even-numbered exercises can be downloaded from the Student Website and all others can be downloaded from the Instructor Website.

three objectives

Sections 8.2–8.5

8.1 (*The `Rectangle` class*) Following the example of the `Circle` class in §8.2, design a class named `Rectangle` to represent a rectangle. The class contains:

- Two `double` data fields named `width` and `height` that specify the width and height of the rectangle. The default values are `1` for both `width` and `height`.
- A no-arg constructor that creates a default rectangle.
- A constructor that creates a rectangle with the specified `width` and `height`.
- A method named `getArea()` that returns the area of this rectangle.
- A method named `getPerimeter()` that returns the perimeter.

Draw the UML diagram for the class. Implement the class. Write a test program that creates two `Rectangle` objects—one with width `4` and height `40` and the other with width `3.5` and height `35.9`. Display the width, height, area, and perimeter of each rectangle in this order.

8.2 (*The `Stock` class*) Following the example of the `Circle` class in §8.2, design a class named `Stock` that contains:

- A string data field named `symbol` for the stock's symbol.
- A string data field named `name` for the stock's name.
- A `double` data field named `previousClosingPrice` that stores the stock price for the previous day.
- A `double` data field named `currentPrice` that stores the stock price for the current time.

- A constructor that creates a stock with specified symbol and name.
- A method named `getChangePercent()` that returns the percentage changed from `previousClosingPrice` to `currentPrice`.

Draw the UML diagram for the class. Implement the class. Write a test program that creates a `Stock` object with the stock symbol JAVA, the name Sun Microsystems Inc, and the previous closing price of `4.5`. Set a new current price to `4.35` and display the price-change percentage.

Section 8.6

8.3* (*Using the Date class*) Write a program that creates a `Date` object, sets its elapsed time to `10000, 100000, 10000000, 100000000, 1000000000, 10000000000, 100000000000, 1000000000000`, and displays the date and time using the `toString()` method, respectively.

8.4* (*Using the Random class*) Write a program that creates a `Random` object with seed `1000` and displays the first 50 random integers between `0` and `100` using the `nextInt(100)` method.

8.5* (*Using the GregorianCalendar class*) Java API has the `GregorianCalendar` class in the `java.util` package that can be used to obtain the year, month, and day of a date. The no-arg constructor constructs an instance for the current date, and the methods `get(GregorianCalendar.YEAR)`, `get(GregorianCalendar.MONTH)`, and `get(GregorianCalendar.DAY_OF_MONTH)` return the year, month, and day. Write a program to perform two tasks:

- Display the current year, month, and day.
- The `GregorianCalendar` class has the `setTimeInMillis(long)`, which can be used to set a specified elapsed time since January 1, 1970. Set the value to `1234567898765L` and display the year, month, and day.

Sections 8.7–8.9

8.6** (*Displaying calendars*) Rewrite the `PrintCalendar` class in [Listing 5.12](#) to display calendars in a message dialog box. Since the output is generated from several static methods in the class, you may define a static `String` variable `output` for storing the output and display it in a message dialog box.

8.7 (*The Account class*) Design a class named **Account** that contains:

- A private **int** data field named **id** for the account (default **0**).
- A private **double** data field named **balance** for the account (default **0**).
- A private **double** data field named **annualInterestRate** that stores the current interest rate (default **0**). Assume all accounts have the same interest rate.
- A private **Date** data field named **dateCreated** that stores the date when the account was created.
- A no-arg constructor that creates a default account.
- A constructor that creates an account with the specified id and initial balance.
- The accessor and mutator methods for **id**, **balance**, and **annualInterestRate**.
- The accessor method for **dateCreated**.
- A method named **getMonthlyInterestRate()** that returns the monthly interest rate.
- A method named **withdraw** that withdraws a specified amount from the account.
- A method named **deposit** that deposits a specified amount to the account.

Draw the UML diagram for the class. Implement the class. Write a test program that creates an **Account** object with an account ID of 1122, a balance of \$20,000, and an annual interest rate of 4.5%. Use the **withdraw** method to withdraw \$2,500, use the **deposit** method to deposit \$3,000, and print the balance, the monthly interest, and the date when this account was created.

8.8 (*The Fan class*) Design a class named **Fan** to represent a fan. The class contains:



Video Note

The Fan class

- Three constants named **SLOW**, **MEDIUM**, and **FAST** with values **1**, **2**, and **3** to denote the fan speed.
- A private **int** data field named **speed** that specifies the speed of the fan (default **SLOW**).
- A private **boolean** data field named **on** that specifies whether the fan is on (default **false**).
- A private **double** data field named **radius** that specifies the radius of the fan (default **5**).
- A string data field named **color** that specifies the color of the fan (default **blue**).
- The accessor and mutator methods for all four data fields.
- A no-arg constructor that creates a default fan.
- A method named **toString()** that returns a string description for the fan. If the fan is on, the method returns the fan speed, color, and radius in one combined string. If the fan is not on, the method returns fan color and radius along with the string “fan is off” in one combined string.

Draw the UML diagram for the class. Implement the class. Write a test program that creates two **Fan** objects. Assign maximum speed, radius **10**, color **yellow**, and turn it on to the first object. Assign medium speed, radius **5**, color **blue**, and turn it off to the second object. Display the objects by invoking their **toString** method.

8.9** (*Geometry: n-sided regular polygon*) In an *n*-sided regular polygon all sides have the same length and all angles have the same degree (i.e., the polygon is both equilateral and equiangular). Design a class named **RegularPolygon** that contains:

- A private **int** data field named **n** that defines the number of sides in the polygon with default value **3**.
- A private **double** data field named **side** that stores the length of the side with default value **1**.
- A private **double** data field named **x** that defines the *x*-coordinate of the center of the polygon with default value **0**.

- A private **double** data field named **y** that defines the *y*-coordinate of the center of the polygon with default value **0**.
- A no-arg constructor that creates a regular polygon with default values.
- A constructor that creates a regular polygon with the specified number of sides and length of side, centered at (**0**, **0**).
- A constructor that creates a regular polygon with the specified number of sides, length of side, and *x*-and *y*-coordinates.
- The accessor and mutator methods for all data fields.
- The method **getPerimeter()** that returns the perimeter of the polygon.
- The method **getArea()** that returns the area of the polygon. The formula for computing the area of a regular polygon is

$$Area = \frac{n \times s^2}{4 \times \tan\left(\frac{\pi}{n}\right)}.$$

Draw the UML diagram for the class. Implement the class. Write a test program that creates three **RegularPolygon** objects, created using the no-arg constructor, using **RegularPolygon(6, 4)**, and using **RegularPolygon(10, 4, 5.6, 7.8)**. For each object, display its perimeter and area.

8.10* (*Algebra: quadratic equations*) Design a class named **QuadraticEquation** for a quadratic equation $ax^2 + bx + c = 0$. The class contains:

- Private data fields **a**, **b**, and **c** that represents three coefficients.
- A constructor for the arguments for **a**, **b**, and **c**.
- Three **get** methods for **a**, **b**, and **c**.
- A method named **getDiscriminant()** that returns the discriminant, which is $b^2 - 4ac$.
- The methods named **getRoot1()** and **getRoot2()** for returning two roots of the equation

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

These methods are useful only if the discriminant is nonnegative. Let these methods return `0` if the discriminant is negative.

Draw the UML diagram for the class. Implement the class. Write a test program that prompts the user to enter values for a , b , and c and displays the result based on the discriminant. If the discriminant is positive, display the two roots. If the discriminant is `0`, display the one root. Otherwise, display “The equation has no roots.” See Exercise 3.1 for sample runs.

8.11* (*Algebra: linear equations*) Design a class named `LinearEquation` for a system of linear equations:

$$\begin{aligned} ax + by &= e & x &= \frac{ed - bf}{ad - bc} & y &= \frac{af - ec}{ad - bc} \\ cx + dy &= f \end{aligned}$$

The class contains:

- Private data fields `a`, `b`, `c`, `d`, `e`, and `f`.
- A constructor with the arguments for `a`, `b`, `c`, `d`, `e`, and `f`.
- Six `get` methods for `a`, `b`, `c`, `d`, `e`, and `f`.
- A method named `isSolvable()` that returns true if is not `0`.
- Methods `getX()` and `getY()` that return the solution for the equation.

Draw the UML diagram for the class. Implement the class. Write a test program that prompts the user to enter `a`, `b`, `c`, `d`, `e`, and `f` and displays the result. If is `0`, report that “The equation has no solution.” See Exercise 3.3 for sample runs.

8.12** (*Geometry: intersection*) Suppose two line segments intersect. The two endpoints for the first line segment are (x_1, y_1) and (x_2, y_2) and for the second line segment are (x_3, y_3) and (x_4, y_5) . Write a program that prompts the user to enter these four endpoints and displays the intersecting point.

(Hint: Use the `LinearEquation` class from the preceding exercise.)



Enter the endpoints of the first line segment: 2.0 2. 0

0 0

↙ Enter

Enter the endpoints of the second line segment: 0 2.0

2.0 0

↙ Enter

The intersecting point is: (1.0, 1.0)

8.13 (The Location class)** Design a class named **Location** for locating a maximal value and its location in a two-dimensional array. The class contains public data fields **row**, **column**, and **maxValue** that store the maximal value and its indices in a two dimensional array with **row** and **column** as **int** type and **maxValue** as **double** type.

Write the following method that returns the location of the largest element in a two-dimensional array.

```
public static Location locateLargest(double[][] a)
```

The return value is an instance of **Location**. Write a test program that prompts the user to enter a two-dimensional array and displays the location of the largest element in the array. Here is a sample run:



Enter the number of rows and columns of the array: Enter

the array: 3 4

↙ Enter

Enter the array:

23.5 35 2 10

↙ Enter

4.5 3 45 3.5

↙ Enter

35 44 5.5 9.6  Enter

The location of the largest element is 45 at (1, 2)

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 263).
<vbk:9781256335153#outline(10)>

CHAPTER 9 STRINGS AND TEXT I/O

Objectives

- To use the `String` class to process fixed strings ([§9.2](#)).
- To use the `Character` class to process a single character ([§9.3](#)).
- To use the `StringBuilder/StringBuffer` class to process flexible strings ([§9.4](#)).
- To distinguish among the `String`, `StringBuilder`, and `StringBuffer` classes ([§9.2–9.4](#)).
- To learn how to pass arguments to the `main` method from the command line ([§9.5](#)).
- To discover file properties and to delete and rename files using the `File` class ([§9.6](#)).
- To write data to a file using the `PrintWriter` class ([§9.7.1](#)).
- To read data from a file using the `Scanner` class ([§9.7.2](#)).
- (GUI) To open files using a dialog box ([§9.8](#)).

9.1 Introduction

Often you encounter problems that involve string processing and file input and output. Suppose you need to write a program that replaces all occurrences of a word in a file with a new word. How do you accomplish this? This chapter introduces strings and text files, which will enable you to solve problems of this type. (Since no new concepts are introduced here, instructors may assign this chapter for students to study on their own.)

problem

9.2 The `String` Class

A *string* is a sequence of characters. In many languages, strings are treated as an array of characters, but in Java a string is an object. The `String` class has 11 constructors and more than 40 methods for manipulating strings. Not only is it very useful in programming, but also it is a good example for learning classes and objects.

9.2.1 Constructing a String

You can create a string object from a string literal or from an array of characters. To create a string from a string literal, use a syntax like this one:

```
String newString = new String(stringLiteral);
```

The argument `stringLiteral` is a sequence of characters enclosed inside double quotes. The following statement creates a `String` object `message` for the string literal `"Welcome to Java"`:

```
String message = new String("Welcome to Java");
```

Java treats a string literal as a `String` object. So, the following statement is valid:

```
String message = "Welcome to Java";
```

string literal object

You can also create a string from an array of characters. For example, the following statements create the string “Good Day”:

```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};  
String message = new String(charArray);
```



Note

A `String` variable holds a reference to a `String` object that stores a string value. Strictly speaking, the terms `String variable`, `String object`, and `string value` are different, but most of the time the distinctions between them can be ignored. For simplicity, the term `string` will often be used to refer to `String` variable, `String` object, and `string value`.

string variable, string object, string value

9.2.2 Immutable Strings and Interned Strings

A `String` object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

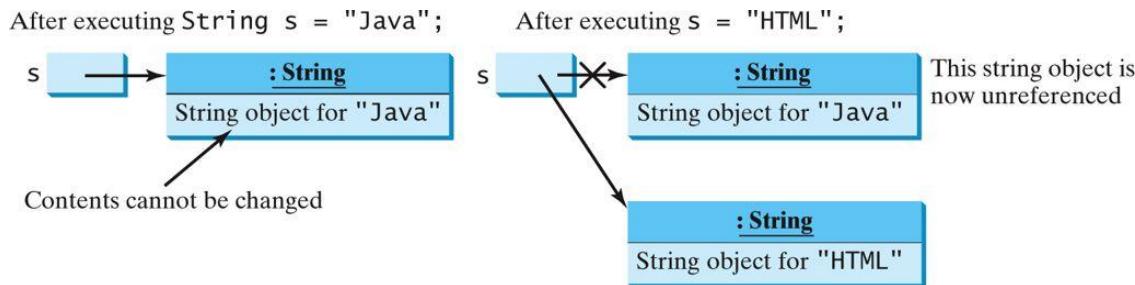
immutable

```
String s = "Java";  
s = "HTML";
```

The answer is no. The first statement creates a `String` object with the content “Java” and assigns its reference to `s`. The second statement creates a new `String` object with the

content “HTML” and assigns its reference to `s`. The first `String` object still exists after the assignment, but it can no longer be accessed, because variable `s` now points to the new object, as shown in [Figure 9.1](#).

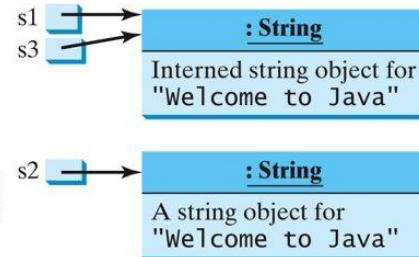
FIGURE 9.1 Strings are immutable; once created, their contents cannot be changed.



Since strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called *interned*. For example, the following statements:

interned string

```
String s1 = "Welcome to Java";
String s2 = new String("Welcome to Java");
String s3 = "Welcome to Java";
System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```



display

```
s1 == s2 is false
s1 == s3 is true
```

In the preceding statements, `s1` and `s3` refer to the same interned string “Welcome to Java”, therefore `s1 == s3` is `true`. However, `s1 == s2` is `false`, because `s1` and `s2` are two different string objects, even though they have the same contents.

9.2.3 String Comparisons

The `String` class provides the methods for comparing strings, as shown in [Figure 9.2](#).

FIGURE 9.2 The String class contains the methods for comparing strings.

java.lang.String	
+equals(s1: String): boolean	Returns true if this string is equal to string s1.
+equalsIgnoreCase(s1: String): boolean	Returns true if this string is equal to string s1 case insensitive.
+compareTo(s1: String): int	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than s1.
+compareToIgnoreCase(s1: String): int	Same as compareTo except that the comparison is case insensitive.
+regionMatches(index: int, s1: String, s1Index: int, len: int): boolean	Returns true if the specified subregion of this string exactly matches the specified subregion in string s1.
+regionMatches(ignoreCase: boolean, index: int, s1: String, s1Index: int, len: int): boolean	Same as the preceding method except that you can specify whether the match is case sensitive.
+startsWith(prefix: String): boolean	Returns true if this string starts with the specified prefix.
+endsWith(suffix: String): boolean	Returns true if this string ends with the specified suffix.

How do you compare the contents of two strings? You might attempt to use the `==` operator, as follows:

```
if (string1 == string2)
    System.out.println("string1 and string2 are the same
object");
else
    System.out.println("string1 and string2 are different
objects");
```

`==`

However, the `==` operator checks only whether `string1` and `string2` refer to the same object; it does not tell you whether they have the same contents. Therefore, you cannot use the `==` operator to find out whether two string variables have the same contents. Instead, you should use the `equals` method. The code given below, for instance, can be used to compare two strings:

```
if (string1.equals(string2))
    System.out.println("string1 and string2 have the same
contents");
else
    System.out.println("string1 and string2 are not
equal");

string1.equals(string2)
```

For example, the following statements display `true` and then `false`.

```
String s1 = new String("Welcome to Java");
```

```
String s2 = "Welcome to Java";
String s3 = "Welcome to C++";
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // false
```

The `compareTo` method can also be used to compare two strings. For example, consider the following code:

```
s1.compareTo(s2)

s1.compareTo(s2)
```

The method returns the value `0` if `s1` is equal to `s2`, a value less than `0` if `s1` is lexicographically (i.e., in terms of Unicode ordering) less than `s2`, and a value greater than `0` if `s1` is lexicographically greater than `s2`.

The actual value returned from the `compareTo` method depends on the offset of the first two distinct characters in `s1` and `s2` from left to right. For example, suppose `s1` is "abc" and `s2` is "abg", and `s1.compareTo(s2)` returns `-4`. The first two characters (`a` vs. `a`) from `s1` and `s2` are compared. Because they are equal, the second two characters (`b` vs. `b`) are compared. Because they are also equal, the third two characters (`c` vs. `g`) are compared. Since the character `c` is `4` less than `g`, the comparison returns `-4`.



Caution

Syntax errors will occur if you compare strings by using comparison operators, such as `>`, `>=`, `<`, or `<=`. Instead, you have to use `s1.compareTo(s2)`.



Note

The `equals` method returns `true` if two strings are equal and `false` if they are not.

The `compareTo` method returns `0`, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.

The `String` class also provides `equalsIgnoreCase`, `compareToIgnoreCase`, and `regionMatches` methods for comparing strings. The `equalsIgnoreCase` and `compareToIgnoreCase` methods ignore the case of the letters when comparing two strings. The `regionMatches` method compares portions of two strings for equality. You can also use `str.startsWith(prefix)` to check whether string `str` starts with a

specified prefix, and `str.endsWith(suffix)` to check whether string `str` ends with a specified `suffix`.

9.2.4 String Length, Characters, and Combining Strings

The `String` class provides the methods for obtaining length, retrieving individual characters, and concatenating strings, as shown in [Figure 9.3](#).

FIGURE 9.3 The String class contains the methods for getting string length, individual characters, and combining strings.

java.lang.String	
<code>+length(): int</code>	Returns the number of characters in this string.
<code>+charAt(index: int): char</code>	Returns the character at the specified index from this string.
<code>+concat(s1: String): String</code>	Returns a new string that concatenates this string with string <code>s1</code> .

You can get the length of a string by invoking its `length()` method. For example, `message.length()` returns the length of the string `message`.

`length()`



Caution

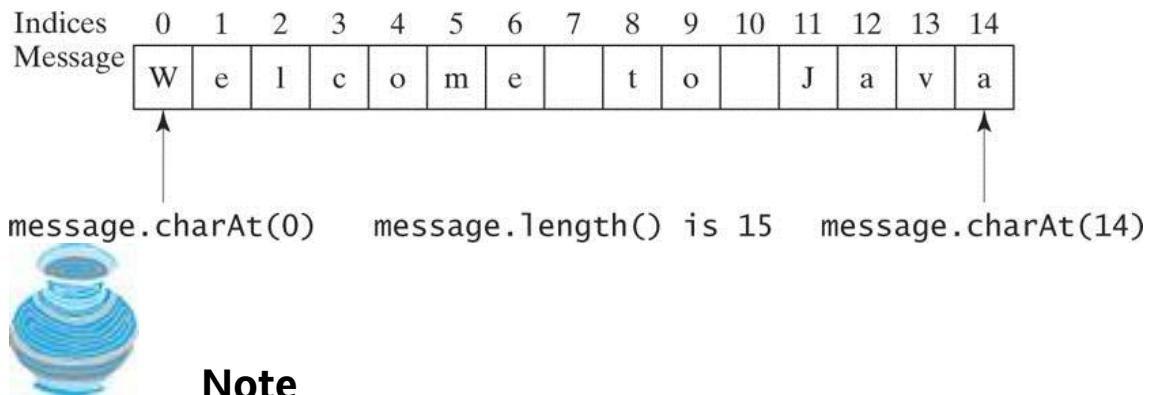
`length` is a method in the `String` class but is a property of an array object. So you have to use `s.length()` to get the number of characters in string `s`, and `a.length` to get the number of elements in array `a`.

`length()`

The `s.charAt(index)` method can be used to retrieve a specific character in a string `s`, where the index is between `0` and `s.length() - 1`. For example, `message.charAt(0)` returns the character `W`, as shown in [Figure 9.4](#).

`charAt(index)`

FIGURE 9.4 A String object is represented using an array internally.



Note

When you use a string, you often know its literal value. For convenience, Java allows you to use the string literal to refer directly to strings without creating new variables. Thus, `"Welcome to Java".charAt(0)` is correct and returns `W`.

string literal



Note

A string value is represented using a private array variable internally. The array cannot be accessed outside of the `String` class. The `String` class provides many public methods, such as `length()` and `charAt(index)`, to retrieve the array information. This is a good example of encapsulation: the data field of the class is hidden from the user through the private modifier, and thus the user cannot directly manipulate it. If the array were not private, the user would be able to change the string content by modifying the array. This would violate the tenet that the `String` class is immutable.

encapsulating string



Caution

Attempting to access characters in a string `s` out of bounds is a common programming error. To avoid it, make sure that you do not use an index beyond `s.length() - 1`. For example, `s.charAt(s.length())` would cause a `StringIndexOutOfBoundsException`.

string index range

You can use the `concat` method to concatenate two strings. The statement shown below, for example, concatenates strings `s1` and `s2` into `s3`:

```
String s3 = s1.concat(s2);  
  
s1.concat(s2)
```

Since string concatenation is heavily used in programming, Java provides a convenient way to accomplish it. You can use the plus (+) sign to concatenate two or more strings. So the above statement is equivalent to

```
String s3 = s1 + s2;  
  
s1 + s2
```

The following code combines the strings `message`, " and ", and "HTML" into one string:

```
String myString = message + " and " + "HTML";
```

Recall that the + sign can also concatenate a number with a string. In this case, the number is converted into a string and then concatenated. Note that at least one of the operands must be a string in order for concatenation to take place.

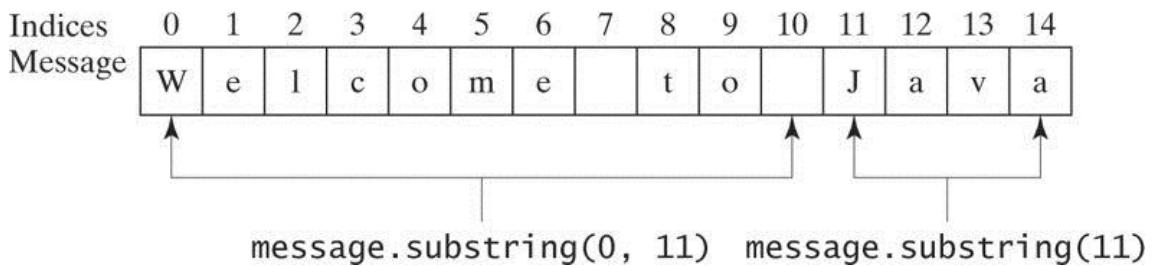
9.2.5 Obtaining Substrings

You can obtain a single character from a string using the `charAt` method, as shown in [Figure 9.3](#). You can also obtain a substring from a string using the `substring` method in the `String` class, as shown in [Figure 9.5](#).

FIGURE 9.5 The String class contains the methods for obtaining substrings.

java.lang.String	
<code>+substring(beginIndex: int): String</code>	Returns this string's substring that begins with the character at the specified <code>beginIndex</code> and extends to the end of the string, as shown in Figure 9.6.
<code>+substring(beginIndex: int, endIndex: int): String</code>	Returns this string's substring that begins at the specified <code>beginIndex</code> and extends to the character at index <code>endIndex - 1</code> , as shown in Figure 9.6. Note that the character at <code>endIndex</code> is not part of the substring.

FIGURE 9.6 The `substring` method obtains a substring from a string.



For example,

```
String message = "Welcome to Java".substring(0, 11) +
    "HTML";
```

The string `message` now becomes `"Welcome to HTML"`.



Note

If `beginIndex <= endIndex` returns an empty string with length **0**. If `beginIndex > endIndex`, it would be a runtime error.

`beginIndex <= endIndex`

9.2.6 Converting, Replacing, and Splitting Strings

The `String` class provides the methods for converting, replacing, and splitting strings, as shown in [Figure 9.7](#).

FIGURE 9.7 The String class contains the methods for converting, replacing, and splitting strings.

java.lang.String	
<code>+toLowerCase(): String</code>	Returns a new string with all characters converted to lowercase.
<code>+toUpperCase(): String</code>	Returns a new string with all characters converted to uppercase.
<code>+trim(): String</code>	Returns a new string with blank characters trimmed on both sides.
<code>+replace(oldChar: char, newChar: char): String</code>	Returns a new string that replaces all matching characters in this string with the new character.
<code>+replaceFirst(oldString: String, newString: String): String</code>	Returns a new string that replaces the first matching substring in this string with the new substring.
<code>+replaceAll(oldString: String, newString: String): String</code>	Returns a new string that replaces all matching substrings in this string with the new substring.
<code>+split(delimiter: String): String[]</code>	Returns an array of strings consisting of the substrings split by the delimiter.

Once a string is created, its contents cannot be changed. The methods `toLowerCase`, `toUpperCase`, `trim`, `replace`, `replaceFirst`, and `replaceAll` return a new string derived from the original string (without changing the original string!). The `toLowerCase` and `toUpperCase` methods return a new string by converting all the characters in the string to lowercase or uppercase. The `trim` method returns a new string by eliminating blank characters from both ends of the string. Several versions of the `replace` methods are provided to replace a character or a substring in the string with a new character or a new substring.

For example,

<code>"Welcome".toLowerCase()</code>	returns a new string, <code>welcome</code> .	<code>toLowerCase()</code>
<code>"Welcome".toUpperCase()</code>	returns a new string, <code>WELCOME</code> .	<code>toUpperCase()</code>
<code>" Welcome ".trim()</code>	returns a new string, <code>Welcome</code> .	<code>trim()</code>
<code>"Welcome".replace('e', 'A')</code>	returns a new string, <code>WALcomeA</code> .	<code>replace</code>
<code>"Welcome".replaceFirst("e", "AB")</code>	returns a new string, <code>WABLcome</code> .	<code>replaceFirst</code>
<code>"Welcome".replace("e", "AB")</code>	returns a new string, <code>WABLcomAB</code> .	<code>replace</code>
<code>"Welcome".replace("el", "AB")</code>	returns a new string, <code>WABcome</code> .	<code>replace</code>

The `split` method can be used to extract tokens from a string with the specified delimiters. For example, the following code

```
String[] tokens = "Java#HTML#Perl".split("#", 0);
for (int i = 0; i < tokens.length; i++)
    System.out.print(tokens[i] + " ");

split
```

displays

Java HTML Perl

9.2.7 Matching, Replacing and Splitting by Patterns

You can match, replace, or split a string by specifying a pattern. This is an extremely useful and powerful feature, commonly known as *regular expression*. Regular expressions seem complex to beginning students. For this reason, two simple patterns are used in this section. Please refer to Supplement III.H, “Regular Expressions,” for further studies.

regular expression

Let us begin with the `matches` method in the `String` class. At first glance, the `matches` method is very similar to the `equals` method. For example, the following two statements both evaluate to `true`.

matches (regex)

```
"Java".matches("Java");  
"Java".equals("Java");
```

However, the `matches` method is more powerful. It can match not only a fixed string, but also a set of strings that follow a pattern. For example, the following statements all evaluate to `true`:

```
"Java is fun".matches("Java.*")  
"Java is cool".matches("Java.*")  
"Java is powerful".matches("Java.*")
```

`"Java.*"` in the preceding statements is a regular expression. It describes a string pattern that begins with Java followed by *any* zero or more characters. Here, the substring `.`* matches any zero or more characters.

The `replaceAll`, `replaceFirst`, and `split` methods can be used with a regular expression. For example, the following statement returns a new string that replaces `$`, `+`, or `#` in `"a+b$#c"` with the string `NNN`.

```
replaceAll(regex)           String s = "a+b$#c".replaceAll("[+$#]", "NNN");  
                           System.out.println(s);
```

Here the regular expression `[$+#]` specifies a pattern that matches `$`, `+`, or `#`. So, the output is `aNNNbNNNNNNc`.

The following statement splits the string into an array of strings delimited by punctuation marks.

```
split(regex)               String[] tokens = "Java,C?C#,C++".split("[.,;:]");  
  
                           for (int i = 0; i < tokens.length; i++)  
                               System.out.println(tokens[i]);
```

Here the regular expression `[.,;:]` specifies a pattern that matches `,`, `,`, `:`, `,`, or `?`. Each of these characters is a delimiter for splitting the string. So, the string is split into `Java`, `C`, `C #`, and `C++`, which are stored into array `tokens`.

9.2.8 Finding a Character or a Substring in a String

The `String` class provides several overloaded `indexOf` and `lastIndexOf` methods to find a character or a substring in a string, as shown in [Figure 9.8](#).

For example,

```

indexOf           "Welcome to Java".indexOf('W') returns 0.
                  "Welcome to Java".indexOf('o') returns 4.
                  "Welcome to Java".indexOf('o', 5) returns 9.
                  "Welcome to Java".indexOf("come") returns 3.
                  "Welcome to Java".indexOf("Java", 5) returns 11.
                  "Welcome to Java".indexOf("java", 5) returns -1.

lastIndexOf      "Welcome to Java".lastIndexOf('W') returns 0.
                  "Welcome to Java".lastIndexOf('o') returns 9.
                  "Welcome to Java".lastIndexOf('o', 5) returns 4.
                  "Welcome to Java".lastIndexOf("come") returns 3.
                  "Welcome to Java".lastIndexOf("Java", 5) returns -1.
                  "Welcome to Java".lastIndexOf("Java") returns 11.

```

FIGURE 9.8 The String class contains the methods for matching substrings.

java.lang.String	
+indexOf(ch: char): int	Returns the index of the first occurrence of ch in the string. Returns -1 if not matched.
+indexOf(ch: char, fromIndex: int): int	Returns the index of the first occurrence of ch after fromIndex in the string. Returns -1 if not matched.
+indexOf(s: String): int	Returns the index of the first occurrence of string s in this string. Returns -1 if not matched.
+indexOf(s: String, fromIndex: int): int	Returns the index of the first occurrence of string s in this string after fromIndex. Returns -1 if not matched.
+lastIndexOf(ch: int): int	Returns the index of the last occurrence of ch in the string. Returns -1 if not matched.
+lastIndexOf(ch: int, fromIndex: int): int	Returns the index of the last occurrence of ch before fromIndex in this string. Returns -1 if not matched.
+lastIndexOf(s: String): int	Returns the index of the last occurrence of string s. Returns -1 if not matched.
+lastIndexOf(s: String, fromIndex: int): int	Returns the index of the last occurrence of string s before fromIndex. Returns -1 if not matched.

9.2.9 Conversion between Strings and Arrays

Strings are not arrays, but a string can be converted into an array, and vice versa. To convert a string to an array of characters, use the `toCharArray` method. For example, the following statement converts the string `"Java"` to an array.

```
char[] chars = "Java".toCharArray();
```

`toCharArray`

So `chars[0]` is & J &, `chars[1]` is & a &, `chars[2]` is & v &, and `chars[3]` is &a&.

You can also use the `getChars(int srcBegin, int srcEnd, char[] dst, int dst-Begin)` method to copy a substring of the string from index `srcBegin` to index `srcEnd-1` into a character array `dst` starting from index `dstBegin`. For

example, the following code copies a substring “3720” in “CS3720” from index **2** to index **6-1** into the character array **dst** starting from index **4**.

```
char[] dst = {'J', 'A', 'V', 'A', '1', '3', '0', '1'};           getChars
"CS3720".getChars(2, 6, dst, 4);
```

Thus **dst** becomes **& J &, &A&, &V&, &A&, &3&, &7&, &2&, &0&**

To convert an array of characters into a string, use the **String(char[])** constructor or the **valueOf(char[])** method. For example, the following statement constructs a string from an array using the **String** constructor.

```
String str = new String(new char[]{'J', 'a', 'v', 'a'});
```

The next statement constructs a string from an array using the **valueOf** method.

```
String str = String.valueOf(new char[]{'J', 'a', 'v', 'a'});           valueOf
```

9.2.10 Converting Characters and Numeric Values to Strings

The static **valueOf** method can be used to convert an array of characters into a string. There are several overloaded versions of the **valueOf** method that can be used to convert a character and numeric values to strings with different parameter types, **char**, **double**, **long**, **int**, and **float**, as shown in [Figure 9.9](#).

overloaded **valueOf**

FIGURE 9.9 The String class contains the static methods for creating strings from primitive type values.

java.lang.String	
+valueOf(c: char): String	Returns a string consisting of the character c.
+valueOf(data: char[]): String	Returns a string consisting of the characters in the array.
+valueOf(d: double): String	Returns a string representing the double value.
+valueOf(f: float): String	Returns a string representing the float value.
+valueOf(i: int): String	Returns a string representing the int value.
+valueOf(l: long): String	Returns a string representing the long value.
+valueOf(b: boolean): String	Returns a string representing the boolean value.

For example, to convert a **double** value **5.44** to a string, use **String.valueOf(5.44)**. The return value is a string consisting of the characters **&5&, .&, &4&**, and **&4&**.



Note

Use `Double.parseDouble(str)` or `Integer.parseInt(str)` to convert a string to a `double` value or an `int` value.

9.2.11 Formatting Strings

The `String` class contains the static `format` method in the `String` class to create a formatted string. The syntax to invoke this method is

```
String.format(format, item1, item2, ..., item k)
```

This method is similar to the `printf` method except that the `format` method returns a formatted string, whereas the `printf` method displays a formatted string. For example,

```
String s = String.format("%5.2f", 45.556);
```

creates a formatted string `"45.56"`.

9.2.12 Problem: Checking Palindromes

A string is a palindrome if it reads the same forward and backward. The words “mom,” “dad,” and “noon,” for instance, are all palindromes.



Video Note

Check palindrome

The problem is to write a program that prompts the user to enter a string and reports whether the string is a palindrome. One solution is to check whether the first character in the string is the same as the last character. If so, check whether the second character is the same as the second-to-last character. This process continues until a mismatch is found or all the characters in the string are checked, except for the middle character if the string has an odd number of characters.

To implement this idea, use two variables, say `low` and `high`, to denote the position of two characters at the beginning and the end in a string `s`, as shown in [Listing 9.1](#) (lines 22, 25). Initially, `low` is `0` and `high` is `s.length() - 1`. If the two characters at these

positions match, increment `low` by 1 and decrement `high` by 1 (lines 31–32). This process continues until (`low >= high`) or a mismatch is found.

LISTING 9.1 CheckPalindrome.java

```
1 import java.util.Scanner;  
2  
3 public class CheckPalindrome {  
4     /** Main method */  
  
5     public static void main(String[] args) {  
6         // Create a Scanner  
7         Scanner input = new Scanner(System.in);  
8  
9         // Prompt the user to enter a string  
10        System.out.print("Enter a string: ");  
11        String s = input.nextLine();  
12  
13        if (isPalindrome(s))  
14            System.out.println(s + " is a palindrome");  
15        else  
16            System.out.println(s + " is not a palindrome");  
17    }  
18  
19    /** Check if a string is a palindrome */  
20    public static boolean isPalindrome(String s) {  
21        // The index of the first character in the string  
22        int low = 0;  
23  
24        // The index of the last character in the string  
25        int high = s.length() - 1;  
26  
27        while (low < high) {  
28            if (s.charAt(low) != s.charAt(high))  
29                return false; // Not a palindrome  
30  
31            low++;  
32            high--;  
33        }  
34  
35        return true; // The string is a palindrome  
36    }  
37 }
```

input string

low index

high index

update indices



Enter a string: 

noon is a palindrome

Enter a string: 

moon is not a palindrome

The `nextLine()` method in the `Scanner` class (line 11) reads a line into `s`.
`isPalindrome(s)` checks whether `s` is a palindrome (line 13).

9.2.13 Problem: Converting Hexadecimals to Decimals

Section 5.7 gives a program that converts a decimal to a hexadecimal. This section presents a program that converts a hex number into a decimal.

Given a hexadecimal number $h_nh_{n-1}h_{n-2} \dots h_2h_1h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

For example, the hex number `AB8C` is

$$10 \times 16^3 + 11 \times 16^2 + 8 \times 16^1 + 12 \times 16^0 = 43916$$

Our program will prompt the user to enter a hex number as a string and convert it into a decimal using the following method:

```
public static int hexToDecimal(String hex)
```

A brute-force approach is to convert each hex character into a decimal number, multiply it by for a hex digit at the `i`'s position, and add all the items together to obtain the equivalent decimal value for the hex number.

Note that

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_1 \times 16^1 + h_0 \times 16^0 = ((h_n \times 16 + h_{n-1}) \times 16 + h_{n-2}) \times 16 + \dots + h_1) \times 16 + h_0$$

This observation leads to the following efficient algorithm for converting a hex string to a decimal number:

```
int decimalValue = 0;
for (int i = 0; i < hex.length(); i++) {
    char hexChar = hex.charAt(i);
```

```
    decimalValue = decimalValue * 16 +  
    hexCharToDecimal(hexChar);  
}
```

Here is a trace of the algorithm for hex number **AB8C**:

i

hexChar

hexCharToDecimal(hexChar)

decimalValue

before the loop

0

after the 1st iteration

0

A

10

10

after the 2nd iteration

1

B

11

$10 * 16 + 11$

after the 3rd iteration

2

8

8

$(10 * 16 + 11) * 16 + 8$

after the 4th iteration

3

C

12

$((10 * 16 + 11) * 16 + 8) * 16 + 12$

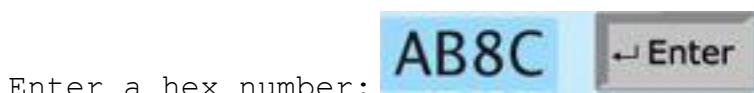
[Listing 9.2](#) gives the complete program.

LISTING 9.2 HexToDecimalConversion.java

```
1 import java.util.Scanner;
2
3 public class HexToDecimalConversion {
4     /** Main method */
5     public static void main(String[] args) {
6         // Create a Scanner
7         Scanner input = new Scanner(System.in);
8
9         // Prompt the user to enter a string
10        System.out.print("Enter a hex number: ");
11        String hex = input.nextLine();
12
13        System.out.println("The decimal value for hex number "
14            + hex + " is " + hexToDecimal(hex.toUpperCase()));
15    }
16
17    public static int hexToDecimal(String hex) {
18        int decimalValue = 0;
19        for (int i = 0; i < hex.length(); i++) {
20            char hexChar = hex.charAt(i);
21            decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
22        }
23    }
}
```

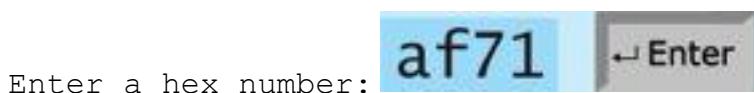
```

24     return decimalValue;
25 }
26
27 public static int hexCharToDecimal(char ch) {
28     if (ch >= 'A' && ch <= 'F')
29         return 10 + ch - 'A';
30     else // ch is '0', '1', ..., or '9'
31         return ch - '0';
32 }
33 }
```



Enter a hex number:

The decimal value for hex number AB8C is 43916



Enter a hex number:

The decimal value for hex number af71 is 44913

The program reads a string from the console (line 11), and invokes the `hexToDecimal` method to convert a hex string to decimal number (line 14). The characters can be in either lowercase or uppercase. They are converted to uppercase before invoking the `hexToDecimal` method (line 14).

The `hexToDecimal` method is defined in lines 17–25 to return an integer. The length of the string is determined by invoking `hex.length()` in line 19.

The `hexCharToDecimal` method is defined in lines 27–32 to return a decimal value for a hex character. The character can be in either lowercase or uppercase. Recall that to subtract two characters is to subtract their Unicodes. For example, `&5& - &0&` is `5`.

9.3 The Character Class

Java provides a wrapper class for every primitive data type. These classes are `Character`, `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, and `Double` for `char`, `boolean`, `byte`, `short`, `int`, `long`, `float`, and `double`. All these classes are in the `java.lang`

package. They enable the primitive data values to be treated as objects. They also contain useful methods for processing primitive values. This section introduces the **Character** class. The other wrapper classes will be introduced in Chapter 14, “Abstract Classes and Interfaces.”

The **Character** class has a constructor and several methods for determining a character’s category (uppercase, lowercase, digit, and so on) and for converting characters from uppercase to lowercase, and vice versa, as shown in [Figure 9.10](#).

You can create a **Character** object from a **char** value. For example, the following statement creates a **Character** object for the character **&a&**.

```
Character character =new Character(&a&);
```

The **charValue** method returns the character value wrapped in the **Character** object. The **compareTo** method compares this character with another character and returns an integer that is the difference between the Unicodes of this character and the other character. The **equals** method returns **true** if and only if the two characters are the same. For example, suppose **charObject** is **new Character(&b&)**:

```
charObject.compareTo(new Character(&a&)) returns 1  
charObject.compareTo(new Character(&b&)) returns 0  
charObject.compareTo(new Character(&c&)) returns -1  
charObject.compareTo(new Character(&d&)) returns -2
```

FIGURE 9.10 The Character class provides the methods for manipulating a character.

java.lang.Character	
+Character(value: char)	Constructs a character object with char value.
+charValue(): char	Returns the char value from this object.
+compareTo(anotherCharacter: Character): int	Compares this character with another.
+equals(anotherCharacter: Character): boolean	Returns true if this character is equal to another.
+isDigit(ch: char): boolean	Returns true if the specified character is a digit.
+isLetter(ch: char): boolean	Returns true if the specified character is a letter.
+isLetterOrDigit(ch: char): boolean	Returns true if the character is a letter or a digit.
+isLowerCase(ch: char): boolean	Returns true if the character is a lowercase letter.
+isUpperCase(ch: char): boolean	Returns true if the character is an uppercase letter.
+toLowerCase(ch: char): char	Returns the lowercase of the specified character.
+toUpperCase(ch: char): char	Returns the uppercase of the specified character.
charObject.equals(new Character(&b&)) returns true	
charObject.equals(new Character(&d&)) returns false	

Most of the methods in the **Character** class are static methods. The **isDigit(char ch)** method returns **true** if the character is a digit. The **isLetter(char ch)** method

returns `true` if the character is a letter. The `isLetterOrDigit(char ch)` method returns `true` if the character is a letter or a digit. The `isLowerCase(char ch)` method returns `true` if the character is a lowercase letter. The `isUpperCase(char ch)` method returns `true` if the character is an uppercase letter. The `toLowerCase(char ch)` method returns the lowercase letter for the character, and the `toUpperCase(char ch)` method returns the uppercase letter for the character.

9.3.1 Problem: Counting Each Letter in a String

The problem is to write a program that prompts the user to enter a string and counts the number of occurrences of each letter in the string regardless of case.

Here are the steps to solve this problem:

1. Convert all the uppercase letters in the string to lowercase using the `toLowerCase` method in the `String` class.
2. Create an array, say `counts` of 26 `int` values, each of which counts the occurrences of a letter. That is, `counts[0]` counts the number of `a`'s, `counts[1]` counts the number of `b`'s, and so on.
3. For each character in the string, check whether it is a (lowercase) letter. If so, increment the corresponding count in the array.

[Listing 9.3](#) gives the complete program:

LISTING 9.3 CountEachLetter.java

```
1 import java.util.Scanner;  
2  
3 public class CountEachLetter {  
4     /** Main method */  
5     public static void main(String[] args) {  
6         // Create a Scanner  
7         Scanner input = new Scanner(System.in);
```

```

8      // Prompt the user to enter a string
9      System.out.print("Enter a string: ");
10     String s = input.nextLine();                                input string
11
12     // Invoke the countLetters method to count each letter
13     int[] counts = countLetters(s.toLowerCase());                count letters
14
15     // Display results
16     for (int i = 0; i < counts.length; i++) {
17         if (counts[i] != 0)
18             System.out.println((char)('a' + i) + " appears " +
19                     counts[i] + ((counts[i] == 1) ? " time" : " times"));
20     }
21 }
22
23
24     /** Count each letter in the string */
25     public static int[] countLetters(String s) {
26         int[] counts = new int[26];
27
28         for (int i = 0; i < s.length(); i++) {
29             if (Character.isLetter(s.charAt(i)))
30                 counts[s.charAt(i) - 'a']++;                         count a letter
31         }
32
33         return counts;
34     }
35 }
```



Enter a string: **abababx**

a appears 3 times

b appears 3 times

x appears 1 time

The main method reads a line (line 11) and counts the number of occurrences of each letter in the string by invoking the `countLetters` method (line 14). Since the case of the letters is ignored, the program uses the `toLowerCase` method to convert the string into all lowercase and pass the new string to the `countLetters` method.

The `countLetters` method (lines 25–34) returns an array of 26 elements. Each element counts the number of occurrences of a letter in the string `s`. The method processes each character in the string. If the character is a letter, its corresponding count is increased by 1. For example, if the character (`s.charAt(i)`) is `&a&`, the corresponding count is `counts[&a& &a&]` (i.e., `counts[0]`). If the character is `&b&`, the corresponding count is `counts[&b& &a&]` (i.e., `counts[1]`), since the Unicode of `&b&` is 1 more than that of `&a&`. If the character is `&z&`, the corresponding count is `counts[&z& - &a&]` (i.e., `counts[25]`), since the Uni-code of `&z&` is 25 more than that of `&a&`.

9.4 The `StringBuilder/StringBuffer` Class

The `StringBuilder/StringBuffer` class is an alternative to the `String` class. In general, a `StringBuilder/StringBuffer` can be used wherever a string is used. `StringBuilder/StringBuffer` is more flexible than `String`. You can add, insert, or append new contents into a `StringBuilder` or a `StringBuffer`, whereas the value of a `String` object is fixed, once the string is created.

The `StringBuilder` class is similar to `StringBuffer` except that the methods for modifying buffer in `StringBuffer` are synchronized. Use `StringBuffer` if it may be accessed by multiple tasks concurrently. Using `StringBuilder` is more efficient if it is accessed by a single task. The constructors and methods in `StringBuffer` and `StringBuilder` are almost the same. This section covers `StringBuilder`. You may replace `StringBuilder` by `StringBuffer`. The program can compile and run without any other changes.

`StringBuilder`

The `StringBuilder` class has three constructors and more than 30 methods for managing the builder and modifying strings in the builder. You can create an empty string builder or a string builder from a string using the constructors, as shown in [Figure 9.11](#).

`StringBuilder` constructors

FIGURE 9.11 The `StringBuilder` class contains the constructors for creating instances of `StringBuilder`.

<code>java.lang.StringBuilder</code>	
<code>+StringBuilder()</code>	Constructs an empty string builder with capacity 16.
<code>+StringBuilder(capacity: int)</code>	Constructs a string builder with the specified capacity.
<code>+StringBuilder(s: String)</code>	Constructs a string builder with the specified string.

9.4.1 Modifying Strings in the `StringBuilder`

You can append new contents at the end of a string builder, insert new contents at a specified position in a string builder, and delete or replace characters in a string builder, using the methods listed in [Figure 9.12](#):

FIGURE 9.12 The `StringBuilder` class contains the methods for modifying string builders.

java.lang.StringBuilder	
+append(data: char[]): StringBuilder	Appends a <code>char</code> array into this string builder.
+append(data: char[], offset: int, len: int): StringBuilder	Appends a subarray in <code>data</code> into this string builder.
+append(v: aPrimitiveType): StringBuilder	Appends a primitive type value as a string to this builder.
+append(s: String): StringBuilder	Appends a string to this string builder.
+delete(startIndex: int, endIndex: int): StringBuilder	Deletes characters from <code>startIndex</code> to <code>endIndex-1</code> .
+deleteCharAt(index: int): StringBuilder	Deletes a character at the specified index.
+insert(index: int, data: char[], offset: int, len: int): StringBuilder	Inserts a subarray of the data in the array to the builder at the specified index.
+insert(offset: int, data: char[]): StringBuilder	Inserts data into this builder at the position offset.
+insert(offset: int, b: aPrimitiveType): StringBuilder	Inserts a value converted to a string into this builder.
+insert(offset: int, s: String): StringBuilder	Inserts a string into this builder at the position offset.
+replace(startIndex: int, endIndex: int, s: String): StringBuilder	Replaces the characters in this builder from <code>startIndex</code> to <code>endIndex-1</code> with the specified string.
+reverse(): StringBuilder	Reverses the characters in the builder.
+setCharAt(index: int, ch: char): void	Sets a new character at the specified index in this builder.

The `StringBuilder` class provides several overloaded methods to append `boolean`, `char`, `char array`, `double`, `float`, `int`, `long`, and `String` into a string builder. For example, the following code appends strings and characters into `stringBuilder` to form a new string, `"Welcome to Java"`.

```
StringBuilder stringBuilder = new StringBuilder();

stringBuilder.append("Welcome");
stringBuilder.append(' ');
stringBuilder.append("to");
stringBuilder.append(' ');
stringBuilder.append("Java");
```

The `StringBuilder` class also contains overloaded methods to insert `boolean`, `char`, `char array`, `double`, `float`, `int`, `long`, and `String` into a string builder.

Consider the following code:

```
stringBuilder.insert(11, "HTML and ");
```

insert

Suppose `stringBuilder` contains `"Welcome to Java"` before the `insert` method is applied. This code inserts `"HTML and "` at position 11 in `stringBuilder` (just before `J`). The new `stringBuilder` is `"Welcome to HTML and Java"`.

You can also delete characters from a string in the builder using the two `delete` methods, reverse the string using the `reverse` method, replace characters using the `replace` method, or set a new character in a string using the `setCharAt` method.

For example, suppose `stringBuilder` contains "Welcome to Java" before each of the following methods is applied.

<code>stringBuilder.delete(8, 11)</code> changes the builder to Welcome Java.	<code>delete</code>
<code>stringBuilder.deleteCharAt(8)</code> changes the builder to Welcome o Java.	<code>deleteCharAt</code>
<code>stringBuilder.reverse()</code> changes the builder to avaJ ot emocleW.	<code>reverse</code>
<code>stringBuilder.replace(11, 15, "HTML")</code> changes the builder to Welcome to HTML.	<code>replace</code>
<code>stringBuilder.setCharAt(0, 'w')</code> sets the builder to welcome to Java.	<code>setCharAt</code>

All these modification methods except `setCharAt` do two things:

1. Change the contents of the string builder
2. Return the reference of the string builder

For example, the following statement

```
StringBuilder stringBuilder1 = stringBuilder.reverse();
```

reverses the string in the builder and assigns the reference of the builder to `stringBuilder1`. Thus, `stringBuilder` and `stringBuilder1` both point to the same `StringBuilder` object. Recall that a value-returning method may be invoked as a statement, if you are not interested in the return value of the method. In this case, the return value is simply ignored. For example, in the following statement

```
ignore return value  
stringBuilder.reverse();
```

the return value is ignored.



Tip

If a string does not require any change, use `String` rather than `StringBuilder`. Java can perform some optimizations for `String`, such as sharing interned strings.

`String` or `StringBuilder`?

9.4.2 The `toString`, `capacity`, `length`, `setLength`, and `charAt` Methods

The **StringBuilder** class provides the additional methods for manipulating a string builder and obtaining its properties, as shown in [Figure 9.13](#).

FIGURE 9.13 The **StringBuilder** class contains the methods for modifying string builders.

java.lang.StringBuilder	
+ toString() : String	Returns a string object from the string builder.
+ capacity() : int	Returns the capacity of this string builder.
+ charAt(index: int) : char	Returns the character at the specified index.
+ length() : int	Returns the number of characters in this builder.
+ setLength(newLength: int) : void	Sets a new length in this builder.
+ substring(startIndex: int) : String	Returns a substring starting at startIndex .
+ substring(startIndex: int, endIndex: int) : String	Returns a substring from startIndex to endIndex-1 .
+ trimToSize() : void	Reduces the storage size used for the string builder.

The **capacity()** method returns the current capacity of the string builder. The capacity is the number of characters it is able to store without having to increase its size.

capacity()

The **length()** method returns the number of characters actually stored in the string builder. The **setLength(newLength)** method sets the length of the string builder. If the **newLength** argument is less than the current length of the string builder, the string builder is truncated to contain exactly the number of characters given by the **newLength** argument. If the **newLength** argument is greater than or equal to the current length, sufficient null characters (**\u0000**) are appended to the string builder so that **length** becomes the **newLength** argument. The **newLength** argument must be greater than or equal to **0**.

length()

setLength(int)

The **charAt(index)** method returns the character at a specific **index** in the string builder. The index is **0** based. The first character of a string builder is at index **0**, the next at index **1**, and so on. The **index** argument must be greater than or equal to **0**, and less than the length of the string builder.

charAt(int)



Note

The length of the string is always less than or equal to the capacity of the builder. The length is the actual size of the string stored in the builder, and the capacity is the current size of the builder. The builder's capacity is automatically increased if more characters are added to exceed its capacity. Internally, a string builder is an array of characters, so the builder's capacity is the size of the array. If the builder's capacity is exceeded, the array is replaced by a new array. The new array size is **2 * (the previous array size + 1)**.

length and capacity



Tip

You can use **new StringBuilder(initialCapacity)** to create a **StringBuilder** with a specified initial capacity. By carefully choosing the initial capacity, you can make your program more efficient. If the capacity is always larger than the actual length of the builder, the JVM will never need to reallocate memory for the builder. On the other hand, if the capacity is too large, you will waste memory space. You can use the **trimToSize()** method to reduce the capacity to the actual size.

initial capacity

trimToSize()

9.4.3 Problem: Ignoring Nonalphanumeric Characters When Checking Palindromes

[Listing 9.1](#), CheckPalindrome.java, considered all the characters in a string to check whether it was a palindrome. Write a new program that ignores nonalphanumeric characters in checking whether a string is a palindrome.

Here are the steps to solve the problem:

1. Filter the string by removing the nonalphanumeric characters. This can be done by creating an empty string builder, adding each alphanumeric character in the string to a string builder, and returning the string from the string builder. You can use the

`isLetterOrDigit(ch)` method in the `Character` class to check whether character `ch` is a letter or a digit.

2. Obtain a new string that is the reversal of the filtered string. Compare the reversed string with the filtered string using the `equals` method.

The complete program is shown in [Listing 9.4](#).

LISTING 9.4

PalindromeIgnoreNonAlphanumeric.java

```

1 import java.util.Scanner;
2
3 public class PalindromeIgnoreNonAlphanumeric {
4     /** Main method */
5     public static void main(String[] args) {
6         // Create a Scanner
7         Scanner input = new Scanner(System.in);
8
9         // Prompt the user to enter a string
10        System.out.print("Enter a string: ");
11        String s = input.nextLine();
12
13        // Display result
14        System.out.println("Ignoring nonalphanumeric characters, \nis "
15            + s + " a palindrome? " + isPalindrome(s));
16    }
17
18    /** Return true if a string is a palindrome */
19    public static boolean isPalindrome(String s) {
20        // Create a new string by eliminating nonalphanumeric chars
21        String s1 = filter(s);
22
23        // Create a new string that is the reversal of s1
24        String s2 = reverse(s1);
25
26        // Compare if the reversal is the same as the original string
27        return s2.equals(s1);
28    }
29
30    /** Create a new string by eliminating nonalphanumeric chars */
31    public static String filter(String s) {
32        // Create a string builder
33        StringBuilder stringBuilder = new StringBuilder();
34
35        // Examine each char in the string to skip alphanumeric char
36        for (int i = 0; i < s.length(); i++) {
37            if (Character.isLetterOrDigit(s.charAt(i))) {
38                stringBuilder.append(s.charAt(i));
39            }
40        }
41
42        // Return a new filtered string
43        return stringBuilder.toString();
44    }
45
46    /** Create a new string by reversing a specified string */
47    public static String reverse(String s) {
48        StringBuilder stringBuilder = new StringBuilder(s);
49        stringBuilder.reverse(); // Invoke reverse in StringBuilder
50        return stringBuilder.toString();
51    }
52 }

```

check palindrome

add letter or digit



Enter a string:

↵ Enter

```
Ignoring nonalphanumeric characters,  
is ab<c>cb?a a palindrome? true
```

Enter a string: **abcc><?cab** 

```
Ignoring nonalphanumeric characters,  
is abcc><?cab a palindrome? false
```

The `filter(String s)` method (lines 31–44) examines each character in string `s` and copies it to a string builder if the character is a letter or a numeric character. The `filter` method returns the string in the builder. The `reverse(String s)` method (lines 47–52) creates a new string that reverses the specified string `s`. The `filter` and `reverse` methods both return a new string. The original string is not changed.

The program in [Listing 9.1](#) checks whether a string is a palindrome by comparing pairs of characters from both ends of the string. [Listing 9.4](#) uses the `reverse` method in the `StringBuilder` class to reverse the string, then compares whether the two strings are equal to determine whether the original string is a palindrome.

9.5 Command-Line Arguments

Perhaps you have already noticed the unusual declarations for the `main` method, which has parameter `args` of `String[]` type. It is clear that `args` is an array of strings. The `main` method is just like a regular method with a parameter. You can call a regular method by passing actual parameters. Can you pass arguments to `main`? Yes, of course you can. For example, the `main` method in class `TestMain` is invoked by a method in `A`, as shown below:

```
public class A {  
    public static void main(String[] args) {  
        String[] strings = {"New York",  
                           "Boston", "Atlanta"};  
        TestMain.main(strings);  
    }  
}
```

```
public class TestMain {  
    public static void main(String[] args) {  
        for (int i = 0; i < args.length; i++)  
            System.out.println(args[i]);  
    }  
}
```

A main method is just a regular method. Furthermore, you can pass arguments from the command line.

9.5.1 Passing Strings to the `main` Method

You can pass strings to a `main` method from the command line when you run the program. The following command line, for example, starts the program `TestMain` with three strings: `arg0`, `arg1`, and `arg2`:

```
java TestMain arg0 arg1 arg2
```

`arg0`, `arg1`, and `arg2` are strings, but they don't have to appear in double quotes on the command line. The strings are separated by a space. A string that contains a space must be enclosed in double quotes. Consider the following command line:

```
java TestMain "First num" alpha 53
```

It starts the program with three strings: `"First num"`, `alpha`, and `53`, a numeric string. Since `"First num"` is a string, it is enclosed in double quotes. Note that `53` is actually treated as a string. You can use `"53"` instead of `53` in the command line.

When the `main` method is invoked, the Java interpreter creates an array to hold the command-line arguments and pass the array reference to `args`. For example, if you invoke a program with `n` arguments, the Java interpreter creates an array like this one:

```
args = new String[n];
```

The Java interpreter then passes `args` to invoke the `main` method.



Note

If you run the program with no strings passed, the array is created with `new String[0]`. In this case, the array is empty with length `0`. `args` references to this empty array. Therefore, `args` is not `null`, but `args.length` is `0`.

9.5.2 Problem: Calculator

Suppose you are to develop a program that performs arithmetic operations on integers. The program receives three arguments: an integer followed by an operator and another integer. For example, to add two integers, use this command:



video note

Command-line argument

```
java Calculator 2 + 3
```

The program will display the following output:

2 + 3 = 5

[Figure 9.14](#) shows sample runs of the program.

FIGURE 9.14 The program takes three arguments (operand1 operator operand2) from the command line and displays the expression and the result of the arithmetic operation.

```
c:\book>java Calculator
Usage: java Calculator operand1 operator operand2
Add      → c:\book>java Calculator 63 + 40
          63 + 40 = 103
Subtract → c:\book>java Calculator 63 - 40
          63 - 40 = 23
Multiply  → c:\book>java Calculator 63 "*" 40
          63 * 40 = 2520
Divide    → c:\book>java Calculator 63 / 40
          63 / 40 = 1
c:\book>
```

The strings passed to the main program are stored in `args`, which is an array of strings. The first string is stored in `args[0]`, and `args.length` is the number of strings passed.

Here are the steps in the program:

- Use `args.length` to determine whether three arguments have been provided in the command line. If not, terminate the program using `System.exit(0)`.
- Perform a binary arithmetic operation on the operands `args[0]` and `args[2]` using the operator specified in `args[1]`.

The program is shown in [Listing 9.5](#).

LISTING 9.5 Calculator.java

```

1 public class Calculator {
2     /** Main method */
3     public static void main(String[] args) {
4         // Check number of strings passed
5         if (args.length != 3) {
6             System.out.println(
7                 "Usage: java Calculator operand1 operator operand2");
8             System.exit(0);
9         }
10
11        // The result of the operation
12        int result = 0;
13
14        // Determine the operator
15        switch (args[1].charAt(0)) {
16            case '+': result = Integer.parseInt(args[0]) +
17                          Integer.parseInt(args[2]);
18                break;
19            case '-': result = Integer.parseInt(args[0]) -
20                          Integer.parseInt(args[2]);
21                break;
22            case '*': result = Integer.parseInt(args[0]) *
23                          Integer.parseInt(args[2]);
24                break;
25            case '/': result = Integer.parseInt(args[0]) /
26                          Integer.parseInt(args[2]);
27        }
28
29        // Display result
30        System.out.println(args[0] + ' ' + args[1] + ' ' + args[2]
31                         + " = " + result);
32    }
33 }

```

`Integer.parseInt(args[0])` (line 16) converts a digital string into an integer. The string must consist of digits. If not, the program will terminate abnormally.



Note

In the sample run, `"**"` had to be used instead of `*` for the command
`java Calculator 63 ** 40`

The `*` symbol refers to all the files in the current directory when it is used on a command line. Therefore, in order to specify the multiplication operator, the `*` must be enclosed in quote marks in the command line. The following program displays all the files in the current directory when issuing the command `java Test *`:

```

public class Test {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++)
            System.out.println(args[i]);
    }
}

```

```
    }  
}  
  
special * character
```

9.6 The File Class

Data stored in variables, arrays, and objects are temporary; they are lost when the program terminates. To permanently store the data created in a program, you need to save them in a file on a disk or a CD. The file can be transported and can be read later by other programs. Since data are stored in files, this section introduces how to use the `File` class to obtain file properties and to delete and rename files. The next section introduces how to read/write data from/to text files.

why file?

Every file is placed in a directory in the file system. An *absolute file name* contains a file name with its complete path and drive letter. For example, `c:\book\Welcome.java` is the absolute file name for the file `Welcome.java` on the Windows operating system. Here `c:\book` is referred to as the *directory path* for the file. Absolute file names are machine dependent. On the Unix platform, the absolute file name may be `/home/liang/book/Welcome.java`, where `/home/liang/book` is the directory path for the file `Welcome.java`.

absolute file name

directory path

The `File` class is intended to provide an abstraction that deals with most of the machine-dependent complexities of files and path names in a machine-independent fashion. The `File` class contains the methods for obtaining file properties and for renaming and deleting files, as shown in [Figure 9.15](#). However, *the File class does not contain the methods for reading and writing file contents.*

FIGURE 9.15 The File class can be used to obtain file and directory properties and to delete and rename files.

java.io.File	
+File(pathname: String)	Creates a File object for the specified path name. The path name may be a directory or a file.
+File(parent: String, child: String)	Creates a File object for the child under the directory parent. The child may be a file name or a subdirectory.
+File(parent: File, child: String)	Creates a File object for the child under the directory parent. The parent is a File object. In the preceding constructor, the parent is a string.
+exists(): boolean	Returns true if the file or the directory represented by the File object exists.
+canRead(): boolean	Returns true if the file represented by the File object exists and can be read.
+canWrite(): boolean	Returns true if the file represented by the File object exists and can be written.
+isDirectory(): boolean	Returns true if the File object represents a directory.
+isFile(): boolean	Returns true if the File object represents a file.
+isAbsolute(): boolean	Returns true if the File object is created using an absolute path name.
+isHidden(): boolean	Returns true if the file represented in the File object is hidden. The exact definition of <i>hidden</i> is system dependent. On Windows, you can mark a file hidden in the File Properties dialog box. On Unix systems, a file is hidden if its name begins with a period character '.'.
+getAbsolutePath(): String	Returns the complete absolute file or directory name represented by the File object.
+getCanonicalPath(): String	Returns the same as getAbsolutePath() except that it removes redundant names, such as ".", and "..", from the path name, resolves symbolic links (on Unix platforms), and converts drive letters to standard uppercase (on Win32 platforms).
+getName(): String	Returns the last name of the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getName() returns test.dat.
+getPath(): String	Returns the complete directory and file name represented by the File object. For example, new File("c:\\book\\test.dat").getPath() returns c:\\book\\test.dat.
+getParent(): String	Returns the complete parent directory of the current directory or the file represented by the File object. For example, new File("c:\\book\\test.dat").getParent() returns c:\\book.
+lastModified(): long	Returns the time that the file was last modified.
+length(): long	Returns the size of the file, or 0 if it does not exist or if it is a directory.
+listFile(): File[]	Returns the files under the directory for a directory File object.
+delete(): boolean	Deletes this file. The method returns true if the deletion succeeds.
+renameTo(dest: File): boolean	Renames this file. The method returns true if the operation succeeds.

The file name is a string. The `File` class is a wrapper class for the file name and its directory path. For example, `new File("c:\\book")` creates a `File` object for the directory `c:\\book`, and `new File("c:\\book\\test.dat")` creates a `File` object for the file `c:\\book\\test.dat`, both on Windows. You can use the `File` class's `isDirectory()` method to check whether the object represents a directory, and the `isFile()` method to check whether the object represents a file.



Caution

The directory separator for Windows is a backslash (`\`). The backslash is a special character in Java and should be written as `\` in a string literal (see [Table 2.6](#)).

`\` in file names



Note

*Constructing a **File** instance does not create a file on the machine.* You can create a **File** instance for any file name regardless whether it exists or not. You can invoke the **exists()** method on a **File** instance to check whether the file exists.

Do not use absolute file names in your program. If you use a file name such as "**c:\\book\\Welcome.java**", it will work on Windows but not on other platforms. You should use a file name relative to the current directory. For example, you may create a **File** object using **new File("Welcome.java")** for the file **Welcome.java** in the current directory. You may create a **File** object using **new File("image/us.gif")** for the file **us.gif** under the **image** directory in the current directory. The forward slash (/) is the Java directory separator, which is the same as on Unix. The statement **new File("image/us.gif")** works on Windows, Unix, and any other platform.

relative file name

Java directory separator (/)

[Listing 9.6](#) demonstrates how to create a **File** object and use the methods in the **File** class to obtain its properties. The program creates a **File** object for the file **us.gif**. This file is stored under the **image** directory in the current directory.

LISTING 9.6 TestFileClass.java

```
1 public class TestFileClass {  
2     public static void main(String[] args) {  
3         java.io.File file = new java.io.File("image/us.gif");  
4         System.out.println("Does it exist? " + file.exists());  
5         System.out.println("The file has " + file.length() + " bytes");  
6         System.out.println("Can it be read? " + file.canRead());  
7         System.out.println("Can it be written? " + file.canWrite());  
8         System.out.println("Is it a directory? " + file.isDirectory());  
9         System.out.println("Is it a file? " + file.isFile());  
10        System.out.println("Is it absolute? " + file.isAbsolute());  
11        System.out.println("Is it hidden? " + file.isHidden());  
12        System.out.println("Absolute path is " +  
13            file.getAbsolutePath());  
14        System.out.println("Last modified on " +  
15            new java.util.Date(file.lastModified()));  
16    }  
17 }
```

The **lastModified()** method returns the date and time when the file was last modified, measured in milliseconds since the beginning of Unix time (00:00:00 GMT, January 1, 1970). The **Date** class is used to display it in a readable format in lines 14–15.

FIGURE 9.16 The program creates a **File object and displays file properties.**

The figure consists of two side-by-side screenshots of terminal windows.
 (a) On Windows: A 'Command Prompt' window titled 'Command Prompt'. It shows the command 'java TestFileClass' followed by its output: 'Does it exist? true', 'The file has 2998 bytes', 'Can it be read? true', 'Can it be written? true', 'Is it a directory? false', 'Is it a file? true', 'Is it absolute? false', 'Is it hidden? false', 'Absolute path is C:\book\image\us.gif', and 'Last modified on Tue Nov 02 08:20:45 EST 2004'. The prompt 'C:\book>' is at the bottom.
 (b) On Unix: An 'SSH Secure Shell' window titled 'panda.armstrong.edu - default - SSH Secure Shell'. It shows the command 'java TestFileClass' followed by its output: 'Does it exist? true', 'The file has 2998 bytes', 'Can it be read? true', 'Can it be written? true', 'Is it a directory? false', 'Is it a file? true', 'Is it absolute? false', 'Is it hidden? false', 'Absolute path is /home/daniel/book/image/us.gif', and 'Last modified on Tue Nov 02 08:20:45 EST 2004'. The prompt '[daniel@panda book]\$' is at the bottom. Below the window, it says 'Connected to panda.armstrong.edu' and 'SSH2 - aes128-cbc - hmac-md5'.

(a) On Windows

(b) On Unix

[Figure 9.16\(a\)](#) shows a sample run of the program on Windows, and [Figure 9.16\(b\)](#), a sample run on Unix. As shown in the figures, the path-naming conventions on Windows are different from those on Unix.

9.7 File Input and Output

A [File](#) object encapsulates the properties of a file or a path but does not contain the methods for creating a file or for reading/writing data from/to a file. In order to perform I/O, you need to create objects using appropriate Java I/O classes. The objects contain the methods for reading/writing data from/to a file. This section introduces how to read/write strings and numeric values from/to a text file using the [Scanner](#) and [PrintWriter](#) classes.

9.7.1 Writing Data Using PrintWriter

The [java.io.PrintWriter](#) class can be used to create a file and write data to a text file. First, you have to create a [PrintWriter](#) object for a text file as follows:

```
PrintWriter output = new PrintWriter(filename);
```

Then, you can invoke the [print](#), [println](#), and [printf](#) methods on the [PrintWriter](#) object to write data to a file. [Figure 9.17](#) summarizes frequently used methods in [PrintWriter](#).

FIGURE 9.17 The PrintWriter class contains the methods for writing data to a text file.

java.io.PrintWriter	
+PrintWriter(file: File)	Creates a PrintWriter object for the specified file object.
+PrintWriter(filename: String)	Creates a PrintWriter object for the specified file-name string.
+print(s: String): void	Writes a string to the file.
+print(c: char): void	Writes a character to the file.
+print(cArray: char[]): void	Writes an array of characters to the file.
+print(i: int): void	Writes an int value to the file.
+print(l: long): void	Writes a long value to the file.
+print(f: float): void	Writes a float value to the file.
+print(d: double): void	Writes a double value to the file.
+print(b: boolean): void	Writes a boolean value to the file.
Also contains the overloaded println methods.	A println method acts like a print method; additionally it prints a line separator. The line-separator string is defined by the system. It is \r\n on Windows and \n on Unix.
Also contains the overloaded printf methods.	The printf method was introduced in §3.17, “Formatting Console Output.”

[Listing 9.7](#) gives an example that creates an instance of `PrintWriter` and writes two lines to the file “scores.txt”. Each line consists of first name (a string), middle-name initial (a character), last name (a string), and score (an integer).

LISTING 9.7 WriteData.java

```

1 public class WriteData {
2     public static void main(String[] args) throws Exception {
3         java.io.File file = new java.io.File("scores.txt");
4         if (file.exists()) {
5             System.out.println("File already exists");
6             System.exit(0);
7         }
8         // Create a file
9         java.io.PrintWriter output = new java.io.PrintWriter(file);           create PrintWriter
10
11         // Write formatted output to the file
12         output.print("John T Smith ");
13         output.println(90);
14         output.print("Eric K Jones ");
15         output.println(85);
16
17         // Close the file
18         output.close();
19     }
20 }
21 }
```

Lines 3–7 check whether the file scores.txt exists. If so, exit the program (line 6).

Invoking the constructor of `PrintWriter` will create a new file if the file does not exist. If the file already exists, the current content in the file will be discarded.

create a file

Invoking the constructor of `PrintWriter` may throw an I/O exception. Java forces you to write the code to deal with this type of exception. You will learn how to handle it in

Chapter 13, “Exception Handling.” For now, simply declare `throws Exception` in the method header (line 2).

```
throws Exception
```

You have used the `System.out.print` and `System.out.println` methods to write text to the console. `System.out` is a standard Java object for the console. You can create objects for writing text to any file using `print`, `println`, and `printf` (lines 13–16).

```
print method close file
```

The `close()` method must be used to close the file. If this method is not invoked, the data may not be saved properly in the file.

9.7.2 Reading Data Using Scanner

The `java.util.Scanner` class was used to read strings and primitive values from the console in §2.3, “Reading Input from the Console.” A `Scanner` breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a `Scanner` for `System.in`, as follows:

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

To read from a file, create a `Scanner` for a file, as follows:

```
Scanner input = new Scanner(new File(filename));
```

[Figure 9.18](#) summarizes frequently used methods in `Scanner`.

FIGURE 9.18 The Scanner class contains the methods for scanning data.

java.util.Scanner	
+Scanner(source: File)	Creates a scanner that produces values scanned from the specified file.
+Scanner(source: String)	Creates a scanner that produces values scanned from the specified string.
+close()	Closes this scanner.
+hasNext(): boolean	Returns true if this scanner has more data to be read.
+next(): String	Returns next token as a string from this scanner.
+nextLine(): String	Returns a line ending with the line separator from this scanner.
+nextByte(): byte	Returns next token as a byte from this scanner.
+nextShort(): short	Returns next token as a short from this scanner.
+nextInt(): int	Returns next token as an int from this scanner.
+nextLong(): long	Returns next token as a long from this scanner.
+nextFloat(): float	Returns next token as a float from this scanner.
+nextDouble(): double	Returns next token as a double from this scanner.
+useDelimiter(pattern: String): Scanner	Sets this scanner's delimiting pattern and returns this scanner.

[Listing 9.8](#) gives an example that creates an instance of **Scanner** and reads data from the file “scores.txt”.

LISTING 9.8 ReadData.java

```

1 import java.util.Scanner;
2
3 public class ReadData {
4     public static void main(String[] args) throws Exception {
5         // Create a File instance
6         java.io.File file = new java.io.File("scores.txt");           create a File
7
8         // Create a Scanner for the file
9         Scanner input = new Scanner(file);                            create a Scanner
10
11        // Read data from a file
12        while (input.hasNext()) {                                     scores.txt
13            String firstName = input.next();                         John T Smith 90
14            String mi = input.next();                                Eric K Jones 85
15            String lastName = input.next();                          has next?
16            int score = input.nextInt();                           read items
17            System.out.println(
18                firstName + " " + mi + " " + lastName + " " + score);
19        }
20
21        // Close the file
22        input.close();                                         close file
23    }
24 }
```

Note that `new Scanner(String)` creates a **Scanner** for a given string. To create a **Scanner** to read data from a file, you have to use the `java.io.File` class to create an instance of the `File` using the constructor `new File(filename)` (line 6), and use `new Scanner(File)` to create a **Scanner** for the file (line 9).

File class

Invoking the constructor `new Scanner(File)` may throw an I/O exception. So the `main` method declares `throws Exception` in line 4.

`throws Exception`

Each iteration in the `while` loop reads first name, mi, last name, and score from the text file (lines 12–19). The file is closed in line 22.

It is not necessary to close the input file (line 22), but it is a good practice to do so to release the resources occupied by the file.

`close file`

9.7.3 How Does Scanner Work?

The `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, and `next()` methods are known as *token-reading methods*, because they read tokens separated by delimiters. By default, the delimiters are whitespace. You can use the `useDelimiter-(String regex)` method to set a new pattern for delimiters.

`token-reading method`

`change delimiter`

How does an input method work? A token-reading method first skips any delimiters (whitespace by default), then reads a token ending at a delimiter. The token is then automatically converted into a value of the `byte`, `short`, `int`, `long`, `float`, or `double` type for `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, and `nextDouble()`, respectively. For the `next()` method, no conversion is performed. If the token does not match the expected type, a runtime exception `java.util.InputMismatchException` will be thrown.

`InputMismatchException`

Both methods `next()` and `nextLine()` read a string. The `next()` method reads a string delimited by delimiters, but `nextLine()` reads a line ending with a line separator.

`next() vs. nextLine()`



Note

The line-separator string is defined by the system. It is `\r\n` on Windows and `\n` on Unix. To get the line separator on a particular platform, use

```
String lineSeparator =  
    System.getProperty("line.separator");
```

If you enter input from a keyboard, a line ends with the *Enter* key, which corresponds to the `\n` character.

line separator

The token-reading method does not read the delimiter after the token. If the `nextLine()` is invoked after a token-reading method, the method reads characters that start from this delimiter and end with the line separator. The line separator is read, but it is not part of the string returned by `nextLine()`.

behavior of `nextLine()`

input from file

Suppose a text file named test.txt contains a line

34 567

After the following code is executed,

```
Scanner input = new Scanner(new File("test.txt"));  
int intValue = input.nextInt();  
String line = input.nextLine();
```

`intValue` contains **34** and `line` contains characters **&**, **&**, **&**, **&**, **&**, **&**.

What happens if the input is *entered from the keyboard*? Suppose you enter **34**, the Enter key, **567**, and the Enter key for the following code:

input from keyboard

```
Scanner input = new Scanner(System.in);  
int intValue = input.nextInt();  
String line = input.nextLine();
```

You will get **34** in `intValue` and an empty string in `line`. Why? Here is the reason. The token-reading method `nextInt()` reads in **34** and stops at the delimiter, which in

this case is a line separator (the *Enter* key). The `nextLine()` method ends after reading the line separator and returns the string read before the line separator. Since there are no characters before the line separator, `line` is empty.

9.7.4 Problem: Replacing Text

Suppose you are to write a program named `ReplaceText` that replaces all occurrences of a string in a text file with a new string. The file name and strings are passed as command-line arguments as follows:

```
java ReplaceText sourceFile targetFile oldString  
newString
```

For example, invoking

```
java ReplaceText FormatString.java t.txt StringBuilder  
StringBuffer
```

replaces all the occurrences of `StringBuilder` by `StringBuffer` in `FormatString.java` and saves the new file in `t.txt`.

[Listing 9.9](#) gives the solution to the problem. The program checks the number of arguments passed to the `main` method (lines 7–11), checks whether the source and target files exist (lines 14–25), creates a `Scanner` for the source file (line 28), creates a `PrintWriter` for the target file, and repeatedly reads a line from the source file (line 32), replaces the text (line 33), and writes a new line to the target file (line 34). You must close the output file (line 38) to ensure that data are saved to the file properly.

LISTING 9.9 ReplaceText.java

```

1 import java.io.*;
2 import java.util.*;
3
4 public class ReplaceText {
5     public static void main(String[] args) throws Exception {
6         // Check command-line parameter usage
7         if (args.length != 4) {                                         check command usage
8             System.out.println(
9                 "Usage: java ReplaceText sourceFile targetFile oldStr newStr");
10            System.exit(0);
11        }
12
13        // Check if source file exists
14        File sourceFile = new File(args[0]);
15        if (!sourceFile.exists()) {                                     source file exists?
16            System.out.println("Source file " + args[0] + " does not exist");
17            System.exit(0);
18        }
19
20        // Check if target file exists
21        File targetFile = new File(args[1]);
22        if (targetFile.exists() ) {                                    target file exists?
23            System.out.println("Target file " + args[1] + " already exists");
24            System.exit(0);
25        }
26
27        // Create a Scanner for input and a PrintWriter for output
28        Scanner input = new Scanner(sourceFile);                      create a Scanner
29        PrintWriter output = new PrintWriter(targetFile);           create a PrintWriter
30
31        while (input.hasNext() ) {                                       has next?
32            String s1 = input.nextLine();                                read a line
33            String s2 = s1.replaceAll(args[2], args[3]);
34            output.println(s2);
35        }
36
37        input.close();                                              close file
38        output.close();
39    }
40 }

```

9.8 (GUI) File Dialogs

Java provides the `javax.swing.JFileChooser` class for displaying a file dialog, as shown in [Figure 9.19](#). From this dialog box, the user can choose a file.

[Listing 9.10](#) gives a program that prompts the user to choose a file and displays its contents on the console.

LISTING 9.10 ReadFileUsingJFileChooser.java

```

1 import java.util.Scanner;
2 import javax.swing.JFileChooser;
3
4 public class ReadFileUsingJFileChooser {
5     public static void main(String[] args) throws Exception {
6         JFileChooser fileChooser = new JFileChooser();                  create a JFileChooser
7         if (fileChooser.showOpenDialog(null))                         display file chooser

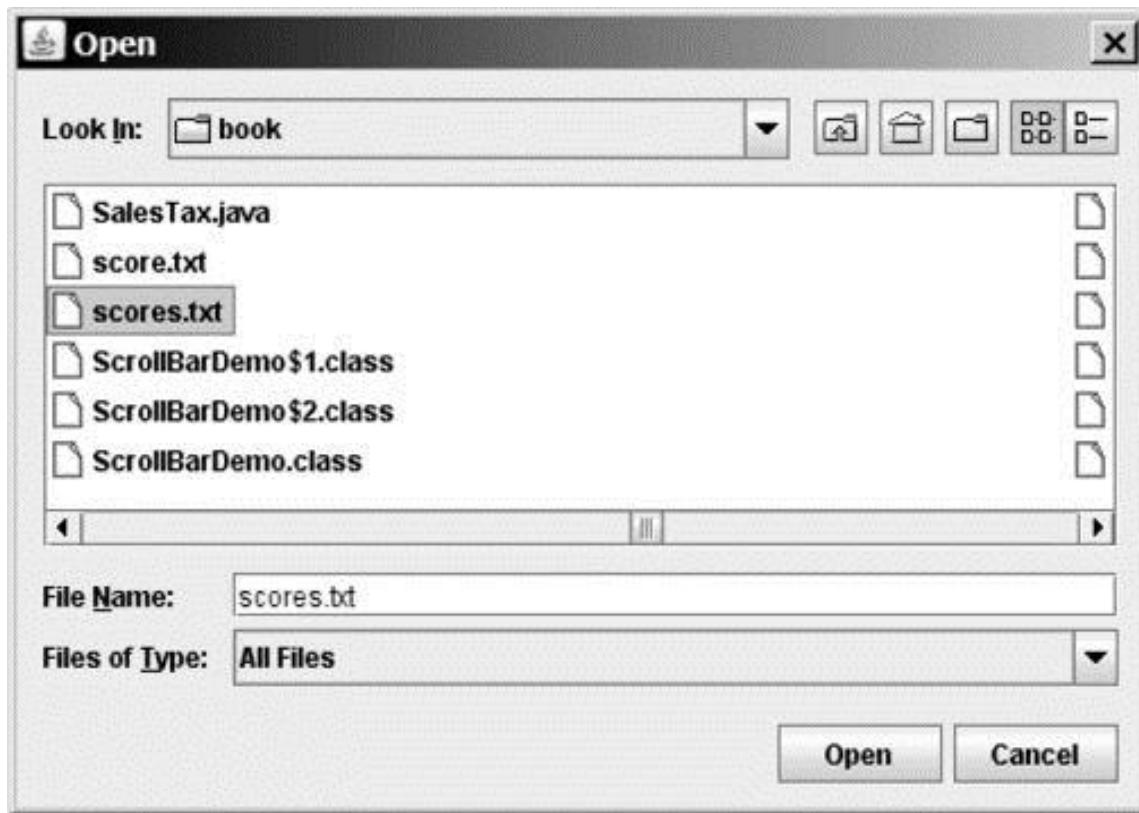
```

```

check status           8      == JFileChooser.APPROVE_OPTION) {
getSelectedFile       9      // Get the selected file
10     java.io.File file = fileChooser.getSelectedFile();
11
12     // Create a Scanner for the file
13     Scanner input = new Scanner(file);
14
15     // Read text from the file
16     while (input.hasNext()) {
17         System.out.println(input.nextLine());
18     }
19
20     // Close the file
21     input.close();
22 }
23 else {
24     System.out.println("No file selected");
25 }
26 }
27 }

```

FIGURE 9.19 `JFileChooser` can be used to display a file dialog for opening a file.



The program creates a `JFileChooser` in line 6. The `showOpenDialog(null)` method displays a dialog box, as shown in [Figure 9.19](#). The method returns an `int` value, either

`APPROVE_OPTION` or `CANCEL_OPTION`, which indicates whether the *Open* button or the *Cancel* button was clicked.

`showOpenDialog`

`APPROVE_OPTION`

The `getSelectedFile()` method (line 10) returns the selected file from the file dialog box. Line 13 creates a scanner for the file. The program continuously reads the lines from the file and displays them to the console (lines 16–18).

`getSelectedFile`

CHAPTER SUMMARY

1. Strings are objects encapsulated in the `String` class. A string can be constructed using one of the 11 constructors or using a string literal shorthand initializer.
2. A `String` object is immutable; its contents cannot be changed. To improve efficiency and save memory, the JVM stores two literal strings that have the same character sequence in a unique object. This unique object is called an interned string object.
3. You can get the length of a string by invoking its `length()` method, retrieve a character at the specified `index` in the string using the `charAt(index)` method, and use the `indexOf` and `lastIndexOf` methods to find a character or a substring in a string.
4. You can use the `concat` method to concatenate two strings, or the plus (+) sign to concatenate two or more strings.
5. You can use the `substring` method to obtain a substring from the string.
6. You can use the `equals` and `compareTo` methods to compare strings. The `equals` method returns `true` if two strings are equal, and `false` if they are not equal. The `compareTo` method returns 0, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.
7. The `Character` class is a wrapper class for a single character. The `Character` class provides useful static methods to determine whether a character is a letter (`isLetter(char)`), a digit (`isDigit(char)`), uppercase (`isUpperCase(char)`), or lowercase (`isLowerCase(char)`).

8. The **StringBuilder/StringBuffer** class can be used to replace the **String** class. The **String** object is immutable, but you can add, insert, or append new contents into a **StringBuilder/StringBuffer** object. Use **String** if the string contents do not require any change, and use **StringBuilder/StringBuffer** if they change.
9. You can pass strings to the **main** method from the command line. Strings passed to the **main** program are stored in **args**, which is an array of strings. The first string is represented by **args[0]**, and **args.length** is the number of strings passed.
10. The **File** class is used to obtain file properties and manipulate files. It does not contain the methods for creating a file or for reading/writing data from/to a file.
11. You can use **Scanner** to read string and primitive data values from a text file and use **PrintWriter** to create a file and write data to a text file.
12. The **JFileChooser** class can be used to display files graphically.

REVIEW QUESTIONS

Section 9.2

9.1 Suppose that **s1**, **s2**, **s3**, and **s4** are four strings, given as follows:

```
String s1 = "Welcome to Java";
String s2 = s1;
String s3 = new String("Welcome to Java");
String s4 = "Welcome to Java";
```

What are the results of the following expressions?

- (1) `s1 == s2`
- (2) `s2 == s3`
- (3) `s1.equals(s2)`
- (4) `s2.equals(s3)`
- (5) `s1.compareTo(s2)`
- (6) `s2.compareTo(s3)`
- (7) `s1 == s4`
- (8) `s1.charAt(0)`

```
(9)    s1.indexOf(&j&)
(10)   s1.indexOf("to")
(11)   s1.lastIndexOf(&a&)
(12)   s1.lastIndexOf("o", 15)
(13)   s1.length()
(14)   s1.substring(5)
(15)   s1.substring(5, 11)
(16)   s1.startsWith("Wel")
(17)   s1.endsWith("Java")
(18)   s1.toLowerCase()
(19)   s1.toUpperCase()
(20)   " Welcome ".trim()
(21)   s1.replace(&o&, &T&)
(22)   s1.replaceAll("o", "T")
(23)   s1.replaceFirst("o", "T")
(24)   s1.toCharArray()
```

To create a string “Welcome to Java”, you may use a statement like this:

```
String s = "Welcome to Java";
```

or

```
String s = new String("Welcome to Java");
```

Which one is better? Why?

9.2 Suppose that **s1** and **s2** are two strings. Which of the following statements or expressions are incorrect?

```
String s = new String("new string");
String s3 = s1 + s2;
String s3 = s1 - s2;
s1 == s2;
s1 >= s2;
s1.compareTo(s2);
```

```
int i = s1.length();
char c = s1(0 );
char c = s1.charAt(s1.length());
```

9.3 What is the printout of the following code?

```
String s1 ="Welcome to Java";
String s2 = s1.replace("o", "abc");
System.out.println(s1);
System.out.println(s2);
```

9.4 Let **s1** be " Welcome " and **s2** be " welcome ". Write the code for the following statements:

- Check whether **s1** is equal to **s2** and assign the result to a Boolean variable **isEqual**.
- Check whether **s1** is equal to **s2**, ignoring case, and assign the result to a Boolean variable **isEqual**.
- Compare **s1** with **s2** and assign the result to an **int** variable **x**.
- Compare **s1** with **s2**, ignoring case, and assign the result to an **int** variable **x**.
- Check whether **s1** has prefix "AAA" and assign the result to a Boolean variable **b**.
- Check whether **s1** has suffix "AAA" and assign the result to a Boolean variable **b**.
- Assign the length of **s1** to an **int** variable **x**.
- Assign the first character of **s1** to a **char** variable **x**.
- Create a new string **s3** that combines **s1** with **s2**.
- Create a substring of **s1** starting from index **1**.
- Create a substring of **s1** from index **1** to index **4**.
- Create a new string **s3** that converts **s1** to lowercase.
- Create a new string **s3** that converts **s1** to uppercase.
- Create a new string **s3** that trims blank spaces on both ends of **s1**.

- Replace all occurrences of character `e` with `E` in `s1` and assign the new string to `s3`.
- Split "`Welcome to Java and HTML`" into an array `tokens` delimited by a space.
- Assign the index of the first occurrence of character `e` in `s1` to an `int` variable `x`.
- Assign the index of the last occurrence of string `abc` in `s1` to an `int` variable `x`.

9.5 Does any method in the `String` class change the contents of the string?

9.6 Suppose string `s` is created using `new String()`; what is `s.length()`?

9.7 How do you convert a `char`, an array of characters, or a number to a string?

9.8 Why does the following code cause a `NullPointerException`?

```

1 public class Test {
2   private String text;
3
4   public Test(String s) {
5     String text = s;
6   }
7
8   public static void main(String[] args) {
9     Test test = new Test("ABC");
10    System.out.println(test.text.toLowerCase());
11  }
12 }
```

9.9 What is wrong in the following program?

```
1 public class Test {  
2     String text;  
3  
4     public void Test(String s) {  
5         this.text = s;  
6     }  
7  
8     public static void main(String[] args) {  
9         Test test = new Test("ABC");  
10        System.out.println(test);  
11    }  
12 }
```

Section 9.3

- 9.10 How do you determine whether a character is in lowercase or uppercase?
- 9.11 How do you determine whether a character is alphanumeric?

Section 9.4

- 9.12 What is the difference between `StringBuilder` and `StringBuffer`?
- 9.13 How do you create a string builder for a string? How do you get the string from a string builder?
- 9.14 Write three statements to reverse a string `s` using the `reverse` method in the `StringBuilder` class.
- 9.15 Write a statement to delete a substring from a string `s` of 20 characters, starting at index 4 and ending with index 10. Use the `delete` method in the `StringBuilder` class.
- 9.16 What is the internal structure of a string and a string builder?
- 9.17 Suppose that `s1` and `s2` are given as follows:
`StringBuilder s1 = new StringBuilder("Java");`
`StringBuilder s2 = new StringBuilder("HTML");`

Show the value of **s1** after each of the following statements. Assume that the statements are independent.

- (1) s1.append(" is fun");
- (2) s1.append(s2);
- (3) s1.insert(2, "is fun");
- (4) s1.insert(1, s2);
- (5) s1.charAt(2);
- (6) s1.length();
- (7) s1.deleteCharAt(3);
- (8) s1.delete(1, 3);
- (9) s1.reverse();
- (10) s1.replace(1, 3, "Computer");
- (11) s1.substring(1, 3);
- (12) s1.substring(2);

9.18 Show the output of the following program:

```
public class Test {  
    public static void main(String[] args) {  
        String s = "Java";  
        StringBuilder builder = new StringBuilder(s);  
        change(s, builder);  
        System.out.println(s);  
        System.out.println(builder);  
    }  
    private static void change(String s, StringBuilder  
builder) {  
        s = s + " and HTML";  
        builder.append(" and HTML");  
    }  
}
```

Section 9.5

9.19 This book declares the **main** method as

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

Can it be replaced by one of the following lines?

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

9.20 Show the output of the following program when invoked using

1. java Test I have a dream
2. java Test "1 2 3"
3. java Test
4. java Test "**"

5. java Test *

```
public class Test {
    public static void main(String[] args) {
        System.out.println("Number of strings is " +
args.length);
        for (int i = 0 ; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

Section 9.6

9.21 What is wrong about creating a `File` object using the following statement?

```
new File("c:\book\test.dat");
```

9.22 How do you check whether a file already exists? How do you delete a file? How do you rename a file? Can you find the file size (the number of bytes) using the `File` class?

9.23 Can you use the `File` class for I/O? Does creating a `File` object create a file on the disk?

Section 9.7

9.24 How do you create a `PrintWriter` to write data to a file? What is the reason to declare `throws Exception` in the main method in [Listing 9.7](#),

WriteData.java? What would happen if the `close()` method were not invoked in [Listing 9.7](#)?

9.25 Show the contents of the file temp.txt after the following program is executed.

```
public class Test {
    public static void main(String[] args) throws
Exception {
    java.io.PrintWriter output = new
        java.io.PrintWriter("temp.txt");
    output.printf("amount is %f %e\r\n", 32.32,
32.32);
    output.printf("amount is %5.4f %5.4e\r\n", 32.32,
32.32);
    output.printf("%6b\r\n", (1 > 2));
    output.printf("%6s\r\n", "Java");
    output.close();
}
}
```

9.26 How do you create a `Scanner` to read data from a file? What is the reason to define `throws Exception` in the main method in [Listing 9.8](#), ReadData.java? What would happen if the `close()` method were not invoked in [Listing 9.8](#)?

9.27 What will happen if you attempt to create a `Scanner` for a nonexistent file? What will happen if you attempt to create a `PrintWriter` for an existing file?

9.28 Is the line separator the same on all platforms? What is the line separator on Windows?

9.29 Suppose you enter **45 57.8 789**, then press the *Enter key*. Show the contents of the variables after the following code is executed.

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

9.30 Suppose you enter **45**, the Enter key, **57.8**, the Enter key, **789**, the Enter key. Show the contents of the variables after the following code is executed.

```
Scanner input = new Scanner(System.in);
int intValue = input.nextInt();
double doubleValue = input.nextDouble();
String line = input.nextLine();
```

PROGRAMMING EXERCISES

Sections 9.2–9.3

9.1* (*Checking SSN*) Write a program that prompts the user to enter a social security number in the format DDD-DD-DDDD, where D is a digit. The program displays "**Valid SSN**" for a correct social security number and "**Invalid SSN**" otherwise.

9.2** (*Checking substrings*) You can check whether a string is a substring of another string by using the `indexOf` method in the `String` class. Write your own method for this function. Write a program that prompts the user to enter two strings, and check whether the first string is a substring of the second.

9.3** (*Checking password*) Some Websites impose certain rules for passwords. Write a method that checks whether a string is a valid password. Suppose the password rule is as follows:

- A password must have at least eight characters.
- A password consists of only letters and digits.
- A password must contain at least two digits.

Write a program that prompts the user to enter a password and displays "**ValidPassword**" if the rule is followed or "**Invalid Password**" otherwise.

9.4 (*Occurrences of a specified character*) Write a method that finds the number of occurrences of a specified character in the string using the following header:

```
public static int count(String str, char a)
```

For example, `count("Welcome", 'e')` returns `2`. Write a test program that prompts the user to enter a string followed by a character and displays the number of occurrences of the character in the string.

9.5** (*Occurrences of each digit in a string*) Write a method that counts the occurrences of each digit in a string using the following header:

```
public static int[] count(String s)
```

The method counts how many times a digit appears in the string. The return value is an array of ten elements, each of which holds the count for a digit. For example, after

executing `int[] counts = count("12203AB3")`, `counts[0]` is 1,
`counts[1]` is 1, `counts[2]` is 2, `counts[3]` is 2.

Write a test program that prompts the user to enter a string and displays the number of occurrences of each digit in the string.

9.6* (*Counting the letters in a string*) Write a method that counts the number of letters in a string using the following header:

```
public static int countLetters(String s)
```

Write a test program that prompts the user to enter a string and displays the number of letters in the string.

9.7* (*Phone keypads*) The international standard letter/number mapping found on the telephone is shown below:

1

2

3

ABC

DEF

4

5

6

GHI

JKL

MNO

7

8

9

PQRS

TUV

WXYZ

0

Write a method that returns a number, given an uppercase letter, as follows:

```
public static int getNumber(char uppercaseLetter)
```

Write a test program that prompts the user to enter a phone number as a string. The input number may contain letters. The program translates a letter (upper- or lowercase) to a digit and leaves all other characters intact. Here is a sample run of the program:



Enter a string: 1-800-**Flowers**

1-800-**Flowers**



Enter a string: 1800**flowers**

18003569377

9.8* (*Binary to decimal*) Write a method that parses a binary number as a string into a decimal integer. The method header is as follows:

```
public static int binaryToDecimal(String  
binaryString)
```

For example, binary string 10001 is $1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2 + 1 = 17$. So, `binaryToDecimal("10001")` returns **17**. Note that

`Integer.parseInt("10001", 2)` parses a binary string to a decimal value.

Do not use this method in this exercise.

Write a test program that prompts the user to enter a binary string and displays the corresponding decimal integer value.

Section 9.4

9.9** (*Binary to hex*) Write a method that parses a binary number into a hex number. The method header is as follows:

```
public static String binaryToHex(String binaryValue)
```

Write a test program that prompts the user to enter a binary number and displays the corresponding hexadecimal value.

9.10** (*Decimal to binary*) Write a method that parses a decimal number into a binary number as a string. The method header is as follows:



Video Note

Number conversion

```
public static String decimalToBinary(int value)
```

Write a test program that prompts the user to enter a decimal integer value and displays the corresponding binary value.

9.11** (*Sorting characters in a string*) Write a method that returns a sorted string using the following header:

```
public static String sort(String s)
```

For example, `sort("acb")` returns `abc`.

Write a test program that prompts the user to enter a string and displays the sorted string.

9.12** (*Anagrams*) Write a method that checks whether two words are anagrams.

Two words are anagrams if they contain the same letters in any order. For example,

`"silent"` and `"listen"` are anagrams. The header of the method is as follows:

```
public static boolean isAnagram(String s1, String s2)
```

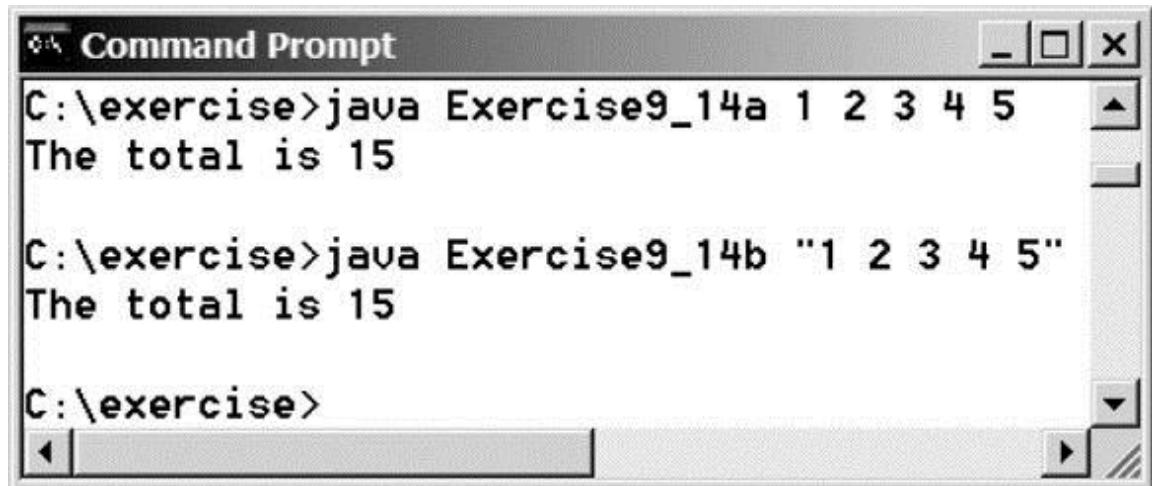
Write a test program that prompts the user to enter two strings and, if they are anagrams, displays `"anagram"`, otherwise displays `"not anagram"`.

Section 9.5

9.13* (*Passing a string to check palindromes*) Rewrite [Listing 9.1](#) by passing the string as a command-line argument.

9.14* (*Summing integers*) Write two programs. The first program passes an unspecified number of integers as separate strings to the `main` method and displays their total. The second program passes an unspecified number of integers delimited by one space in a string to the `main` method and displays their total. Name the two programs [Exercise9_14a](#) and [Exercise9_14b](#), as shown in [Figure 9.20](#).

FIGURE 9.20 The program adds all the numbers passed from the command line.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". It contains two separate command-line sessions. In the first session, the command "java Exercise9_14a 1 2 3 4 5" is run, followed by the output "The total is 15". In the second session, the command "java Exercise9_14b \"1 2 3 4 5\"" is run, also followed by the output "The total is 15". The prompt "C:\exercise>" is visible at the bottom of the window.

9.15* (*Finding the number of uppercase letters in a string*) Write a program that passes a string to the `main` method and displays the number of uppercase letters in a string.

Sections 9.7–9.8

9.16** (*Reformatting Java source code*) Write a program that converts the Java source code from the next-line brace style to the end-of-line brace style. For example, the Java source in (a) below uses the next-line brace style. Your program converts it to the end-of-line brace style in (b).

```
public class Test
{
    public static void main(String[] args)
    {
        // Some statements
    }
}
```

(a) Next-line brace style

```
public class Test {
    public static void main(String[] args) {
        // Some statements
    }
}
```

(b) End-of-line brace style

Your program can be invoked from the command line with the Java source-code file as the argument. It converts the Java source code to a new format. For example, the following command converts the Java source-code file **Test.java** to the end-of-line brace style.

```
java Exercise9_16 Test.java
```

9.17* (*Counting characters, words, and lines in a file*) Write a program that will count the number of characters (excluding control characters `\r` and `\n`), words, and lines, in a file. Words are separated by spaces, tabs, carriage return, or line-feed characters. The file name should be passed as a command-line argument, as shown in [Figure 9.21](#).

FIGURE 9.21 The program adds all the numbers passed from the command line.



The screenshot shows a Windows Command Prompt window titled "Command Prompt". The window contains the following text output:

```
C:\exercise>java Exercise9_17 Loan.java
File Loan.java has
1777 characters
210 words
71 lines
```

The command `java Exercise9_17 Loan.java` was run, and the program output the character count (1777), word count (210), and line count (71) of the file `Loan.java`.

9.18* (*Processing scores in a text file*) Suppose that a text file **Exercise9_18.txt** contains an unspecified number of scores. Write a program that reads the scores from the file and displays their total and average. Scores are separated by blanks.

9.19* (*Writing/Reading data*) Write a program to create a file named **Exercise9_19.txt** if it does not exist. Write 100 integers created randomly into the file using text I/O. Integers are separated by spaces in the file. Read the data back from the file and display the sorted data.

9.20** (*Replacing text*) [Listing 9.9](#), ReplaceText.java, gives a program that replaces text in a source file and saves the change into a new file. Revise the program to save the change into the original file. For example, invoking

```
java Exercise9_20 file oldString newString
```

replaces **oldString** in the source file with **newString**

9.21** (*Removing text*) Write a program that removes all the occurrences of a specified string from a text file. For example, invoking

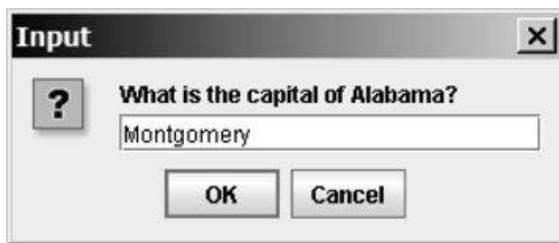
```
java Exercise9_21 John filename
```

removes string **John** from the specified file.

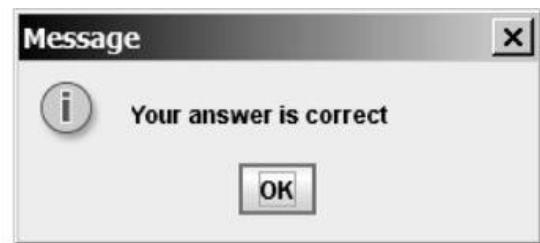
Comprehensive

9.22** (*Guessing the capitals*) Write a program that repeatedly prompts the user to enter a capital for a state, as shown in [Figure 9.22\(a\)](#). Upon receiving the user input, the program reports whether the answer is correct, as shown in [Figure 9.22\(b\)](#). Assume that 50 states and their capitals are stored in a two-dimensional array, as shown in [Figure 9.23](#). The program prompts the user to answer all ten states' capitals and displays the total correct count.

FIGURE 9.22 The program prompts the user to enter the capital in (a) and reports the correctness of the answer.



(a)



(b)

FIGURE 9.23 A two-dimensional array stores states and their capitals.

Alabama	Montgomery
Alaska	Juneau
Arizona	Phoenix
...	...
...	...

9.23** (*Implementing the **String** class*) The **String** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyString1**):

```
public MyString1(char[] chars);
public char charAt(int index);
public int length();
public MyString1 substring(int begin, int end);
public MyString1 toLowerCase();
public boolean equals(MyString1 s);
public static MyString1 valueOf(int i);
```

9.24** (*Implementing the **String** class*) The **String** class is provided in the Java library. Provide your own implementation for the following methods(name the new class **MyString2**):

```
public MyString2(String s);
public int compare(String s);
public MyString2 substring(int begin);
public MyString2 toUpperCase();
public char[] toChars();
public static MyString2 valueOf(boolean b);
```

9.25 (*Implementing the **Character** class*) The **Character** class is provided in the Java library. Provide your own implementation for this class. Name the new class **MyCharacter**.

9.26** (*Implementing the **StringBuilder** class*) The **StringBuilder** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyStringBuilder1**):

```
public MyStringBuilder1(String s);
```

```

public MyStringBuilder1 append(MyStringBuilder1 s);
public MyStringBuilder1 append(int i);
public int length();
public char charAt(int index);
public MyStringBuilder1 toLowerCase();
public MyStringBuilder1 substring(int
public begin, int end);
public String toString();

```

9.27** (*Implementing the **StringBuilder** class*) The **StringBuilder** class is provided in the Java library. Provide your own implementation for the following methods (name the new class **MyStringBuilder2**):

```

public MyStringBuilder2();
public MyStringBuilder2(char[] chars);
public MyStringBuilder2(String s);
public MyStringBuilder2 insert(int offset,
MyStringBuilder2 s);
public MyStringBuilder2 reverse();
public MyStringBuilder2 substring(int begin);
public MyStringBuilder2 toUpperCase();

```

9.28* (*Common prefix*) Write a method that returns the common prefix of two strings. For example, the common prefix of "**distance**" and "**disinfection**" is "**dis**". The header of the method is as follows:

```
public static String prefix(String s1, String s2)
```

If the two strings have no common prefix, the method returns an empty string.

Write a **main** method that prompts the user to enter two strings and display their common prefix.

9.29** (*New string **split** method*) The **split** method in the **String** class returns an array of strings consisting of the substrings split by the delimiters. However, the delimiters are not returned. Implement the following new method that returns an array of strings consisting of the substrings split by the matches, including the matches.

```
public static String[] split(String s, String regex)
```

For example, **split("ab#12#453", "#")** returns **ab, #, 12, #, 453** in an array of **String**, and **split("a?b?gf#e", "[?#]")** returns **a, b, ?, b, gf, #, e** in an array of **String**.

9.30** (*Financial: credit card number validation*) Rewrite Exercise 5.31 using a string input for credit card number. Redesign the program using the following method:

```

/** Return true if the card number is valid */
public static boolean isValid(String cardNumber)
/** Get the result from Step 2 */
public static int sumOfDoubleEvenPlace(String
cardNumber)
/** Return this number if it is a single digit;
otherwise,
    *return the sum of the two digits */
public static int getDigit(int number)
/** Return sum of odd place digits in number */
public static int sumOfOddPlace(String cardNumber)

```

9.31*** (*Game: hangman*) Write a hangman game that randomly generates a word and prompts the user to guess one letter at a time, as shown in the sample run. Each letter in the word is displayed as an asterisk. When the user makes a correct guess, the actual letter is then displayed. When the user finishes a word, display the number of misses and ask the user whether to continue for another word. Declare an array to store words, as follows:

```
// Use any words you wish
String[] words = {"write", "that", ...};
```



```

(Guess) Enter a letter in word ***** > p
(Guess) Enter a letter in word p***** > r
(Guess) Enter a letter in word pr**r** > p
    p is already in the word
(Guess) Enter a letter in word pr**r** > o
(Guess) Enter a letter in word pro*r** > g
(Guess) Enter a letter in word program** > n
    n is not in the word
(Guess) Enter a letter in word program** > m
(Guess) Enter a letter in word program*m > a
The word is program. You missed 1 time
Do you want to guess for another word? Enter y or n>

```

9.32** (*Checking ISBN*) Use string operations to simplify Exercise 3.9. Enter the first 9digits of an ISBN number as a string.

9.33*** (*Game: hangman*) Rewrite Exercise 9.31. The program reads the words stored in a text file named **Exercise9_33.txt**. Words are delimited by spaces.

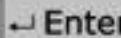
9.34** (*Replacing text*) Revise Exercise 9_20 to place a string in a file with a new string for all files in the specified directory using the following command:

```
java Exercise9_34 dir oldString newList
```

9.35* (*Bioinformatics: finding genes*) Biologists use a sequence of letters **A**, **C**, **T**, and **G** to model a genome. A gene is a substring of a genome that starts after a triplet **ATG** and ends before a triplet **TAG**, **TAA**, or **TGA**. Furthermore, the length of a gene string is a multiple of 3 and the gene does not contain any of the triplets **ATG**, **TAG**, **TAA**, and **TGA**. Write a program that prompts the user to enter a genome and displays all genes in the genome. If no gene is found in the input sequence, displays no gene. Here are the sample runs:



Enter a genome string: TTATTTTAAGGATGGGGCGTTAGTT

 Enter

TTT

GGGCGT



Enter a genome string: TGTGTGTATAT

 Enter

no gene is found

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 301).

<vbk:9781256335153#outline(11)>

APPENDIX A Java Keywords

The following fifty keywords are reserved for use by the Java language:

abstract

assert

boolean

break

byte

case

catch

char

class

const

continue

default

do

double

else

enum

extends

for

final

finally

float

goto

if
implements
import
instanceof
int
interface
long
native
new
package
private
protected
public
return
short
static
strictfp*
super
switch
synchronized
this
throw
throws
transient
try

```
void  
volatile  
while
```

The keywords **goto** and **const** are C++ keywords reserved, but not currently used, in Java. This enables Java compilers to identify them and to produce better error messages if they appear in Java programs.

The literal values **true**, **false**, and **null** are not keywords, just like literal value **100**. However, you cannot use them as identifiers, just as you cannot use **100** as an identifier.

assert is a keyword added in JDK 1.4 and **enum** is a keyword added in JDK 1.5.

*The **strictfp** keyword is a modifier for method or class to use strict floating-point calculations. Floating-point arithmetic can be executed in one of two modes: *strict* or *nonstrict*. The strict mode guarantees that the evaluation result is the same on all Java Virtual Machine implementations. The nonstrict mode allows intermediate results from calculations to be stored in an extended format different from the standard IEEE floating-point number format. The extended format is machine-dependent and enables code to be executed faster. However, when you execute the code using the nonstrict mode on different JVMs, you may not always get precisely the same results. By default, the nonstrict mode is used for floating-point calculations. To use the strict mode in a method or a class, add the **strictfp** keyword in the method or the class declaration. Strict floating-point may give you slightly better precision than nonstrict floating-point, but the distinction will only affect some applications. Strictness is not inherited; that is, the presence of **strictfp** on a class or interface declaration does not cause extended classes or interfaces to be strict.

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 1311).
<vbk:9781256335153#outline(12)>

APPENDIX B The ASCII Character Set

[Tables B.1](#) and [B.2](#) show ASCII characters and their respective decimal and hexadecimal codes. The decimal or hexadecimal code of a character is a combination of its row index and column index. For example, in [Table B.1](#), the letter A is at row 6 and column 5, so its decimal equivalent is 65; in [Table B.2](#), letter A is at row 4 and column 1, so its hexadecimal equivalent is 41.

TABLE B.1 ASCII Character Set in the Decimal Index

	<i>0</i>	<i>I</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	"	#	\$	%	&	,
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	-	·	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	del		

TABLE B.2 ASCII Character Set in the Hexadecimal Index

	<i>0</i>	<i>I</i>	<i>2</i>	<i>3</i>	<i>4</i>	<i>5</i>	<i>6</i>	<i>7</i>	<i>8</i>	<i>9</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>	<i>F</i>
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	ff	cr	so	si
1	dle	dcl	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	"	#	\$	%	&	,	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	-
6	'	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	del

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 1312).
<vbk:9781256335153#outline(13)>

APPENDIX C Operator Precedence Chart

The operators are shown in decreasing order of precedence from top to bottom. Operators in the same group have the same precedence, and their associativity is shown in the table.

Operator

Name

Associativity

()

Parentheses

Left to right

()

Function call

Left to right

[]

Array subscript

Left to right

.

Object member access

Left to right

++

Postincrement

Right to left

--

Postdecrement

Right to left

++

Preincrement

Right to left

--

Predecrement

Right to left

+

Unary plus

Right to left

-

Unary minus

Right to left

!

Unary logical negation

Right to left

(**type**)

Unary casting

Right to left

new

Creating object

Right to left

*

Multiplication

Left to right

/

Division

Left to right

%

Remainder

Left to right

+

Addition

Left to right

-

Subtraction

Left to right

<<

Left shift

Left to right

>>

Right shift with sign extension

Left to right

>>>

Right shift with zero extension

Left to right

<

Less than

Left to right

<=

Less than or equal to

Left to right

>

Greater than

Left to right

>=

Greater than or equal to

Left to right

`instanceof`

Checking object type

Left to right

`==`

`!=`

Equal comparison

Not equal

Left to right

Left to right

`&`

(Unconditional AND)

Left to right

`^`

(Exclusive OR)

Left to right

`|`

(Unconditional OR)

Left to right

`&&`

Conditional AND

Left to right

`||`

Conditional OR

Left to right

`? :`

Ternary condition

Right to left

`=`

Assignment

Right to left

`+ =`

Addition assignment

Right to left

`- =`

Subtraction assignment

Right to left

`* =`

Multiplication assignment

Right to left

`/ =`

Division assignment

Right to left

`% =`

Remainder assignment

Right to left

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 1314).
<vbk:9781256335153#outline(14)>

APPENDIX D Java Modifiers

Modifiers are used on classes and class members (constructors, methods, data, and class-level blocks), but the final modifier can also be used on local variables in a method. A modifier that can be applied to a class is called a *class modifier*. A modifier that can be applied to a method is called a *method modifier*. A modifier that can be applied to a data field is called a *data modifier*. A modifier that can be applied to a class-level block is called a block modifier. The following table gives a summary of the Java modifiers.

Modifier	class	constructor	method	data	block	Explanation
(default)*	✓	✓	✓	✓	✓	A class, constructor, method, or data field is visible in this package.
public	✓	✓	✓	✓		A class, constructor, method, or data field is visible to all the programs in any package.
private		✓	✓	✓		A constructor, method or data field is only visible in this class.
protected		✓	✓	✓		A constructor, method or data field is visible in this package and in subclasses of this class in any package.
static			✓	✓	✓	Define a class method, or a class data field or a static initialization block.
final	✓		✓	✓		A final class cannot be extended. A final method cannot be modified in a subclass. A final data field is a constant.
abstract	✓			✓		An abstract class must be extended. An abstract method must be implemented in a concrete subclass.
native				✓		A native method indicates that the method is implemented using a language other than Java.

* Default access has no modifier associated with it. For example: **class Test {}**

Modifier	class	constructor	method	data	block	Explanation
synchronized			✓		✓	Only one thread at a time can execute this method.
strictfp		✓		✓		Use strict floating-point calculations to guarantee that the evaluation result is the same on all JVMs.
transient					✓	Mark a nonserializable instance data field.

APPENDIX E Special Floating-Point Values

Dividing an integer by zero is invalid and throws [`ArithmaticException`](#), but dividing a floating-point value by zero does not cause an exception. Floating-point arithmetic can overflow to infinity if the result of the operation is too large for a [`double`](#) or a [`float`](#), or underflow to zero if the result is too small for a double or a [`float`](#). Java provides the special floating-point values [`POSITIVE_INFINITY`](#), [`NEGATIVE_INFINITY`](#), and [`NaN`](#) (Not a Number) to denote these results. These values are defined as special constants in the [`Float`](#) class and the Double class.

If a positive floating-point number is divided by zero, the result is [`POSITIVE_INFINITY`](#). If a negative floating-point number is divided by zero, the result is [`NEGATIVE_INFINITY`](#). If a floating-point zero is divided by zero, the result is NaN, which means that the result is undefined mathematically. The string representation of these three values are Infinity, -Infinity, and NaN. For example,

```
System.out.print(1.0 / 0); // Print Infinity
System.out.print(-1.0 / 0); // Print -Infinity
System.out.print(0.0 / 0); // Print NaN
```

These special values can also be used as operands in computations. For example, a number divided by `POSITIVE_INFINITY` yields a positive zero. [Table E.1](#) summarizes various combinations of the `/`, `*`, `%`, `+`, and `-` operators.

TABLE E.1 Special Floating-Point Values

x	y	x/y	$x*y$	$x \% y$	$x + y$	$x - y$
<code>Finite</code>	<code>±0.0</code>	$\pm \infty$	<code>±0.0</code>	<code>NaN</code>	<code>Finite</code>	<code>Finite</code>
<code>Finite</code>	$\pm \infty$	<code>±0.0</code>	<code>±0.0</code>	<code>x</code>	$\pm \infty$	∞
<code>±0.0</code>	<code>±0.0</code>	<code>NaN</code>	<code>±0.0</code>	<code>NaN</code>	<code>±0.0</code>	<code>±0.0</code>
$\pm \infty$	<code>Finite</code>	$\pm \infty$	<code>±0.0</code>	<code>NaN</code>	$\pm \infty$	$\pm \infty$
$\pm \infty$	$\pm \infty$	<code>NaN</code>	<code>±0.0</code>	<code>NaN</code>	$\pm \infty$	∞
<code>±0.0</code>	$\pm \infty$	<code>±0.0</code>	<code>NaN</code>	<code>±0.0</code>	$\pm \infty$	<code>±0.0</code>
<code>NaN</code>	<code>Any</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>
<code>Any</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>



Note

If one of the operands is `NaN`, the result is `NaN`.

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 1318).
 <vbk:9781256335153#outline(16.1)>

APPENDIX F Number Systems

Introduction

Computers use binary numbers internally, because computers are made naturally to store and process 0s and 1s. The binary number system has two digits, 0 and 1. A number or character is stored as a sequence of 0s and 1s. Each 0 or 1 is called a *bit* (binary digit).

binary numbers

In our daily life we use decimal numbers. When we write a number such as 20 in a program, it is assumed to be a decimal number. Internally, computer software is used to convert decimal numbers into binary numbers, and vice versa.

decimal numbers

We write computer programs using decimal numbers. However, to deal with an operating system, we need to reach down to the “machine level” by using binary numbers. Binary numbers tend to be very long and cumbersome. Often hexadecimal numbers are used to abbreviate them, with each hexadecimal digit representing four binary digits. The hexadecimal number system has 16 digits: 0–9, A–F. The letters A, B, C, D, E, and F correspond to the decimal numbers 10, 11, 12, 13, 14, and 15.

hexadecimal number

The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. A decimal number is represented by a sequence of one or more of these digits. The value that each digit represents depends on its position, which denotes an integral power of 10. For example, the digits 7, 4, 2, and 3 in decimal number 7423 represent 7000, 400, 20, and 3, respectively, as shown below:

$$\boxed{7 \quad 4 \quad 2 \quad 3} = 7 \times 10^3 + 4 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$$

$$10^3 \ 10^2 \ 10^1 \ 10^0 = 7000 + 400 + 20 + 3 = 7423$$

The decimal number system has ten digits, and the position values are integral powers of 10. We say that 10 is the *base* or *radix* of the decimal number system. Similarly, since the binary number system has two digits, its base is 2, and since the hex number system has 16 digits, its base is 16.

base

radix

If 1101 is a binary number, the digits 1, 1, 0, and 1 represent 1×2^3 , 1×2^2 , 0×2^1 , and 1×2^0 , respectively:

$$\boxed{1 \ 1 \ 0 \ 1} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

$$2^3 \ 2^2 \ 2^1 \ 2^0 = 8 + 4 + 0 + 1 = 13$$

If 7423 is a hex number, the digits 7, 4, 2, and 3 represent 7×16^3 , 4×16^2 , 2×16^1 , and 3×16^0 , respectively:

$$\boxed{7 \ 4 \ 2 \ 3} = 7 \times 16^3 + 4 \times 16^2 + 2 \times 16^1 + 3 \times 16^0$$

$$16^3 \ 16^2 \ 16^1 \ 16^0 = 28672 + 1024 + 32 + 3 = 29731$$

2 Conversions Between Binary and Decimal Numbers

Given a binary number the equivalent decimal value is

$$b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \dots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

Here are some examples of converting binary numbers to decimals:

Binary	Conversion Formula	Decimal
10	$1 \times 2^1 + 0 \times 2^0$	2
1000	$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$	8
10101011	$1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$	171

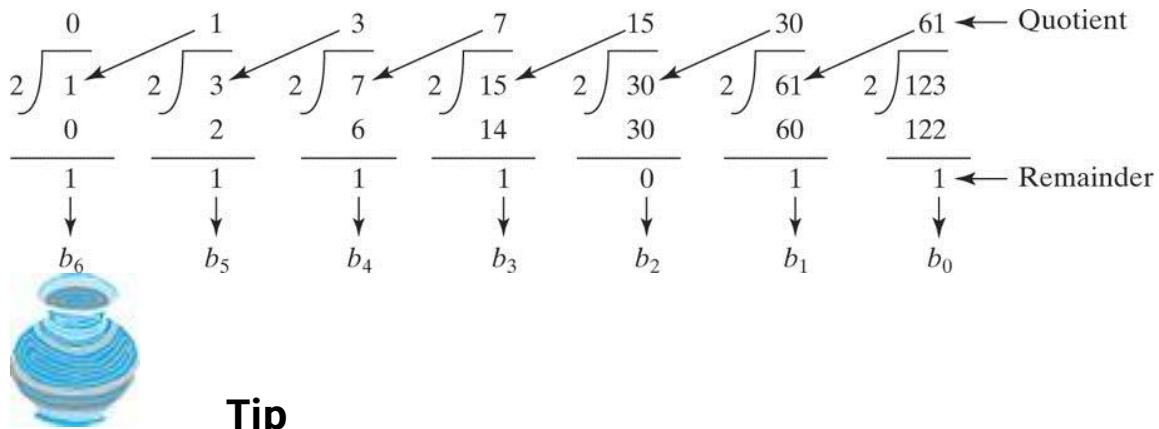
To convert a decimal number d to a binary number is to find the bits $b_n, b_{n-1}, b_{n-2}, \dots, b_2, b_1$, and b_0 such that

decimal to binary

$$d = b_n \times 2^n + b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \dots + b_2 \times 2^2 + b_1 \times 2^1 + b_0 \times 2^0$$

These bits can be found by successively dividing d by 2 until the quotient is 0. The remainders are $b_0, b_1, b_2, \dots, b_{n-1}$, and b_n .

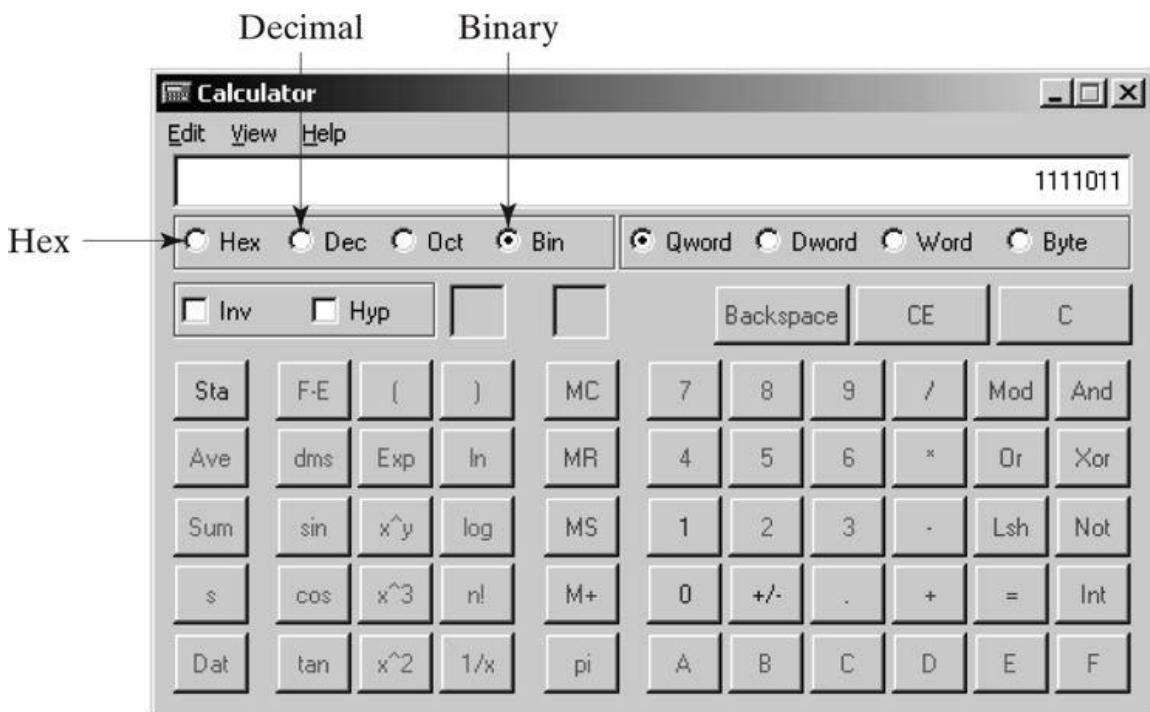
For example, the decimal number 123 is 1111011 in binary. The conversion is done as follows:



Tip

The Windows Calculator, as shown in [Figure F.1](#), is a useful tool for performing number conversions. To run it, choose *Programs*, *Accessories*, and *Calculator* from the *Start* button, then under *View* select *Scientific*.

FIGURE F.1 You can perform number conversions using the Windows Calculator.



3 Conversions Between Hexadecimal and Decimal Numbers

Given a hexadecimal number $h_nh_{n-1}h_{n-2} \dots h_2h_1h_0$, the equivalent decimal value is

$$h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

Here are some examples of converting hexadecimal numbers to decimals:

hex to decimal

Hexadecimal

Conversion Formula

Decimal

7F

$$7 \times 16^1 + 15 \times 16^0$$

127

FFFF

$$15 \times 16^3 + 15 \times 16^2 + 15 \times 16^1 + 15 \times 16^0$$

65535

431

$$4 \times 16^2 + 3 \times 16^1 + 1 \times 16^0$$

1073

To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits $h_n, h_{n-1}, h_{n-2}, \dots, h_2, h_1$ and h_0 such that

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

These numbers can be found by successively dividing d by 16 until the quotient is 0. The remainders are $h_0, h_1, h_2, \dots, h_{n-2}, h_{n-1}$, and h_n .

decimal to hex

For example, the decimal number 123 is 7B in hexadecimal. The conversion is done as follows:

$$\begin{array}{r}
 & 0 & 7 & \xleftarrow{\hspace{1cm}} \text{Quotient} \\
 16 \sqrt{7} & \xleftarrow{\hspace{1cm}} & 16 \sqrt{123} & \\
 & 0 & 112 & \\
 \hline
 & 7 & 11 & \xleftarrow{\hspace{1cm}} \text{Remainder} \\
 & \downarrow & \downarrow & \\
 h_1 & & h_0 &
 \end{array}$$

4 Conversions Between Binary and Hexadecimal Numbers

To convert a hexadecimal to a binary number, simply convert each digit in the hexadecimal number into a four-digit binary number, using Table F.1.

hex to binary

For example, the hexadecimal number 7B is 1111011, where 7 is 111 in binary, and B is 1011 in binary.

To convert a binary number to a hexadecimal, convert every four binary digits from right to left in the binary number into a hexadecimal number.

binary to hex

For example, the binary number 1110001101 is 38D, since 1101 is D, 1000 is 8, and 11 is 3, as shown below.

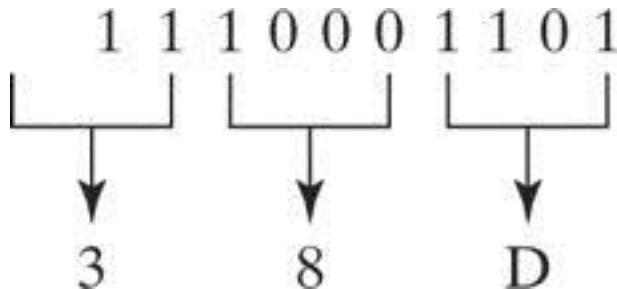


TABLE F.1 Converting Hexadecimal to Binary

<i>Hexadecimal</i>	<i>Binary</i>	<i>Decimal</i>
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

(Note: Octal numbers are also useful. The octal number system has eight digits, 0 to 7. A decimal number 8 is represented in the octal system as 10.)

REVIEW QUESTIONS

1. Convert the following decimal numbers into hexadecimal and binary numbers.

100; 4340; 2000

2. Convert the following binary numbers into hexadecimal and decimal numbers.

1000011001; 100000000; 100111

3. Convert the following hexadecimal numbers into binary and decimal numbers.

FEFA9; 93; 2000

(*Introduction to Java Programming for Sheridan College*. Pearson Learning Solutions p. 1322).

<vbk:9781256335153#outline(17.4)>

Frequently Used Static Constants/Methods

```
Math.PI  
Math.random()  
Math.pow(a, b)  
System.currentTimeMillis()  
System.out.println(anyValue)  
JOptionPane.showMessageDialog(null,  
    message)  
JOptionPane.showInputDialog(  
    prompt-message)  
Integer.parseInt(string)  
Double.parseDouble(string)  
Arrays.sort(type[] list)  
Arrays.binarySearch(type[] list, type key)
```

Array/Length/Initializer

```
int[] list = new int[10];  
list.length;  
int[] list = {1, 2, 3, 4};
```

Multidimensional Array/Length/Initializer

```
int[][] list = new int[10][10];  
list.length;  
list[0].length;  
int[][] list = {{1, 2}, {3, 4}};
```

Ragged Array

```
int[][] m = {{1, 2, 3, 4},  
             {1, 2, 3},  
             {1, 2},  
             {1}};
```

Text File Output

```
PrintWriter output =  
    new PrintWriter(filename);  
output.print(...);  
output.println(...);  
output.printf(...);
```

Text File Input

```
Scanner input = new Scanner(  
    new File(filename));
```

File Class

```
File file =  
    new File(filename);  
file.exists()  
file.renameTo(File)  
file.delete()
```

Object Class

```
Object o = new Object();  
o.toString();  
o.equals(o1);
```

Comparable Interface

```
c.compareTo(Comparable)  
c is a Comparable object
```

String Class

```
String s = "Welcome";  
String s = new String(char[]);  
int length = s.length();  
char ch = s.charAt(index);  
int d = s.compareTo(s1);  
boolean b = s.equals(s1);  
boolean b = s.startsWith(s1);  
boolean b = s.endsWith(s1);  
String s1 = s.trim();  
String s1 = s.toUpperCase();  
String s1 = s.toLowerCase();  
int index = s.indexOf(ch);  
int index = s.lastIndexOf(ch);  
String s1 = s.substring(ch);  
String s1 = s.substring(i,j);  
char[] chs = s.toCharArray();  
String s1 = s.replaceAll(regex,repl);  
String[] tokens = s.split(regex);
```

ArrayList Class

```
ArrayList<E> list = new ArrayList<E>();  
list.add(object);  
list.add(index, object);  
list.clear();  
Object o = list.get(index);  
boolean b = list.isEmpty();  
boolean b = list.contains(object);  
int i = list.size();  
list.remove(index);  
list.set(index, object);  
int i = list.indexOf(object);  
int i = list.lastIndexOf(object);
```

printf Method

```
System.out.printf("%b %c %d %f %e %s",  
    true, 'A', 45, 45.5, 45.5, "Welcome");  
System.out.printf("%-5d %10.2f %10.2e %8s",  
    45, 45.5, 45.5, "Welcome");
```

