

INTRODUCTION TO COMPUTERS, PROGRAMS, AND JAVA

Objectives

- To understand computer basics, programs, and operating systems (§§1.2–1.4).
- To describe the relationship between Java and the World Wide Web (§1.5).
- To understand the meaning of Java language specification, API, JDK, and IDE (§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).
- To display output using the **JOptionPane** message dialog boxes (§1.9).
- To become familiar with Java programming style and documentation (§1.10).
- To explain the differences between syntax errors, runtime errors, and logic errors (§1.11).



1.1 Introduction



what is programming?
programming
program

The central theme of this book is to learn how to solve problems by writing a program.

This book is about programming. So, what is programming? The term *programming* means to create (or develop) software, which is also called a *program*. In basic terms, software contains the instructions that tell a computer—or a computerized device—what to do.

Software is all around you, even in devices that you might not think would need it. Of course, you expect to find and use software on a personal computer, but software also plays a role in running airplanes, cars, cell phones, and even toasters. On a personal computer, you use word processors to write documents, Web browsers to explore the Internet, and e-mail programs to send messages. These programs are all examples of software. Software developers create software with the help of powerful tools called *programming languages*.

This book teaches you how to create programs by using the Java programming language. There are many programming languages, some of which are decades old. Each language was invented for a specific purpose—to build on the strengths of a previous language, for example, or to give the programmer a new and unique set of tools. Knowing that there are so many programming languages available, it would be natural for you to wonder which one is best. But, in truth, there is no “best” language. Each one has its own strengths and weaknesses. Experienced programmers know that one language might work well in some situations, whereas a different language may be more appropriate in other situations. For this reason, seasoned programmers try to master as many different programming languages as they can, giving them access to a vast arsenal of software-development tools.

If you learn to program using one language, you should find it easy to pick up other languages. The key is to learn how to solve problems using a programming approach. That is the main theme of this book.

You are about to begin an exciting journey: learning how to program. At the outset, it is helpful to review computer basics, programs, and operating systems. If you are already familiar with such terms as CPU, memory, disks, operating systems, and programming languages, you may skip the review in Sections 1.2–1.4.

1.2 What Is a Computer?



hardware
software

A computer is an electronic device that stores and processes data.

A computer 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. Knowing computer hardware isn’t essential to learning a programming language, but it can help you better understand the effects that a program’s instructions have on the computer and its components. This section introduces computer hardware components and their functions.

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

- A central processing unit (CPU)
- Memory (main memory)
- Storage devices (such as disks and CDs)
- Input devices (such as the mouse and keyboard)
- Output devices (such as monitors and printers)
- Communication devices (such as modems and network interface cards)

bus

A computer’s components are interconnected by a subsystem called a *bus*. You can think of a bus as a sort of system of roads running among the computer’s components; data and

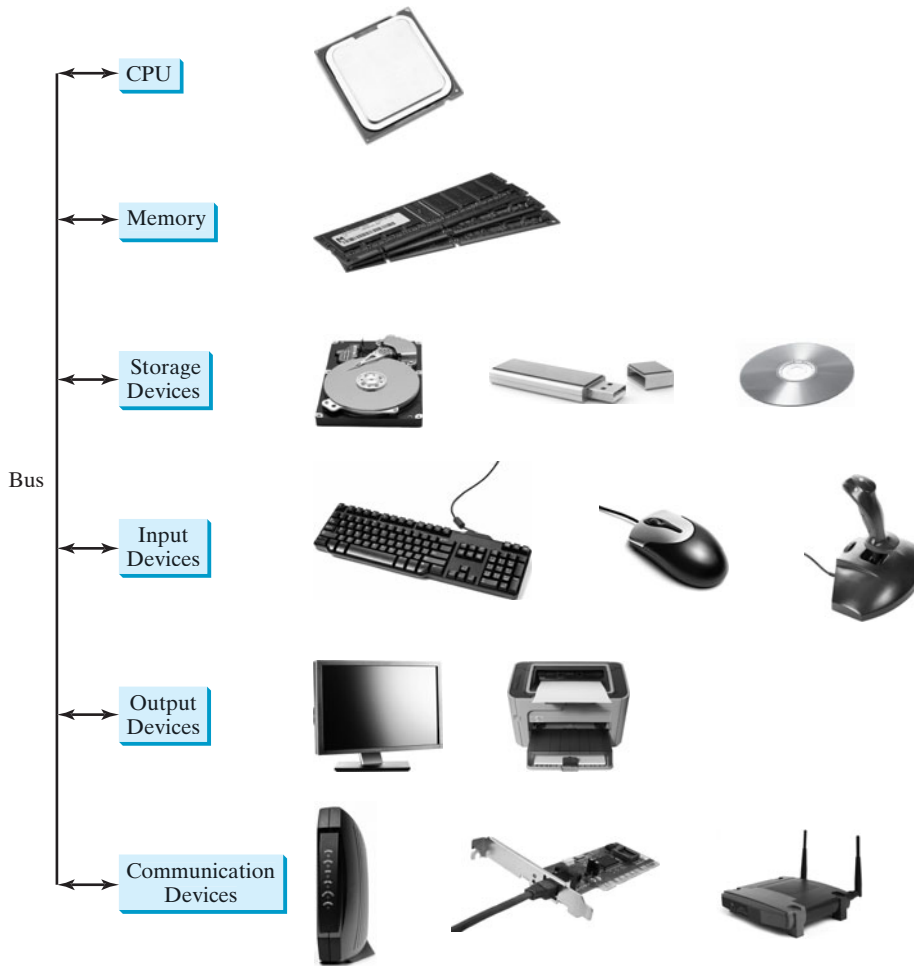


FIGURE 1.1 A computer consists of a CPU, memory, storage devices, input devices, output devices, and communication devices.

power travel along the bus from one part of the computer to another. In personal computers, the bus is built into the computer's *motherboard*, which is a circuit case that connects all of the parts of a computer together, as shown in Figure 1.2. motherboard

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 (addition, subtraction, multiplication, division) and logical operations (comparisons). CPU

Today's CPUs are built on small silicon semiconductor chips that contain millions of tiny electric switches, called *transistors*, for processing information.

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. A higher clock *speed* enables more instructions to be 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. In the 1990s computers measured clocked speed in *megahertz (MHz)*, but CPU speed has been improving continuously, speed hertz megahertz

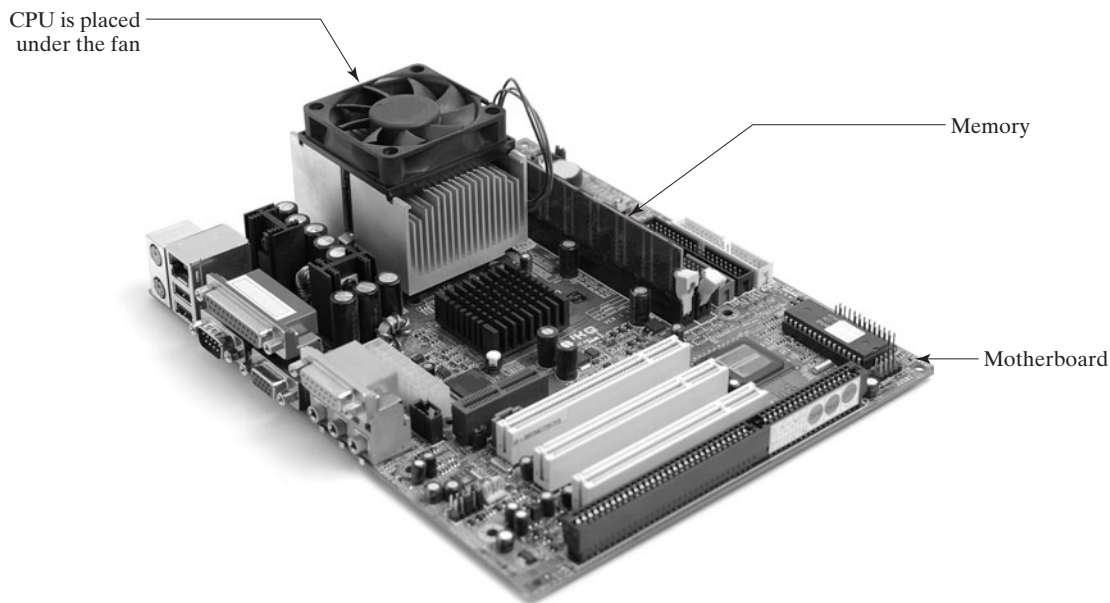


FIGURE 1.2 The motherboard connects all parts of a computer together.

gigahertz

and the clock speed of a computer is now usually stated in *gigahertz (GHz)*. Intel’s newest processors run at about 3 GHz.

core

CPUs were originally developed with only one core. The *core* is the part of the processor that performs the reading and executing of instructions. In order to increase CPU processing power, chip manufacturers are now producing CPUs that contain multiple cores. A multicore CPU is a single component with two or more independent processors. Today’s consumer computers typically have two, three, and even four separate cores. Soon, CPUs with dozens or even hundreds of cores will be affordable.

1.2.2 Bits and Bytes

Before we discuss memory, let’s look at how information (data and programs) are stored in a computer.

bits

A computer is really nothing more than a series of switches. Each switch exists in two states: on or off. Storing information in a computer is simply a matter of setting a sequence of switches on or off. If the switch is on, its value is 1. If the switch is off, its value is 0. These 0s and 1s are interpreted as digits in the binary number system and are called *bits* (binary digits).

byte

The minimum storage unit in a computer is a *byte*. A byte is composed of eight bits. A small number such as 3 can be stored as a single byte. To store a number that cannot fit into a single byte, the computer uses several bytes.

encoding scheme

Data of various kinds, such as numbers and characters, are encoded as a series of bytes. As a programmer, you don’t need to worry about the encoding and decoding of data, which the computer system performs automatically, based on the encoding scheme. An *encoding scheme* is a set of rules that govern how a computer translates characters, numbers, and symbols into data the computer can actually work with. Most schemes translate each character into a predetermined string of numbers. In the popular ASCII encoding scheme, for example, the character C is represented as 01000011 in one byte.

A computer’s storage capacity is measured in bytes and multiples of the byte, as follows:

- A *kilobyte (KB)* is about 1,000 bytes. kilobyte (KB)
- A *megabyte (MB)* is about 1 million bytes. megabyte (MB)
- A *gigabyte (GB)* is about 1 billion bytes. gigabyte (GB)
- A *terabyte (TB)* is about 1 trillion bytes. terabyte (TB)

A typical one-page word document might take 20 KB. Therefore, 1 MB can store 50 pages of documents and 1 GB can store 50,000 pages of documents. A typical two-hour high-resolution movie might take 8 GB, so it would require 160 GB to store 20 movies.

1.2.3 Memory

A computer’s *memory* consists of an ordered sequence of bytes for storing programs as well as data that the program is working with. You can think of memory as the computer’s work area for executing a program. A program and its data must be moved into the computer’s memory before they can be executed by the CPU.

Every byte in the memory has a *unique address*, as shown in Figure 1.3. The address is used to locate the byte for storing and retrieving the data. Since the bytes in the memory can be accessed in any order, the memory is also referred to as *random-access memory (RAM)*.

Memory address		Memory content
	↓	↓
.		.
.		.
.		.
2000		01000011 Encoding for character ‘C’
2001		01110010 Encoding for character ‘r’
2002		01100101 Encoding for character ‘e’
2003		01110111 Encoding for character ‘w’
2004		00000011 Encoding for number 3
.		.

FIGURE 1.3 Memory stores data and program instructions in uniquely addressed memory locations. Each memory location can store one byte of data.

Today’s personal computers usually have at least 1 gigabyte of RAM, but they more commonly have 2 to 4 GB installed. Generally speaking, the more RAM a computer has, the faster it can operate, but there are limits to this simple rule of thumb.

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.

Like the CPU, memory is built on silicon semiconductor chips that have millions of transistors embedded on their surface. Compared to CPU chips, memory chips are less complicated, slower, and less expensive.

1.2.4 Storage Devices

A computer’s memory (RAM) is a volatile form of data storage: any information that has been stored in memory (that is, saved) is lost when the system’s 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 operates at much faster speeds than permanent storage devices can.

There are three main types of storage devices:

- Magnetic disk drives
- Optical disc drives (CD and DVD)
- USB flash drives

drive *Drives* are devices for operating a medium, such as disks and CDs. A storage medium physically stores data and program instructions. The drive reads data from the medium and writes data onto the medium.

Disks

hard disk A computer usually has at least one hard disk drive (Figure 1.4). *Hard disks* are used for permanently storing data and programs. Newer computers have hard disks that can store from 200 to 800 gigabytes of data. Hard disk drives are usually encased inside the computer, but removable hard disks are also available.



FIGURE 1.4 A hard disk is a device for permanently storing programs and data.

CDs and DVDs

CD-R *CD* stands for compact disc. 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; that is, you can write data onto the disc, and then overwrite that data with new data. A single CD can hold up to 700 MB. Most new PCs are equipped with a CD-RW drive that can work with both CD-R and CD-RW discs.

DVD *DVD* stands for digital versatile disc or digital video disc. 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. Like CDs, there are two types of DVDs: DVD-R (read-only) and DVD-RW (rewritable).

USB Flash Drives

Universal serial bus (USB) connectors allow the user to attach many kinds of peripheral devices to the computer. You can use a USB to connect a printer, digital camera, mouse, external hard disk drive, and other devices to the computer.

A *USB flash drive* is a device for storing and transporting data. A flash drive is small—about the size of a pack of gum, as shown in Figure 1.5. 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 256 GB storage capacity.

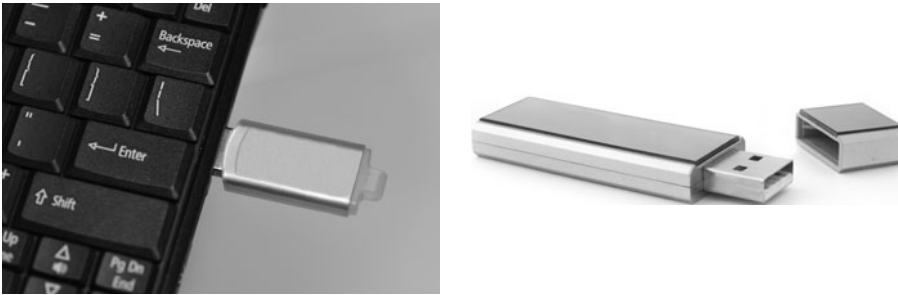


FIGURE 1.5 USB flash drives are very portable and can store a lot of data.

1.2.5 Input and Output Devices

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

The Keyboard

A keyboard is a device for entering input. Figure 1.6 shows a typical keyboard. Compact keyboards are available without a numeric keypad.



FIGURE 1.6 A computer keyboard consists of the keys for sending input to a computer.

Function keys are located across the top of the keyboard and are prefaced with the letter *F*. function key
 Their functions depend on the software currently being used.

8 Chapter 1 Introduction to Computers, Programs, and Java

modifier key

A *modifier key* is a special key (such as the *Shift*, *Alt*, and *Ctrl* keys) that modifies the normal action of another key when the two are pressed simultaneously.

numeric keypad

The *numeric keypad*, located on the right side of most keyboards, is a separate set of keys styled like a calculator to use for entering numbers quickly.

arrow keys

Arrow keys, located between the main keypad and the numeric keypad, are used to move the mouse pointer up, down, left, and right on the screen in many kinds of programs.

Insert key

Delete key

Page Up key

Page Down key

The *Insert*, *Delete*, *Page Up*, and *Page Down* keys are used in word processing and other programs for inserting text and objects, deleting text and objects, and moving up or down through a document one screen at a time.

The Mouse

A *mouse* is a pointing device. It is used to move a graphical pointer (usually in the shape of an arrow) called a *cursor* around the screen or to click on-screen objects (such as a button) to trigger them to perform an action.

The Monitor

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

screen resolution

pixels

The *screen resolution* specifies the number of pixels in horizontal and vertical dimensions of the display device. *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 1,024 pixels wide and 768 pixels high. The resolution can be set manually. The higher the resolution, the sharper and clearer the image is.

dot pitch

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

1.2.6 Communication Devices

modem

Computers can be networked through communication devices, such as a dial-up *modem* (modulator/demodulator), a DSL or cable modem, a wired network interface card, or a wireless adapter.

- A dial-up modem uses a phone line and can transfer data at a speed up to 56,000 bps (bits per second).

digital subscriber line (DSL)

- A *digital subscriber line (DSL)* connection also uses a standard phone line, but it can transfer data 20 times faster than a standard dial-up modem.

cable modem

- A *cable modem* uses the cable TV line maintained by the cable company and is generally faster than DSL.

network interface card (NIC)

local area network (LAN)

- A *network interface card (NIC)* is a device that connects a computer to a *local area network (LAN)*, as shown in Figure 1.7. LANs are commonly used in universities, businesses, and government agencies. A high-speed NIC called *1000BaseT* can transfer data at 1,000 million bits per second (mbps).

million bits per second
(mbps)

- Wireless networking is now extremely popular in homes, businesses, and schools. Every laptop computer sold today is equipped with a wireless adapter that enables the computer to connect to a local area network and the Internet.



Note

Answers to checkpoint questions are on the Companion Website.



MyProgrammingLab™

1.1 What are hardware and software?

1.2 List five major hardware components of a computer.

1.3 What does the acronym “CPU” stand for?

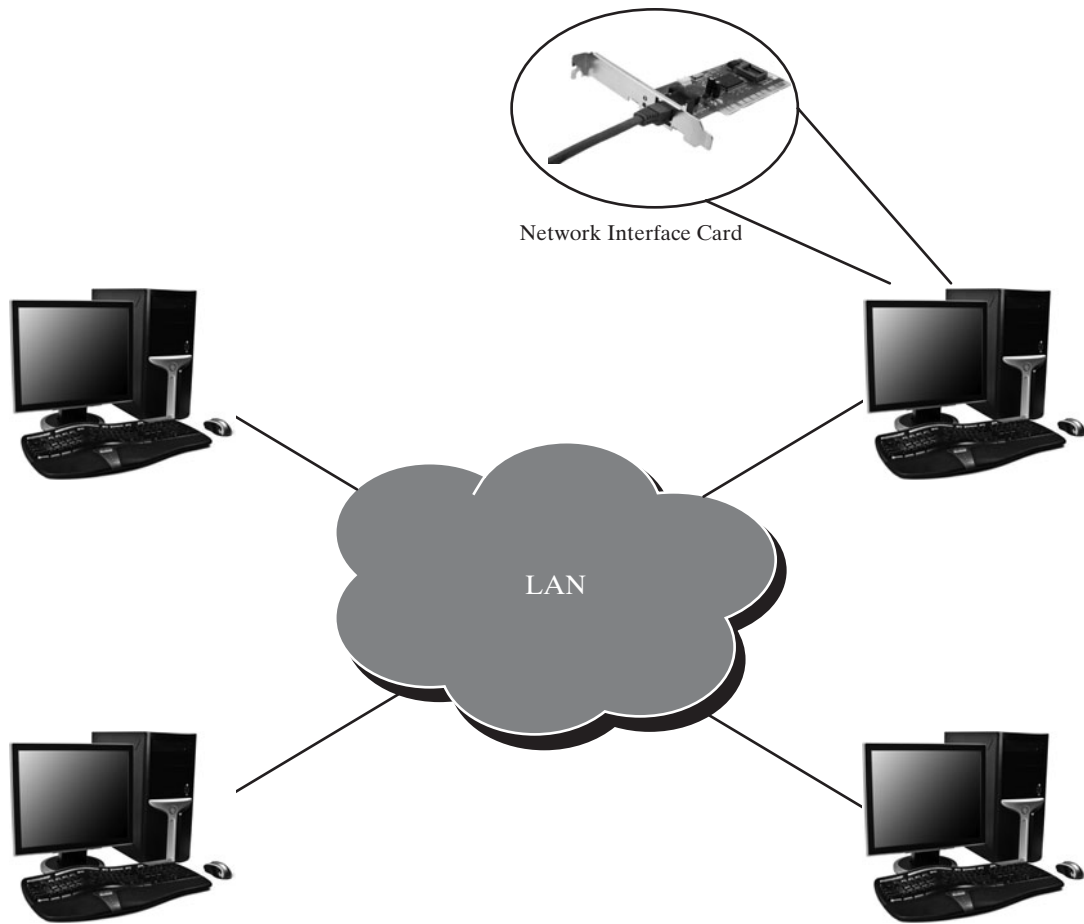


FIGURE 1.7 A local area network connects computers in close proximity to each other.

- 1.4** What unit is used to measure CPU speed?
- 1.5** What is a bit? What is a byte?
- 1.6** What is memory for? What does RAM stand for? Why is memory called RAM?
- 1.7** What unit is used to measure memory size?
- 1.8** What unit is used to measure disk size?
- 1.9** What is the primary difference between memory and a storage device?

1.3 Programming Languages

Computer programs, known as software, are instructions that tell a computer what to do.



Computers do not understand human languages, so programs must be written in a language a computer can use. There are hundreds of programming languages, and they were developed to make the programming process easier for people. However, all programs must be converted into a language the computer can understand.

1.3.1 Machine Language

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 if you want to give a computer an instruction in its native language, you

machine language

have to enter the instruction as binary code. For example, to add two numbers, you might have to write an instruction in binary code, like this:

```
1101101010011010
```

1.3.2 Assembly Language

Programming in machine language is a tedious process. Moreover, programs written in machine language are very difficult to read and modify. For this reason, *assembly language* was created in the early days of computing as an alternative to machine languages. Assembly language uses a short descriptive word, known as a *mnemonic*, to represent each of the machine-language instructions. For example, the mnemonic **add** typically means to add numbers and **sub** means to subtract numbers. To add the numbers **2** and **3** and get the result, you might write an instruction in assembly code like this:

```
add 2, 3, result
```

Assembly languages were developed to make programming easier. However, because the computer cannot understand assembly language, another program—called an *assembler*—is used to translate assembly-language programs into machine code, as shown in Figure 1.8.

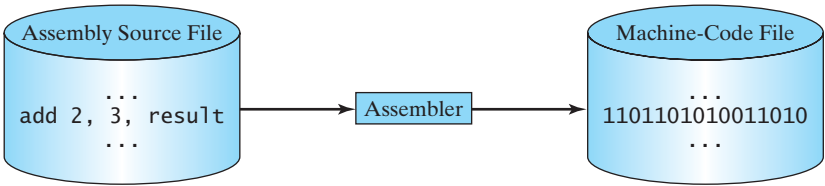


FIGURE 1.8 An assembler translates assembly-language instructions into machine code.

Writing code in assembly language is easier than in machine language. However, it is still tedious to write code in assembly language. An instruction in assembly language essentially corresponds to an instruction in machine code. Writing in assembly requires that you know how the CPU works. Assembly language is referred to as a *low-level language*, because assembly language is close in nature to machine language and is machine dependent.

1.3.3 High-Level Language

In the 1950s, a new generation of programming languages known as *high-level languages* emerged. They are platform-independent, which means that you can write a program in a high-level language and run it in different types of machines. High-level languages are English-like and easy to learn and use. The instructions in a high-level programming language are called *statements*. Here, for example, is a high-level language statement that computes the area of a circle with a radius of **5**:

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

There are many high-level programming languages, and each was designed for a specific purpose. Table 1.1 lists some popular ones.

A program written in a high-level language is called a *source program* or *source code*. Because a computer cannot understand a source program, a source program must be translated into machine code for execution. The translation can be done using another programming tool called an *interpreter* or a *compiler*.

- An interpreter reads one statement from the source code, translates it to the machine code or virtual machine code, and then executes it right away, as shown in Figure 1.9a.

TABLE 1.1 Popular High-Level Programming Languages

Language	Description
Ada	Named for Ada Lovelace, who worked on mechanical general-purpose computers. The Ada language was developed for the Department of Defense and is used mainly in defense projects.
BASIC	Beginner's All-purpose Symbolic Instruction Code. It was designed to be learned and used easily by beginners.
C	Developed at Bell Laboratories. C combines the power of an assembly language with the ease of use and portability of a high-level language.
C++	C++ is an object-oriented language, based on C.
C#	Pronounced "C Sharp." It is a hybrid of Java and C++ and was developed by Microsoft.
COBOL	COMmon Business Oriented Language. Used for business applications.
FORTRAN	FORmula TRANslation. Popular for scientific and mathematical applications.
Java	Developed by Sun Microsystems, now part of Oracle. It is widely used for developing platform-independent Internet applications.
Pascal	Named for Blaise Pascal, who pioneered calculating machines in the seventeenth century. It is a simple, structured, general-purpose language primarily for teaching programming.
Python	A simple general-purpose scripting language good for writing short programs.
Visual Basic	Visual Basic was developed by Microsoft and it enables the programmers to rapidly develop graphical user interfaces.

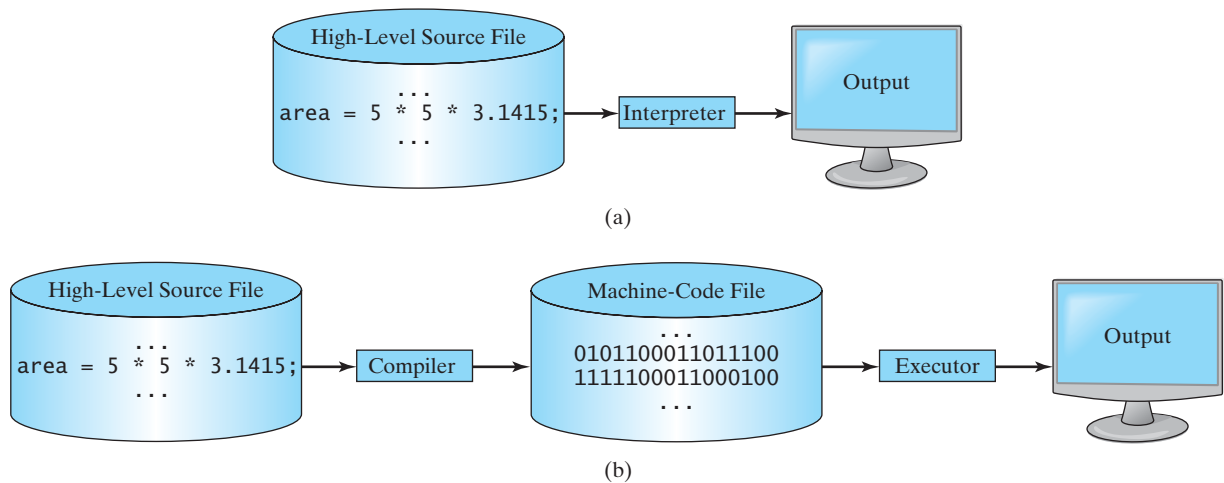


FIGURE 1.9 (a) An interpreter translates and executes a program one statement at a time. (b) A compiler translates the entire source program into a machine-language file for execution.

Note that a statement from the source code may be translated into several machine instructions.

- A compiler translates the entire source code into a machine-code file, and the machine-code file is then executed, as shown in Figure 1.9b.

1.10 What language does the CPU understand?

1.11 What is an assembly language?



MyProgrammingLab™

- I.12** What is an assembler?
- I.13** What is a high-level programming language?
- I.14** What is a source program?
- I.15** What is an interpreter?
- I.16** What is a compiler?
- I.17** What is the difference between an interpreted language and a compiled language?

1.4 Operating Systems



*The operating system (OS) is the most important program that runs on a computer.
The OS manages and controls a computer's activities.*

operating system (OS)

The popular *operating systems* for general-purpose computers are Microsoft Windows, Mac OS, and Linux. Application programs, such as a Web browser or a word processor, cannot run unless an operating system is installed and running on the computer. Figure 1.10 shows the interrelationship of hardware, operating system, application software, and the user.

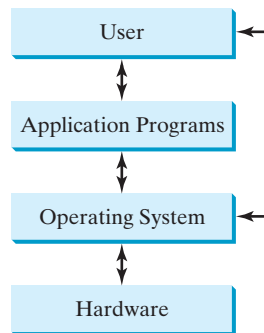


FIGURE 1.10 Users and applications access the computer's hardware via the operating system.

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 folders on storage devices, and controlling peripheral devices, such as disk drives and printers. An operating system must also ensure that different programs and users working at the same time do not interfere with each other. In addition, the OS is responsible for security, ensuring that unauthorized users and programs do not access the system.

1.4.2 Allocating and Assigning System Resources

The operating system is responsible for determining what computer resources a program needs (such as CPU time, memory space, 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' activities to make efficient use of system resources. Many of today's operating systems support such techniques as *multiprogramming*, *multithreading*, and *multiprocessing* to increase system performance.

Multiprogramming allows multiple programs to run simultaneously by sharing the same 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 a disk or waiting for other system resources to respond. A multiprogramming OS takes advantage of this situation by allowing multiple programs to use the CPU when it would otherwise be idle. For example, multiprogramming enables you to use a word processor to edit a file at the same time as your Web browser is downloading a file.

multiprogramming

Multithreading allows a single program to execute multiple tasks at the same time. For instance, a word-processing program allows users to simultaneously edit text and save it to a disk. In this example, editing and saving are two tasks within the same application. These two tasks may run concurrently.

multithreading

Multiprocessing, or *parallel processing*, uses two or more processors together to perform subtasks concurrently and then combine solutions of the subtasks to obtain a solution for the entire task. It is like a surgical operation where several doctors work together on one patient.

multiprocessing

1.18 What is an operating system? List some popular operating systems.

1.19 What are the major responsibilities of an operating system?

1.20 What are multiprogramming, multithreading, and multiprocessing?



MyProgrammingLab™

1.5 Java, the World Wide Web, and Beyond

Java is a powerful and versatile programming language for developing software running on mobile devices, desktop computers, and servers.



This book introduces Java programming. Java was developed by a team led by James Gosling at Sun Microsystems. Sun Microsystems was purchased by Oracle in 2010. Originally called *Oak*, Java was designed in 1991 for use in embedded chips in consumer electronic appliances. In 1995, renamed *Java*, it was redesigned for developing Web applications. For the history of Java, see www.java.com/en/javahistory/index.jsp.

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 its designer, 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, desktop computers, 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 forty years. The colorful World Wide Web and sophisticated Web browsers are the major reason for the Internet's popularity.

applet

HTML

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 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. Applets are embedded in an HTML file. *HTML* (*Hypertext Markup Language*) is a simple scripting language for laying out documents, linking documents on the Internet, and bringing images, sound, and video alive on the Web. Figure 1.11 shows an applet running from a Web browser for playing a tic-tac-toe game.

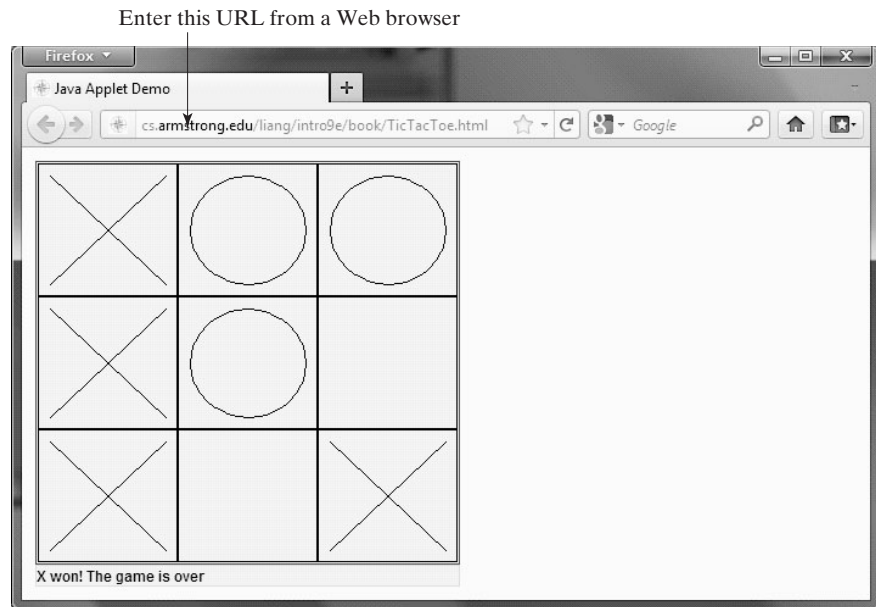


FIGURE 1.11 A Java applet for playing tic-tac-toe runs from a Web browser.



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 is now very popular for developing applications on Web servers. These applications process data, perform computations, and generate dynamic Web pages. The LiveLab automatic grading system, shown in Figure 1.12 and which you can use with this book, was developed using Java.

Java is a versatile programming language: You can use it to develop applications for desktop computers, servers, and small hand-held devices. The software for Android cell phones is developed using Java. Figure 1.13 shows an emulator for developing Android phone applications.



1.21 Who invented Java? Which company owns Java now?

1.22 What is a Java applet?

1.23 What programming language does Android use?

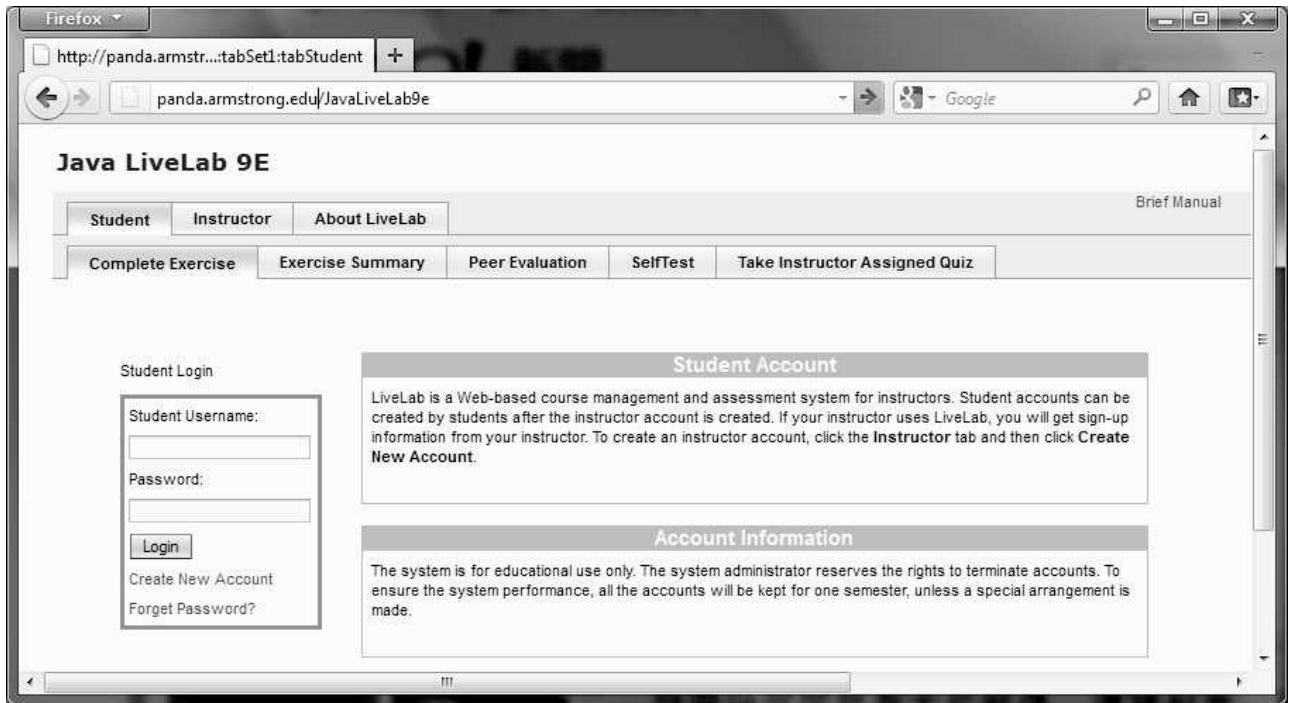


FIGURE 1.12 Java was used to develop LiveLab, the automatic grading system that accompanies this book.



FIGURE 1.13 Java is used in Android phones.

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



Java syntax is defined in the Java language specification, and the Java library is defined in the Java API. The JDK is the software for developing and running Java programs. An IDE is an integrated development environment for rapidly developing programs.

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

Java language specification

The *Java language specification* is a technical definition of the Java programming language's syntax and semantics. You can find the complete Java language specification at java.sun.com/docs/books/jls.

API
library

The *application program interface (API)*, also known as *library*, contains predefined classes and interfaces for developing Java programs. The API is still expanding. You can view and download the latest version of the Java API at www.oracle.com/technetwork/java/index.html.

Java is a full-fledged and powerful language that can be used in many ways. It comes in three editions:

Java SE, EE, and ME

- *Java Standard Edition (Java SE)* to develop client-side standalone applications or applets.
- *Java Enterprise Edition (Java EE)* to develop server-side applications, such as Java servlets, JavaServer Pages (JSP), and JavaServer Faces (JSF).
- *Java Micro Edition (Java ME)* to develop applications for mobile devices, such as cell phones.

This book uses Java SE to introduce Java programming. Java SE is the foundation upon which all other Java technology is based. There are many versions of Java SE. The latest, Java SE 7, is used in this book. Oracle releases each version with a *Java Development Toolkit (JDK)*. For Java SE 7, the Java Development Toolkit is called *JDK 1.7* (also known as *Java 7* or *JDK 7*).

Java Development Toolkit
(JDK)
JDK 1.7 = JDK 7

The JDK consists of a set of separate programs, each invoked from a command line, for developing and testing Java programs. Instead of using the JDK, you can use a Java development tool (e.g., NetBeans, Eclipse, and TextPad)—software that provides an *integrated development environment (IDE)* for developing Java programs quickly. Editing, compiling, building, debugging, and online help are integrated in one graphical user interface. You simply enter source code in one window or open an existing file in a window, and then click a button or menu item or press a function key to compile and run the program.

Integrated development
environment



MyProgrammingLab™

1.24 What is the Java language specification?

1.25 What does JDK stand for?

1.26 What does IDE stand for?

1.27 Are tools like NetBeans and Eclipse different languages from Java, or are they dialects or extensions of Java?

1.7 A Simple Java Program



*A Java program is executed from the **main** method in the class.*

what is a console?
console input
console output

Let's begin with a simple Java program that displays the message **Welcome to Java!** on the console. (The word *console* is an old computer term that refers to the text entry and display device of a computer. *Console input* means to receive input from the keyboard, and *console output* means to display output on the monitor.) The program is shown in Listing 1.1.

LISTING 1.1 Welcome.java

```

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

```

class
main method
display message



VideoNote
Your first Java program



Welcome to Java!

Note that the line numbers are for reference purposes only; they 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. The program is executed from the **main** method. A class may contain several methods. The **main** method is the entry point where the program begins execution.

main method

A method is a construct that contains statements. The **main** method in this program contains the **System.out.println** statement. This statement displays the string **Welcome to Java!** on the console (line 4). *String* is a programming term meaning a sequence of characters. A string must be enclosed in double quotation marks. Every statement in Java ends with a semicolon (;), known as the *statement terminator*.

string

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

keyword

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* or *paragraph 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:

comment

line comment

block comment

```

// This application program displays Welcome to Java!
/* This application program displays Welcome to Java! */
/* This application program
   displays Welcome to Java! */

```

A pair of curly 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. Similarly, 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.

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.

match braces

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

Diagram labels:
- "Class block" points to the outer curly braces of the class.
- "Method block" points to the curly braces of the `main` method.

case sensitive

special characters



Caution

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

You have seen several special characters (e.g., `{ }`, `//`, `;`) in the program. They are used in almost every program. Table 1.2 summarizes their uses.

TABLE 1.2 Special Characters

Character	Name	Description
{ }	Opening and closing braces	Denote a block to enclose statements.
()	Opening and closing parentheses	Used with methods.
[]	Opening and closing brackets	Denote an array.
//	Double slashes	Precede a comment line.
" "	Opening and closing quotation marks	Enclose a string (i.e., sequence of characters).
;	Semicolon	Mark the end of a statement.

common errors

syntax rules

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



Note

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

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 WelcomeWithThreeMessages.java

class
main method
display message

```
1 public class WelcomeWithThreeMessages {  
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 on 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 can see, it is a straightforward process to translate an arithmetic expression to a Java expression. We will discuss Java expressions further in Chapter 2.

- I.28** What is a keyword? List some Java keywords.
- I.29** Is Java case sensitive? What is the case for Java keywords?
- I.30** What is a comment? Is the comment ignored by the compiler? How do you denote a comment line and a comment paragraph?
- I.31** What is the statement to display a string on the console?
- I.32** 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);
    }
}
```



Check
Point

MyProgrammingLab™

1.8 Creating, Compiling, and Executing a Java Program

You save a Java program in a .java file and compile it into a .class file. The .class file is executed by the Java Virtual Machine.



Key
Point

You have to create your program and compile it before it can be executed. This process is repetitive, as shown in Figure 1.14. If your program has compile 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.

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, refer to Supplement II for tutorials. From the command window, you can use a text editor such as Notepad to create the Java source-code file, as shown in Figure 1.15.



VideoNote
Eclipse brief tutorial

command window
IDE Supplements



VideoNote
NetBeans brief tutorial

file name



Note

The source file must end with the extension `.java` and must have the same exact 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`.

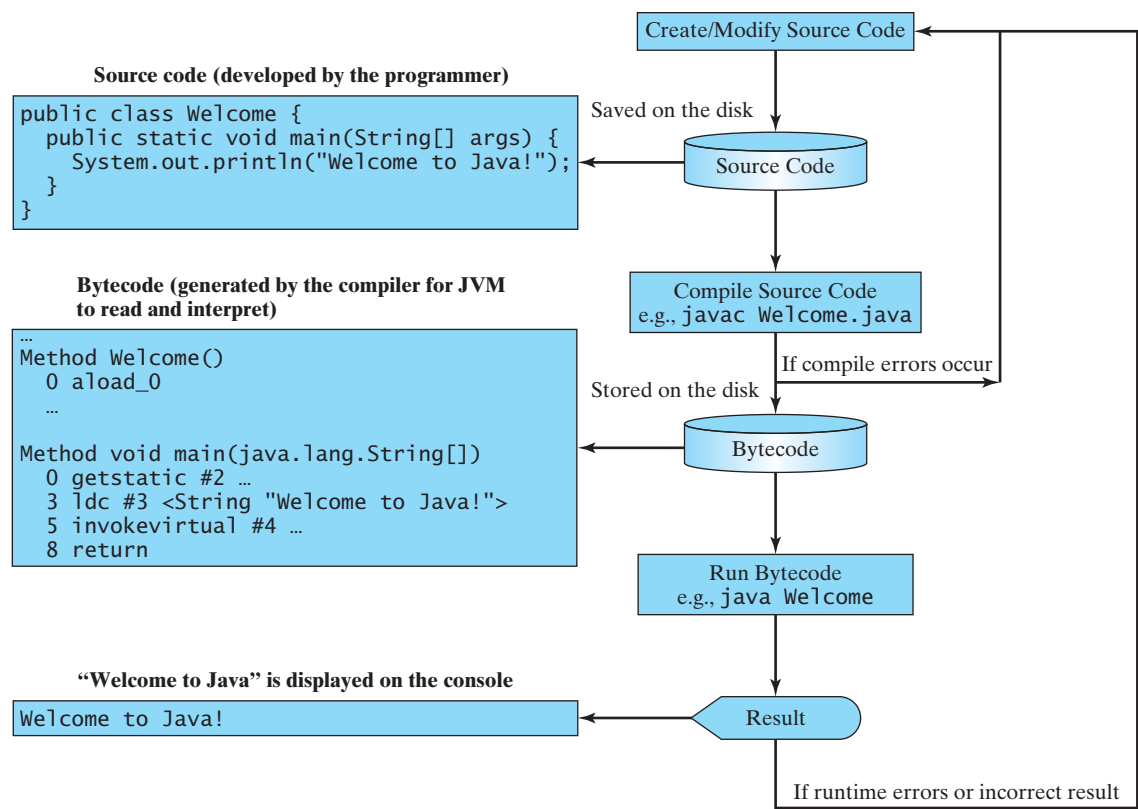


FIGURE 1.14 The Java program-development process consists of repeatedly creating/modifying source code, compiling, and executing programs.



FIGURE 1.15 You can create a Java source file using Windows Notepad.

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

```
javac Welcome.java
```



Note

You must first install and configure the JDK before you can compile and run programs. See Supplement I.B, Installing and Configuring JDK 7, for how to install the JDK and set up the environment to compile and run Java programs. If you have trouble compiling and running programs, 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.

.class bytecode file

If there aren't any syntax errors, the *compiler* generates a bytecode file with a **.class** extension. Thus, the preceding command generates a file named **Welcome.class**, as shown in

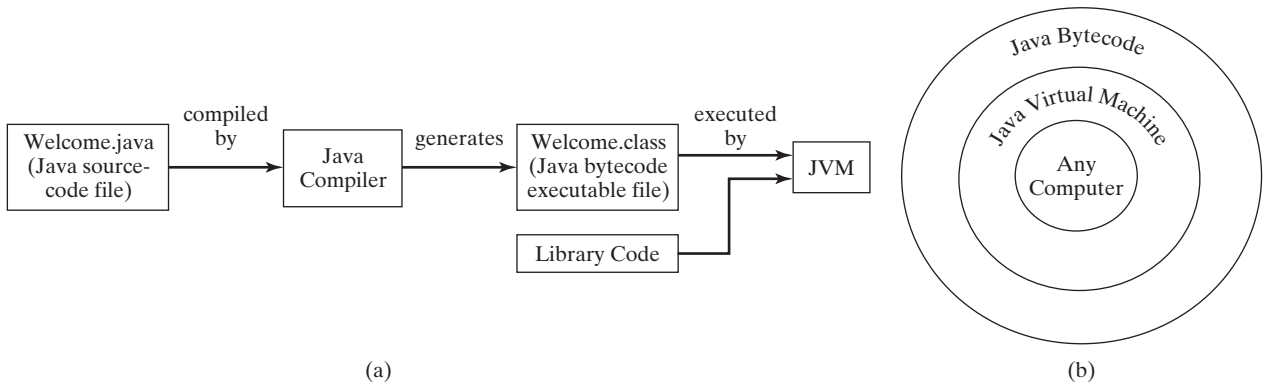


FIGURE 1.16 (a) Java source code is translated into bytecode. (b) Java bytecode can be executed on any computer with a Java Virtual Machine.

Figure 1.16a. The Java language is a high-level language, but 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 1.16b. 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*. Java source code is compiled into Java bytecode and Java bytecode is interpreted by the JVM. Your Java code may use the code in the Java library. The JVM executes your code along with the code in the library.

To execute a Java program is to run the program’s bytecode. You can execute the bytecode on any platform with a JVM, which is an interpreter. It translates the individual instructions in the bytecode into the target machine language code one at a time rather than the whole program as a single unit. Each step is executed immediately after it is translated.

The following command runs the bytecode for Listing 1.1:

```
java Welcome
```

Figure 1.17 shows the **javac** command for compiling **Welcome.java**. The compiler generates the **Welcome.class** file, and this file is executed using the **java** command.



Note

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

```

c:\book>javac Welcome.java
c:\book>dir Welcome.*
Volume in drive C has no label.
Volume Serial Number is 2EF7-CA93

Directory of c:\book

10/29/2011  03:43 PM                424 Welcome.class
10/29/2011  03:42 PM                176 Welcome.java
               2 File(s)                600 bytes
               0 Dir(s)  70,200,397,824 bytes free

c:\book>java Welcome
Welcome to Java!
c:\book>_
  
```



VideoNote

Compile and run a Java program

FIGURE 1.17 The output of Listing 1.1 displays the message “Welcome to Java!”

java ClassName



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`.

NoClassDefFoundError



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.

NoSuchMethodError

class loader



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 the *bytecode verifier* to check the validity of the bytecode and to ensure that the bytecode does not violate Java's security restrictions. Java enforces strict security to make sure that Java class files are not tampered and do not harm your computer.

bytecode verifier

use package



Pedagogical Note

Your instructor may require you to use packages for organizing programs. For example, you may place all programs in this chapter in a package named *chapter1*. For instructions on how to use packages, see Supplement I.F, Using Packages to Organize the Classes in the Text.



Check
Point

MyProgrammingLab™

- I.33** What is the Java source filename extension, and what is the Java bytecode filename extension?
- I.34** What are the input and output of a Java compiler?
- I.35** What is the command to compile a Java program?
- I.36** What is the command to run a Java program?
- I.37** What is the JVM?
- I.38** Can Java run on any machine? What is needed to run Java on a computer?
- I.39** If a `NoClassDefFoundError` occurs when you run a program, what is the cause of the error?
- I.40** If a `NoSuchMethodError` occurs when you run a program, what is the cause of the error?

1.9 Displaying Text in a Message Dialog Box

You can display text in a graphical dialog box.



Key
Point

JOptionPane

showMessageDialog

The program in Listing 1.1 displays the text on the console, as shown in Figure 1.17. 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 library that you can reuse rather than “reinvent the wheel.” You can use the `showMessageDialog` method to display any text in a message dialog box, as shown in Figure 1.18. The new program is given in Listing 1.4.

LISTING 1.4 WelcomeInMessageDialogBox.java

block comment

```
1  /* This application program displays Welcome to Java!
2   *   in a message dialog box.
3   */
```

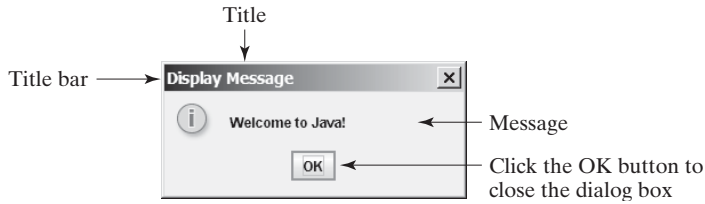


FIGURE 1.18 “Welcome to Java!” is displayed in a message box.

```

4  import javax.swing.JOptionPane;           import
5
6  public class WelcomeInMessageDialogBox {
7      public static void main(String[] args) {           main method
8          // Display Welcome to Java! in a message dialog box
9          JOptionPane.showMessageDialog(null, "Welcome to Java!");           display message
10     }
11 }
```

The first three lines are block comments. The first line begins with `/*` and the last line ends with `*/`. By convention, all other lines begin with an asterisk (`*`).

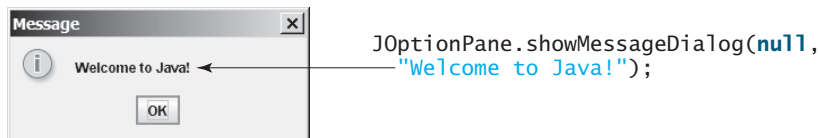
This program uses the Java class `JOptionPane` (line 9). Java’s predefined classes are grouped into packages. `JOptionPane` is in the `javax.swing` package. `JOptionPane` is imported into the program using the `import` statement in line 4 so that the compiler can locate the class without the full name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



Note

If you replace `JOptionPane` in 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. Details of methods will be discussed in Chapter 5. The `showMessageDialog` method can be invoked with two arguments, as shown below.



The first argument can always be `null`. `null` is a Java keyword that will be fully discussed in Chapter 8. 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`

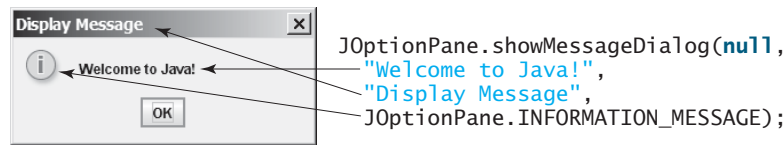
```
JOptionPane.showMessageDialog(null, x);
```

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 information icon () to be displayed in the message box, as shown in the following example.



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 the package `javax.swing`.

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

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

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

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.



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.

specific import

wildcard import

no performance difference

java.lang package
implicitly imported



MyProgrammingLab™

- I.41** What is the statement to display the message “Hello world” in a message dialog box?
- I.42** Why does the `System` class not need to be imported?
- I.43** Are there any performance differences between the following two **import** statements?

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

1.10 Programming Style and Documentation



Good programming style and proper documentation make a program easy to read and help programmers prevent errors.

programming style

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. This section gives several guidelines. For

more detailed guidelines, see Supplement I.D, Java Coding Style Guidelines, on the Companion Website.

I.10.1 Appropriate Comments and Comment Styles

Include a summary at the beginning of the program that explains 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 comments (beginning with `//`) and block comments (beginning with `/*`), 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 the 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 into a javadoc HTML file. For commenting on steps inside a method, use line comments (`//`). To see an example of a javadoc HTML file, check out www.cs.armstrong.edu/liang/javadoc/Exercise1.html. Its corresponding Java code is shown in www.cs.armstrong.edu/liang/javadoc/Exercise1.java.

I.10.2 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 on the same long line, but humans find it easier to read and maintain code that is aligned properly. Indent each sub-component 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:

<code>System.out.println(3+4*4);</code>	← Bad style
<code>System.out.println(3 + 4 * 4);</code>	← Good style

I.10.3 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.

<pre>public class Test { public static void main(String[] args) { System.out.println("Block Styles"); } }</pre>	<pre>public class Test { public static void main(String[] args) { System.out.println("Block Styles"); } }</pre>
Next-line style	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.



I.44 Reformat the following program according to the programming style and documentation guidelines. Use the end-of-line brace style.

MyProgrammingLab™

```
public class Test
{
    // Main method
    public static void main(String[] args) {
        /** Display output */
        System.out.println("Welcome to Java");
    }
}
```

1.1.1 Programming Errors



Programming errors can be categorized into three types: syntax errors, runtime errors, and logic errors.

1.1.1.1 Syntax Errors

syntax errors
compile errors

Errors that are detected by the compiler 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 program in Listing 1.5 has a syntax error, as shown in Figure 1.19.

LISTING 1.5 ShowSyntaxErrors.java

```
1 public class ShowSyntaxErrors {
2     public static main(String[] args) {
3         System.out.println("Welcome to Java");
4     }
5 }
```

Four errors are reported, but the program actually has two errors:

- The keyword **void** is missing before **main** in line 2.
- The string **Welcome to Java** should be closed with a closing quotation mark in line 3.

Since a single error will often display many lines of compile errors, it is a good practice to fix errors from the top line and work downward. Fixing errors that occur earlier in the program may also fix additional errors that occur later.

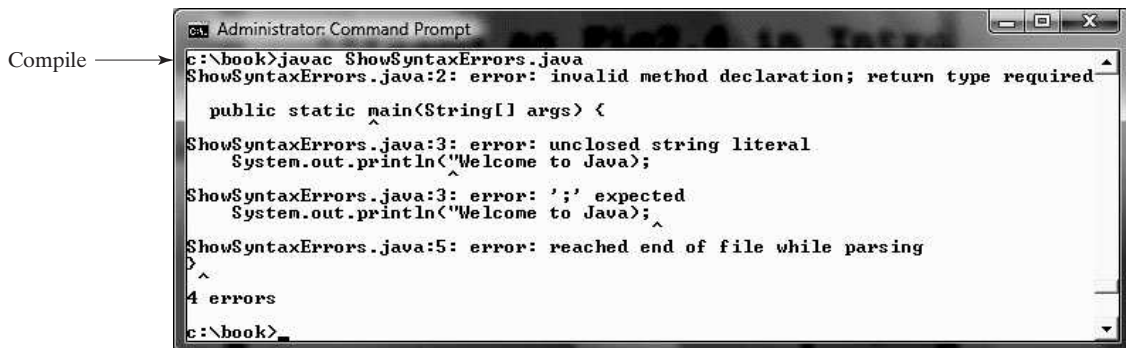


FIGURE 1.19 The compiler reports syntax errors.

**Tip**

If you don't know how to correct it, compare your program closely, character by character, with similar examples in the text. In the first few weeks of this course, you will probably spend a lot of time fixing syntax errors. Soon you will be familiar with Java syntax and can quickly fix syntax errors.

fix syntax errors

I.1.1.2 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 mistakes typically cause runtime errors. An *input error* occurs when the program is waiting for the user to enter a value, but the user enters a 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.

runtime errors

Another example of runtime errors is division by zero. This happens when the divisor is zero for integer divisions. For instance, the program in Listing 1.6 would cause a runtime error, as shown in Figure 1.20.

LISTING 1.6 ShowRuntimeErrors.java

```
1 public class ShowRuntimeErrors {
2     public static void main(String[] args) {
3         System.out.println(1 / 0);
4     }
5 }
```

runtime error



FIGURE 1.20 The runtime error causes the program to terminate abnormally.

I.1.1.3 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 program in Listing 1.7 to convert Celsius 35 degrees to a Fahrenheit degree:

logic errors

LISTING 1.7 ShowLogicErrors.java

```
1 public class ShowLogicErrors {
2     public static void main(String[] args) {
3         System.out.println("Celsius 35 is Fahrenheit degree ");
4         System.out.println((9 / 5) * 35 + 32);
5     }
6 }
```

Celsius 35 is Fahrenheit degree
67



You will get Fahrenheit 67 degrees, which is wrong. It should be 95.0. In Java, the division for integers is an integer—the fractional part is truncated—so in Java $9 / 5$ is 1. To get the correct result, you need to use $9.0 / 5$, which results in 1.8.

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 for the errors are displayed on the console when the program aborts. Finding logic errors, on the other hand, can be very challenging. In the upcoming chapters, you will learn the techniques of tracing programs and finding logic errors.



MyProgrammingLab™

- I.45** What are syntax errors (compile errors), runtime errors, and logic errors?
- I.46** Give examples of syntax errors, runtime errors, and logic errors.
- I.47** If you forget to put a closing quotation mark on a string, what kind error will be raised?
- I.48** If your program needs to read integers, but the user entered strings, an error would occur when running this program. What kind of error is this?
- I.49** Suppose you write a program for computing the perimeter of a rectangle and you mistakenly write your program so that it computes the area of a rectangle. What kind of error is this?
- I.50** 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 }
```

- I.51** The following program is wrong. Reorder the lines so that the program displays **morning** followed by **afternoon**.

```
1 public static void main(String[] args) {
2 }
3 public class Welcome {
4     System.out.println("afternoon");
5     System.out.println("morning");
6 }
```

KEY TERMS

Application Program Interface (API)	16	dot pitch	8
assembler	10	DSL (digital subscriber line)	8
assembly language	10	encoding scheme	4
bit	4	hardware	2
block	17	high-level language	10
block comment	17	integrated development environment (IDE)	16
bus	2	interpreter	10
byte	4	java command	21
bytecode	21	Java Development Toolkit (JDK)	16
bytecode verifier	22	Java language specification	16
cable modem	8	Java Virtual Machine (JVM)	21
central processing unit (CPU)	3	javac command	21
class loader	22	keyword (or reserved word)	17
comment	17	library	16
compiler	10	line comment	17
console	16		

logic error	27	programming	2
low-level language	10	runtime error	27
machine language	9	screen resolution	8
main method	17	software	2
memory	5	source code	10
modem	8	source program	10
motherboard	3	specific import	24
network interface card (NIC)	8	statement	10
operating system (OS)	12	storage devices	5
pixel	8	syntax error	26
program	2	wildcard import	24

**Note**

The above terms are defined in this 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 touched.
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 1,000 bytes, a megabyte about 1 million bytes, a gigabyte about 1 billion bytes, and a terabyte about 1,000 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 on any computer.
22. Java programs can be embedded in HTML pages and downloaded by Web browsers to bring live animation and interaction to Web clients.
23. The Java source file name must match the public class name in the program. Java source code files must 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 (**}**).
29. Methods are contained in a class. To run a Java program, the 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* or *paragraph comment*. Comments are ignored by the compiler.
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.
- 35.** Programming errors can be categorized into three types: *syntax errors*, *runtime errors*, and *logic errors*. Errors that occur during compilation are called syntax errors or *compile errors*. Runtime errors are errors that cause a program to terminate abnormally. Logic errors occur when a program does not perform the way it was intended to.

TEST QUESTIONS

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

PROGRAMMING EXERCISES

MyProgrammingLab™



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** (*Display three messages*) Write a program that displays **Welcome to Java**, **Welcome to Computer Science**, and **Programming is fun**.
- 1.2** (*Display five messages*) Write a program that displays **Welcome to Java** five times.
- *1.3** (*Display 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      A      V      A      A
    
```

- 1.4** (*Print 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** (*Compute 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** (*Approximate π*) π can be computed using the following formula:

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

Write a program that displays the result of $4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} \right)$

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

- 1.8** (*Area and perimeter of a circle*) Write a program that displays the area and perimeter of a circle that has a radius of **5.5** using the following formula:

$$\text{perimeter} = 2 \times \text{radius} \times \pi$$

$$\text{area} = \text{radius} \times \text{radius} \times \pi$$

- 1.9** (*Area and perimeter of a rectangle*) Write a program that displays the area and perimeter of a rectangle with the width of **4.5** and height of **7.9** using the following formula:

$$\text{area} = \text{width} \times \text{height}$$

- 1.10** (*Average speed in miles*) Assume a runner runs **14** kilometers in **45** minutes and **30** seconds. Write a program that displays the average speed in miles per hour. (Note that **1** mile is **1.6** kilometers.)

- *1.11** (*Population projection*) The U.S. Census Bureau projects population based on the following assumptions:

- One birth every 7 seconds
- One death every 13 seconds
- One new immigrant every 45 seconds

Write a program to display the population for each of the next five years. Assume the current population is 312,032,486 and one year has 365 days. *Hint:* In Java, if two integers perform division, the result is an integer. The fraction part is truncated. For example, **5** / **4** is **1** (not **1.25**) and **10** / **4** is **2** (not **2.5**).

- 1.12** (*Average speed in kilometers*) Assume a runner runs **24** miles in **1** hour, **40** minutes, and **35** seconds. Write a program that displays the average speed in kilometers per hour. (Note that **1** mile is **1.6** kilometers.)

ELEMENTARY PROGRAMMING

Objectives

- To write Java programs to perform simple computations (§2.2).
- To obtain input from the console using the **Scanner** class (§2.3).
- To use identifiers to name variables, constants, methods, and classes (§2.4).
- To use variables to store data (§§2.5–2.6).
- To program with assignment statements and assignment expressions (§2.6).
- To use constants to store permanent data (§2.7).
- To name classes, methods, variables, and constants by following their naming conventions (§2.8).
- To explore Java numeric primitive data types: **byte**, **short**, **int**, **long**, **float**, and **double** (§2.9.1).
- To perform operations using operators **+**, **-**, *****, **/**, and **%** (§2.9.2).
- To perform exponent operations using **Math.pow(a, b)** (§2.9.3).
- To write integer literals, floating-point literals, and literals in scientific notation (§2.10).
- To write and evaluate numeric expressions (§2.11).
- To obtain the current system time using **System.currentTimeMillis()** (§2.12).
- To use augmented assignment operators (§2.13).
- To distinguish between postincrement and preincrement and between postdecrement and predecrement (§2.14).
- To cast the value of one type to another type (§2.15).
- To describe the software development process and apply it to develop the loan payment program (§2.16).
- To represent characters using the **char** type (§2.17).
- To represent a string using the **String** type (§2.18).
- To obtain input using the **JOptionPane** input dialog boxes (§2.19).



2.1 Introduction



The focus of this chapter is on learning elementary programming techniques to solve problems.

In Chapter 1 you learned how to create, compile, and run very basic Java programs. Now you will learn how to solve problems by writing programs. Through these problems, you will learn elementary programming using primitive data types, variables, constants, operators, expressions, and input and output.

Suppose, for example, that you need to take out a student loan. Given the loan amount, loan term, and annual interest rate, can you write a program to compute the monthly payment and total payment? This chapter shows you how to write programs like this. Along the way, you learn the basic steps that go into analyzing a problem, designing a solution, and implementing the solution by creating a program.

2.2 Writing a Simple Program



Writing a program involves designing a strategy for solving the problem and then using a programming language to implement that strategy.

problem

Let's first consider the simple problem of computing the area of a circle. How do we write a program for solving this problem?

algorithm

Writing a program involves designing algorithms and translating algorithms into programming instructions, or code. An *algorithm* describes how a problem is solved by listing the actions that need to be taken and the order of their execution. Algorithms can help the programmer plan a program before writing it in a programming language. Algorithms can be described in natural languages or in *pseudocode* (natural language mixed with some programming code). The algorithm for calculating the area of a circle can be described as follows:

pseudocode

1. Read in the circle's radius.
2. Compute the area using the following formula:

$$\text{area} = \text{radius} \times \text{radius} \times \pi$$

3. Display the result.



Tip

It's always good practice to outline your program (or its underlying problem) in the form of an algorithm before you begin coding.

When you *code*—that is, when you write a program—you translate an algorithm into a program. You already know that every Java program begins with a class definition in which the keyword **class** is followed by the class name. Assume that you have chosen **ComputeArea** as the class name. The outline of the program would look like this:

```
public class ComputeArea {
    // Details to be given later
}
```

As you know, every Java program must have a **main** method where program execution begins. The program is then expanded as follows:

```
public class ComputeArea {
    public static void main(String[] args) {
        // Step 1: Read in radius

        // Step 2: Compute area
```

```

    // Step 3: Display the area
}
}

```

The program needs to read the radius entered by the user from the keyboard. This raises two important issues:

- Reading the radius.
- Storing the radius in the program.

Let's address the second issue first. In order to store the radius, the program needs to declare a symbol called a *variable*. A variable represents a value stored in the computer's memory.

Rather than using **x** and **y** as variable names, choose descriptive names: in this case, **radius** for radius, and **area** for area. To let the compiler know what **radius** and **area** are, specify their data types. That is the kind of the data stored in a variable, whether integer, *floating-point number*, or something else. This is known as *declaring variables*. Java provides simple data types for representing integers, floating-point numbers (i.e., numbers with a decimal point), characters, and Boolean types. These types are known as *primitive data types* or *fundamental types*.

Declare **radius** and **area** as double-precision floating-point numbers. The program can be expanded as follows:

```

public class ComputeArea {
    public static void main(String[] args) {
        double radius;
        double area;

        // Step 1: Read in radius

        // Step 2: Compute area

        // Step 3: Display the area
    }
}

```

The program declares **radius** and **area** as variables. The reserved word **double** indicates that **radius** and **area** are double-precision floating-point values stored in the computer.

The first step is to prompt the user to designate the circle's **radius**. You will learn how to prompt the user for information shortly. For now, to learn how variables work, you can assign a fixed value to **radius** in the program as you write the code; later, you'll modify the program to prompt the user for this value.

The second step is to compute **area** by assigning the result of the expression **radius * radius * 3.14159** to **area**.

In the final step, the program will display the value of **area** on the console by using the **System.out.println** method.

Listing 2.1 shows the complete program, and a sample run of the program is shown in Figure 2.1.

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; // radius is now 20

```

variable

descriptive names

data type

declare variables

floating-point number

primitive data types

```
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 }
```

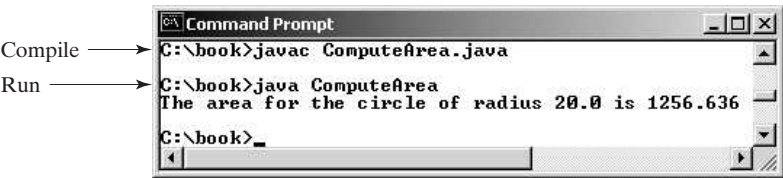


FIGURE 2.1 The program displays the area of a circle.

declare variable
assign value

tracing program

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 variable `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. This method of reviewing how a program works is called *tracing a program*. Tracing programs are helpful for understanding how programs work, and they are useful tools for finding errors in programs.



line#	radius	area
3	no value	
4		no value
7	20	
10		1256.636

concatenate strings

concatenate strings with
numbers

The plus sign (+) has two meanings: one for addition and the other for concatenating (combining) strings. The plus sign (+) in lines 13–14 is called a *string concatenation operator*. It combines two strings into one. If a string is combined with a number, the number is converted into a string and concatenated with the other string. Therefore, 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 Section 2.18.



Caution

A string 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");
```



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.

break a long string

incremental development and testing



MyProgrammingLab™

2.1 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 }
```

2.3 Reading Input from the Console

Reading input from the console enables the program to accept input from the user.

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, so instead you can use the **Scanner** class for console 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 listed in Table 2.1 to read various types of input.

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
2
3 public class ComputeAreaWithConsoleInput {
4     public static void main(String[] args) {
5         // Create a Scanner object
6         Scanner input = new Scanner(System.in);
7     }
```

import class

create a Scanner



VideoNote
Obtain input

TABLE 2.1 Methods for **Scanner** Objects

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

read a double

```
8      // Prompt the user to enter a radius
9      System.out.print("Enter a number for radius: ");
10     double radius = input.nextDouble();
11
12     // Compute area
13     double area = radius * radius * 3.14159;
14
15     // Display results
16     System.out.println("The area for the circle of radius " +
17         radius + " is " + area);
18 }
19 }
```



Enter a number for radius: 2.5

The area for the circle of radius 2.5 is 19.6349375



Enter a number for radius: 23

The area for the circle of radius 23.0 is 1661.90111

print vs. println

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 to the beginning of the next line after displaying the string, but **print** does not advance to the next line when completed.

The statement in line 10 reads input from the keyboard.

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

After the user enters a number and presses the *Enter* key, the program reads the number and assigns it to **radius**.


More details on objects will be introduced in Chapter 8. For the time being, simply accept that this is how to obtain input from the console.

Listing 2.3 gives an example of reading multiple input from the keyboard. The program reads three numbers and displays their average.

LISTING 2.3 ComputeAverage.java




```

1  import java.util.Scanner; // Scanner is in the java.util package      import class
2
3  public class ComputeAverage {
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 three numbers
9          System.out.print("Enter three numbers: ");
10         double number1 = input.nextDouble();                         read a double
11         double number2 = input.nextDouble();
12         double number3 = input.nextDouble();
13
14         // Compute average
15         double average = (number1 + number2 + number3) / 3;
16
17         // Display results
18         System.out.println("The average of " + number1 + " " + number2
19             + " " + number3 + " is " + average);
20     }
21 }
```

Enter three numbers: 1 2 3 
The average of 1.0 2.0 3.0 is 2.0



enter input in one line

Enter three numbers: 10.5 
11 
11.5 
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 for reading input from the keyboard.

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.

If you entered an input other than a numeric value, a runtime error would occur. In Chapter 14, you will learn how to handle the exception so that the program can continue to run.

runtime error



Note

Most of the programs in the early chapters of this book perform three steps: input, process, and output, called *IPO*. Input is receiving input from the user; process is producing results using the input; and output is displaying the results.

IPO



MyProgrammingLab™

2.2 How do you write a statement to let the user enter an integer or a double value from the keyboard?

2.3 What happens if you entered **5a** when executing the following code?
`double radius = input.nextDouble();`

2.4 Identifiers



Identifiers are the names that identify the elements such as classes, methods, and variables in a program.

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. In programming terminology, 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 for a list of reserved words.)
- An identifier cannot be **true**, **false**, or **null**.
- An identifier can be of any length.

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.

identifiers

identifier naming rules

case sensitive



Note

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



Tip

Identifiers are for naming variables, constants, methods, classes, and packages. Descriptive identifiers make programs easy to read. Avoid using abbreviations for identifiers. Using complete words is more descriptive. For example, **numberOfStudents** is better than **numStuds**, **numOfStuds**, or **numOfStudents**. We use descriptive names for complete programs in the text. However, we will occasionally use variables names such as **i**, **j**, **k**, **x**, and **y** in the code snippets for brevity. These names also provide a generic tone to the code snippets.

descriptive names

the \$ character



Tip

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



MyProgrammingLab™

2.4 Which of the following identifiers are valid? Which are Java keywords?

miles, **Test**, **++**, **--a**, **4#R**, **\$4**, **#44**, **apps**
class, **public**, **int**, **x**, **y**, **radius**

2.5 Variables



Variables are used to represent values that may be changed in the program.

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

why called variables?

the program in Listing 2.2, **radius** and **area** are variables of the 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, in the following code, **radius** is initially **1.0** (line 2) and then changed to **2.0** (line 7), and **area** is set to **3.14159** (line 3) and then reset to **12.56636** (line 8).

```

1 // Compute the first area
2 radius = 1.0;                                radius: 1.0
3 area = radius * radius * 3.14159;            area: 3.14159
4 System.out.println("The area is " + area + " for radius " + radius);
5
6 // Compute the second area
7 radius = 2.0;                                radius: 2.0
8 area = radius * radius * 3.14159;            area: 12.56636
9 System.out.println("The area is " + area + " for radius " + radius);

```

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:

declare variable

```

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

```

These examples use the data types **int** and **double**. 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
```

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

```
int count = 1;
```

This is equivalent to the next two statements:

```

int count;
count = 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.

scope of a variable

Every variable has a scope. The *scope of a variable* is the part of the program where the variable can be referenced. The rules that define the scope of a variable will be introduced gradually later in the book. For now, all you need to know is that a variable must be declared and initialized before it can be used. Consider the following code:

```
int interestRate = 0.05
int interest = interestrate * 45
```

This code is wrong, because `interestRate` is assigned a value `0.05`, but `interestrate` has not been declared and initialized. Java is case sensitive, so it considers `interestRate` and `interestrate` to be two different variables.

2.6 Assignment Statements and Assignment Expressions



An assignment statement designates a value for a variable. An assignment statement can be used as an expression in Java.

assignment statement
assignment operator

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;
```

expression

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

```
int y = 1;           // Assign 1 to variable y
double radius = 1.0; // Assign 1.0 to variable radius
int x = 5 * (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
```

You can use a variable in an expression. A variable can also be used in both sides of the `=` operator. 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, you must place the variable name to the left of the assignment operator. Thus, the following statement is wrong:

```
1 = x; // 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 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:

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

assignment expression

which is equivalent to

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

If a value is assigned to multiple variables, you can use this syntax:

```
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 Section 2.15.

2.7 Named Constants

A named constant is an identifier that represents a permanent value.

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, π 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:

```
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);
    }
}
```



constant

final keyword

There are three benefits of using constants: (1) You don't have to repeatedly type the same value if it is used multiple times; (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; and (3) a descriptive name for a constant makes the program easy to read.

benefits of constants

2.8 Naming Conventions



Sticking with the Java naming conventions makes your programs easy to read and avoids errors.

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

name variables and methods

- 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 `showMessageDialog`.

name classes

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

name constants

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

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



Caution

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

name classes



MyProgrammingLab™

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

2.6 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`

2.7 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 `KILOMETERS_PER_MILE` with value `1.609`.

Step 3: Declare a `double` variable named `kilometers`, multiply `miles` and `KILOMETERS_PER_MILE`, and assign the result to `kilometers`.

Step 4: Display `kilometers` to the console.

What is `kilometers` after Step 4?

2.9 Numeric Data Types and Operations



Java has six numeric types for integers and floating-point numbers with operators `+`, `-`, ``, `/`, and `%`.*

2.9.1 Numeric Types

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 and operators.

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 to 2^7-1 (−128 to 127)	8-bit signed	byte type
short	-2^{15} to $2^{15}-1$ (−32768 to 32767)	16-bit signed	short type
int	-2^{31} to $2^{31}-1$ (−2147483648 to 2147483647)	32-bit signed	int type
long	-2^{63} to $2^{63}-1$ (i.e., −9223372036854775808 to 9223372036854775807)	64-bit signed	long type
float	Negative range: $-3.4028235\text{E}+38$ to $-1.4\text{E}-45$ Positive range: $1.4\text{E}-45$ to $3.4028235\text{E}+38$	32-bit IEEE 754	float type
double	Negative range: $-1.7976931348623157\text{E}+308$ to $-4.9\text{E}-324$ Positive range: $4.9\text{E}-324$ to $1.7976931348623157\text{E}+308$	64-bit IEEE 754	double type



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 uses 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 floating-point values, which are listed in Appendix E.

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

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* and **float** as *single precision*. Normally you should use the **double** type, because it is more accurate than the **float** type.



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** will be too large for an **int** value.

```
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** will be 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.

what is underflow?

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 don't need to be concerned about underflow.

2.9.2 Numeric Operators

operators +, -, *, /, %

operands

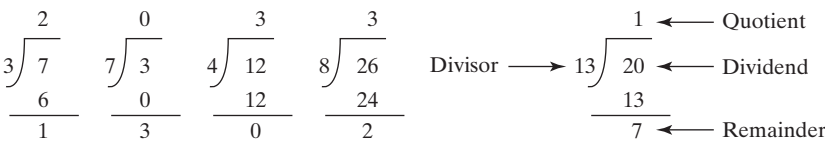
The operators for numeric data types include the standard arithmetic operators: addition (+), subtraction (-), multiplication (*), division (/), and remainder (%), as shown in Table 2.3. The *operands* are the values operated by an operator.

TABLE 2.3 Numeric Operators

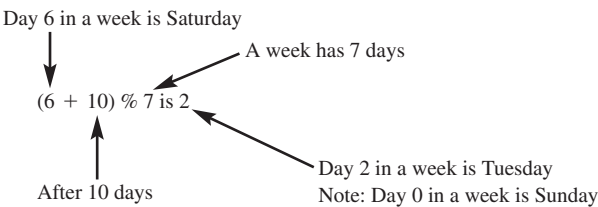
Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 - 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

integer division

When both operands of a division are integers, the result of the division is an integer and 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**.
The **%** operator, known as *remainder* or *modulo* operator, yields the remainder after division. The operand on the left is the dividend and the operand on the right is the divisor. Therefore, **7 % 3** yields **1**, **3 % 7** yields **3**, **12 % 4** yields **0**, **26 % 8** yields **2**, and **20 % 13** yields **7**.



The **%** operator is often used for positive integers, but it can also be used 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**. Thus, 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:



The program in Listing 2.4 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 }
```

import Scanner

create a Scanner

read an integer

divide remainder

Enter an integer for seconds: 500

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 10 obtains the minutes using `seconds / 60`. Line 11 (`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,

floating-point approximation

```
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 0.09999999999999998, not 0.1. Integers are stored precisely. Therefore, calculations with integers yield a precise integer result.

2.9.3 Exponent Operations

`Math.pow(a, b)` method

The **Math.pow(a, b)** method can be used to compute a^b . The **pow** method is defined in the **Math** class in the Java API. You invoke the method using the syntax **Math.pow(a, b)** (i.e., **Math.pow(2, 3)**), which returns the result of a^b (2^3). Here **a** and **b** are parameters for the **pow** method and the numbers **2** and **3** are actual values used to invoke the method. For example,

```
System.out.println(Math.pow(2, 3)); // Displays 8.0
System.out.println(Math.pow(4, 0.5)); // Displays 2.0
System.out.println(Math.pow(2.5, 2)); // Displays 6.25
System.out.println(Math.pow(2.5, -2)); // Displays 0.16
```

Chapter 5 introduces more details on methods. For now, all you need to know is how to invoke the **pow** method to perform the exponent operation.



MyProgrammingLab™

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

2.9 Show the result of the following remainders.

```
56 % 6
78 % -4
-34 % 5
-34 % -5
5 % 1
1 % 5
```

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

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

2.12 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);
```

2.13 Write a statement to display the result of $2^{3.5}$.

2.14 Suppose **m** and **r** are integers. Write a Java expression for mr^2 to obtain a floating-point result.

2.10 Numeric Literals



literal

A *literal* is a constant value that appears directly in a program.

For example, **34** and **0.305** are literals in the following statements:

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

2.10.1 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. For example, to write integer **2147483648** in a Java program, you have to write it as **2147483648L** or **2147483648l**, because **2147483648** exceeds the range for the **int** value. **L** is preferred because **l** (lowercase **L**) can easily be confused with 1 (the digit one).

long type

**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**.

octal and hex literals

```
System.out.println(0x FFFF);
```

Hexadecimal numbers, binary numbers, and octal numbers are introduced in Appendix F.

2.10.2 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 f or F

suffix d or D

**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**.

2.10.3 Scientific Notation

Floating-point literals can be written in scientific notation in the form of $a \times 10^b$. For example, the scientific notation for 123.456 is 1.23456×10^2 and for 0.0123456 is 1.23456×10^{-2} . A special syntax is used to write scientific notation numbers. For example, 1.23456×10^2 is written as **1.23456E2** or **1.23456E+2** and 1.23456×10^{-2} as **1.23456E-2**. **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 internally. 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?

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

12.3, **12.3e+2**, **23.4e-2**, **-334.4**, **20.5**, **39F**, **40D**

2.16 Which of the following are the same as **52.534**?

5.2534e+1, **0.52534e+2**, **525.34e-1**, **5.2534e+0**



Check
Point
MyProgrammingLab™

2.1.1 Evaluating Expressions and Operator Precedence



Key
Point

Java expressions are evaluated in the same way as arithmetic expressions.

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

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

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)
```

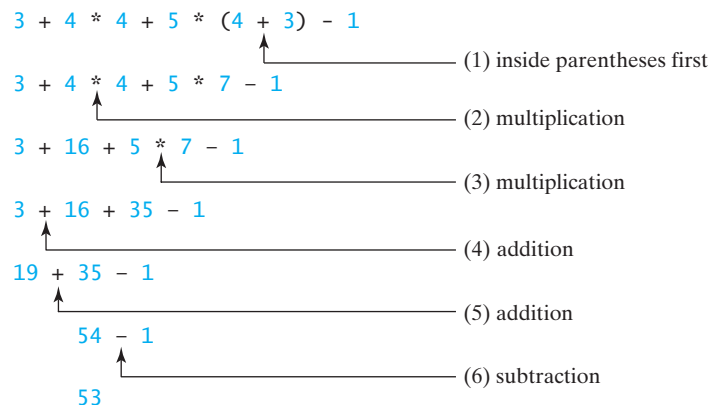
evaluating an expression

operator precedence rule

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. When more than one operator is used in an expression, the following operator precedence rule is used to determine the order of evaluation.

- Multiplication, division, and remainder operators are applied first. 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:



Listing 2.5 gives a program that converts a Fahrenheit degree to Celsius using the formula $celsius = (\frac{5}{9})(fahrenheit - 32)$.

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);
12     System.out.println("Fahrenheit " + fahrenheit + " is " +
13         celsius + " in Celsius");
14 }
15 }

```

divide

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

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

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



MyProgrammingLab™

2.12 Case Study: Displaying the Current Time

You can invoke `System.currentTimeMillis()` to return 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.

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*. The epoch is the point when time starts, and 1970 was the year when the UNIX operating system was formally introduced.



VideoNote

Use operators / and %

`currentTimeMillis`
UNIX epoch

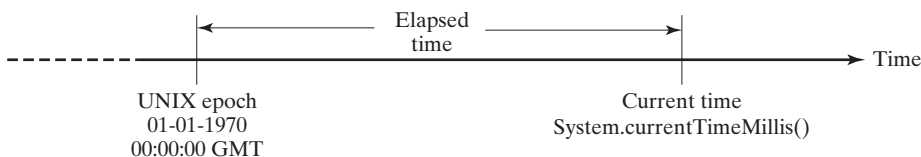


FIGURE 2.2 The `System.currentTimeMillis()` returns the number of milliseconds since the UNIX epoch.

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, January 1, 1970, in `totalMilliseconds` by invoking `System.currentTimeMillis()` (e.g., `1203183068328` milliseconds).

2. Obtain the total seconds `totalSeconds` by dividing `totalMilliseconds` by `1000` (e.g., `1203183068328` milliseconds / `1000` = `1203183068` seconds).
3. Compute the current second from `totalSeconds % 60` (e.g., `1203183068` seconds % `60` = `8`, which is the current second).
4. Obtain the total minutes `totalMinutes` by dividing `totalSeconds` by `60` (e.g., `1203183068` 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) {
totalMilliseconds	3	// Obtain the total milliseconds since midnight, Jan 1, 1970
	4	long totalMilliseconds = System.currentTimeMillis();
	5	
totalSeconds	6	// Obtain the total seconds since midnight, Jan 1, 1970
	7	long totalSeconds = totalMilliseconds / 1000 ;
	8	
currentSecond	9	// Compute the current second in the minute in the hour
	10	long currentSecond = totalSeconds % 60 ;
	11	
totalMinutes	12	// Obtain the total minutes
	13	long totalMinutes = totalSeconds / 60 ;
	14	
currentMinute	15	// Compute the current minute in the hour
	16	long currentMinute = totalMinutes % 60 ;
	17	
totalHours	18	// Obtain the total hours
	19	long totalHours = totalMinutes / 60 ;
	20	
currentHour	21	// Compute the current hour
	22	long currentHour = totalHours % 24 ;
	23	
preparing output	24	// Display results
	25	System.out.println("Current time is " + currentHour + ":"
	26	+ currentMinute + ":" + currentSecond + " GMT");
	27	}
	28	}



Current time is 17:31:8 GMT

Line 4 invokes `System.currentTimeMillis()` to obtain the current time in milliseconds as a long value. Thus, all the variables are declared as the long type in this program. The seconds, minutes, and hours are extracted from the current time using the `/` and `%` operators (lines 6–22).

	line#	4	7	10	13	16	19	22
variables								
totalMilliseconds		1203183068328						
totalSeconds			1203183068					
currentSecond				8				
totalMinutes					20053051			
currentMinute						31		
totalHours							334217	
currentHour								17

In the sample run, a single digit 8 is displayed for the second. The desirable output would be 08. This can be fixed by using a function that formats a single digit with a prefix 0 (see Exercise 5.37).

2.13 Augmented Assignment Operators

The operators +, -, *, /, and % can be combined with the assignment operator to form augmented 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 increases the variable count by 1:

```
count = count + 1;
```

Java allows you to combine assignment and addition operators using an augmented (or compound) assignment operator. For example, the preceding statement can be written as:

```
count += 1;
```

The += is called the *addition assignment operator*. Table 2.4 shows other augmented assignment operators.

TABLE 2.4 Augmented Assignment 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 augmented assignment 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
```

2.14 Increment and Decrement Operators



The *increment* (`++`) and *decrement* (`--`) operators are for incrementing and decrementing a variable by 1.

increment operator (`++`)
decrement operator (`--`)

The `++` and `--` are two 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 in many programming tasks. 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
```

postincrement
postdecrement

`i++` is pronounced as `i` plus plus and `i--` as `i` minus minus. These operators are known as *postfix increment* (or postincrement) and *postfix decrement* (or postdecrement), because the operators `++` and `--` are placed after the variable. These operators can also be placed before the variable. For example,

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

preincrement
predecrement

`++i` increments `i` by 1 and `--j` decrements `j` by 1. These operators are known as *prefix increment* (or preincrement) and *prefix decrement* (or predecrement).

As you see, the effect of `i++` and `++i` or `i--` and `--i` are the same in the preceding examples. However, their effects are different when they are used in statements that do more than just increment and decrement. Table 2.5 describes their differences and gives examples.

TABLE 2.5 Increment and Decrement Operators

Operator	Name	Description	Example (assume <code>i = 1</code>)
<code>++var</code>	preincrement	Increment <code>var</code> by 1, and use the new <code>var</code> value in the statement	<code>int j = ++i;</code> // <code>j</code> is 2, <code>i</code> is 2
<code>var++</code>	postincrement	Increment <code>var</code> by 1, but use the original <code>var</code> value in the statement	<code>int j = i++;</code> // <code>j</code> is 1, <code>i</code> is 2
<code>--var</code>	predecrement	Decrement <code>var</code> by 1, and use the new <code>var</code> value in the statement	<code>int j = --i;</code> // <code>j</code> is 0, <code>i</code> is 0
<code>var--</code>	postdecrement	Decrement <code>var</code> by 1, and use the original <code>var</code> value in the statement	<code>int j = i--;</code> // <code>j</code> is 1, <code>i</code> is 0

Here are additional examples to illustrate the differences between the prefix form of `++` (or `--`) and the postfix form of `++` (or `--`). Consider the following code:

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

Same effect as

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

```
System.out.print("i is " + i
    + ", newNum is " + newNum);
```

i is 11, newNum is 100



In this case, `i` is incremented by `1`, then the *old* value of `i` is 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;
```

```
System.out.print("i is " + i
    + ", newNum is " + newNum);
```

i is 11, newNum is 110



`i` is incremented by `1`, and the new value of `i` is 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`.



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`.

2.18 Which of these statements are true?

- Any expression can be used as a statement.
- The expression `x++` can be used as a statement.
- The statement `x = x + 5` is also an expression.
- The statement `x = y = x = 0` is illegal.

2.19 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++;
```



Check
Point
MyProgrammingLab™

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

2.15 Numeric Type Conversions



Floating-point numbers can be converted into integers using explicit casting.

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 a 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 type with a small range to a type with a larger range is known as *widening a type*. Casting a type with a large range to a type with a smaller range is known as *narrowing a type*. Java will automatically widen a type, but you must narrow a type explicitly.

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

```
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.

casting
widening a type
narrowing a type

possible loss of precision



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. However, be careful when using casting, as loss of information might lead to inaccurate results.



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 still 4.5
```



Note

In Java, an augmented expression of the form **x1 op= x2** is implemented as **x1 = (T)(x1 op x2)**, where **T** is the type for **x1**. Therefore, the following code is correct.

```
int sum = 0;
sum += 4.5; // sum becomes 4 after this statement
```

`sum += 4.5` is equivalent to `sum = (int)(sum + 4.5)`.

casting in an augmented
expression

**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 (see Section 2.10, Numeric Literals).

The program in Listing 2.7 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 }
```

casting

Enter purchase amount: 197.55
Sales tax is \$11.85



line#	purchaseAmount	tax	output
8	197.55		
10		11.853	
11			11.85



The 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.

- 2.21** Can different types of numeric values be used together in a computation?
- 2.22** What does an explicit casting from a **double** to an **int** do with the fractional part of the **double** value? Does casting change the variable being cast?
- 2.23** Show the following output:

```
float f = 12.5F;
int i = (int)f;
```

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

2.24 If you change `(int)(tax * 100) / 100.0` to `(int)(tax * 100) / 100` in line 11 in Listing 2.7, what will be the output for the input purchase amount of **197.556**?

2.16 Software Development Process



The software development life cycle is a multi-stage process that includes requirements specification, analysis, design, implementation, testing, deployment, and maintenance.



VideoNote
Software development process

Developing a software product is an engineering process. Software products, no matter how large or how small, have the same life cycle: requirements specification, analysis, design, implementation, testing, deployment, and maintenance, as shown in Figure 2.3.

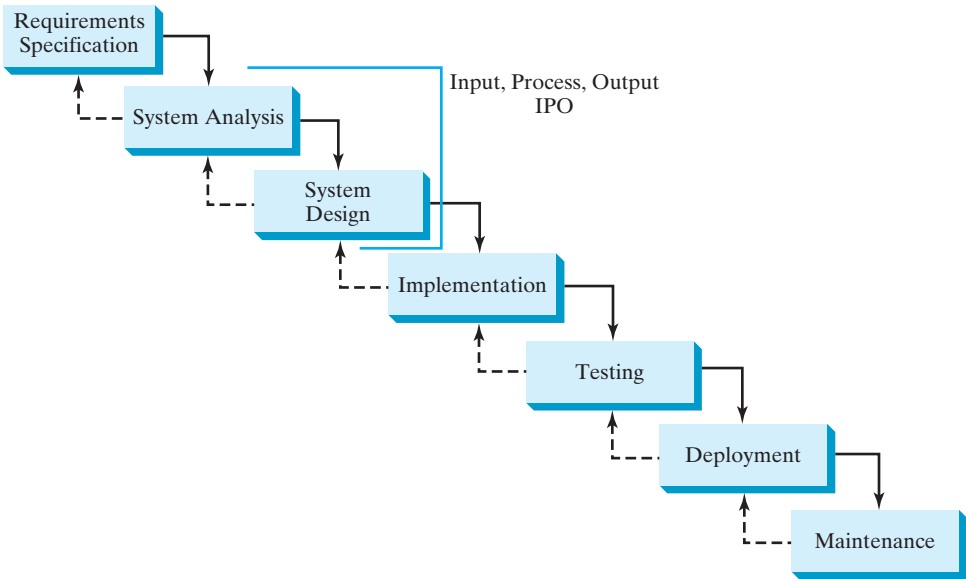


FIGURE 2.3 At any stage of the software development life cycle, it may be necessary to go back to a previous stage to correct errors or deal with other issues that might prevent the software from functioning as expected.

requirements specification

Requirements specification is a formal process that seeks to understand the problem that the software will address and to document in detail what the software system needs to do. This phase involves close interaction between users and developers. Most of the examples in this book are simple, and their requirements are clearly stated. In the real world, however, problems are not always well defined. Developers need to work closely with their customers (the individuals or organizations that will use the software) and study the problem carefully to identify what the software needs to do.

system analysis

System analysis seeks to analyze the data flow and to identify the system’s input and output. When you do analysis, it helps to identify what the output is first, and then figure out what input data you need in order to produce the output.

system design

System design is to design a process for obtaining the output from the input. This phase involves the use of many levels of abstraction to break down the problem into manageable components and design strategies for implementing each component. You can view each component as a subsystem that performs a specific function of the system. The essence of system analysis and design is input, process, and output (IPO).

IPO

Implementation involves translating the system design into programs. Separate programs are written for each component and then integrated to work together. This phase requires the use of a programming language such as Java. The implementation involves coding, self-testing, and debugging (that is, finding errors, called *bugs*, in the code).

implementation

Testing ensures that the code meets the requirements specification and weeds out bugs. An independent team of software engineers not involved in the design and implementation of the product usually conducts such testing.

testing

Deployment makes the software available for use. Depending on the type of the software, it may be installed on each user's machine or installed on a server accessible on the Internet.

deployment

Maintenance is concerned with updating and improving the product. A software product must continue to perform and improve in an ever-evolving environment. This requires periodic upgrades of the product to fix newly discovered bugs and incorporate changes.

maintenance

To see the software development process in action, we will now create a program that computes loan payments. The loan can be a car loan, a student loan, or a home mortgage loan. For an introductory programming course, we focus on requirements specification, analysis, design, implementation, and testing.



VideoNote

Compute loan payments

Stage 1: Requirements Specification

The program must satisfy the following requirements:

- It must let the user enter the interest rate, the loan amount, and the number of years for which payments will be made.
- It must compute and display the monthly payment and total payment amounts.

Stage 2: System Analysis

The output is the monthly payment and total payment, which can be obtained using the following formulas:

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

$$\text{totalPayment} = \text{monthlyPayment} \times \text{numberOfYears} \times 12$$

So, the input needed for the program is the monthly interest rate, the length of the loan in years, and the loan amount.



Note

The requirements specification says that the user must enter the annual interest rate, the loan amount, and the number of years for which payments will be made. During analysis, however, it is possible that you may discover that input is not sufficient or that some values are unnecessary for the output. If this happens, you can go back and modify the requirements specification.



Note

In the real world, you will work with customers from all walks of life. You may develop software for chemists, physicists, engineers, economists, and psychologists, and of course you will not have (or need) complete knowledge of all these fields. Therefore, you don't have to know how formulas are derived, but given the monthly interest rate, the number of years, and the loan amount, you can compute the monthly payment in this program. You will, however, need to communicate with customers and understand how a mathematical model works for the system.

Stage 3: System Design

During system design, you identify the steps in the program.

- Step 1. Prompt the user to enter the annual interest rate, the number of years, and the loan amount.
- Step 2. The input for the annual interest rate is a number in percent format, such as 4.5%. The program needs to convert it into a decimal by dividing it by 100. To obtain the monthly interest rate from the annual interest rate, divide it by 12, since a year has 12 months. So, to obtain the monthly interest rate in decimal format, you need to divide the annual interest rate in percentage by 1200. For example, if the annual interest rate is 4.5%, then the monthly interest rate is $4.5/1200 = 0.00375$.
- Step 3. Compute the monthly payment using the preceding formula.
- Step 4. Compute the total payment, which is the monthly payment multiplied by 12 and multiplied by the number of years.
- Step 5. Display the monthly payment and total payment.

Stage 4: Implementation

`Math.pow(a, b)` method

Implementation is also known as *coding* (writing the code). In the formula, you have to compute $(1 + \text{monthlyInterestRate})^{\text{numberOfYears} \times 12}$, which can be obtained using `Math.pow(1 + monthlyInterestRate, numberOfYears * 12)`.

Listing 2.8 gives the complete program.

LISTING 2.8 `ComputeLoan.java`

```

import class      1  import java.util.Scanner;
                  2
                  3  public class ComputeLoan {
                  4      public static void main(String[] args) {
                  5          // Create a Scanner
create a Scanner  6          Scanner input = new Scanner(System.in);
                  7
                  8          // Enter annual interest rate in percentage, e.g., 7.25%
enter interest rate 9          System.out.print("Enter annual interest rate, e.g., 7.25%: ");
                  10         double annualInterestRate = input.nextDouble();
                  11
                  12         // Obtain monthly interest rate
                  13         double monthlyInterestRate = annualInterestRate / 1200;
                  14
                  15         // Enter number of years
enter years       16         System.out.print(
                  17             "Enter number of years as an integer, e.g., 5: ");
                  18         int numberOfYears = input.nextInt();
                  19
                  20         // Enter loan amount
enter loan amount 21         System.out.print("Enter loan amount, e.g., 120000.95: ");
                  22         double loanAmount = input.nextDouble();
                  23
monthlyPayment    24         // Calculate payment
                  25         double monthlyPayment = loanAmount * monthlyInterestRate / (1
                  26             - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
totalPayment      27         double totalPayment = monthlyPayment * numberOfYears * 12;
                  28
                  29         // Display results

```



```
30     System.out.println("The monthly payment is $" +
31         (int)(monthlyPayment * 100) / 100.0);
32     System.out.println("The total payment is $" +
33         (int)(totalPayment * 100) / 100.0);
34 }
35 }
```

casting

casting

Enter annual interest rate, e.g., 5.75%: 5.75

Enter number of years as an integer, e.g., 5: 15

Enter loan amount, e.g., 120000.95: 250000

The monthly payment is \$2076.02

The total payment is \$373684.53



line#	10	13	18	22	25	27
variables						
annualInterestRate	5.75					
monthlyInterestRate		0.00479166666666				
numberOfYears			15			
loanAmount				250000		
monthlyPayment					2076.0252175	
totalPayment						373684.539



Line 10 reads the annual interest rate, which is converted into the monthly interest rate in line 13.

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 points.

The program uses the `Scanner` class, imported in line 1. The program also uses the `Math` class, and you might be wondering why that class isn't imported into the program. The `Math` class is in the `java.lang` package, and all classes in the `java.lang` package are implicitly imported. Therefore, you don't need to explicitly import the `Math` class.

java.lang package

Stage 5: Testing

After the program is implemented, test it with some sample input data and verify whether the output is correct. Some of the problems may involve many cases, as you will see in later chapters. For these types of problems, you need to design test data that cover all cases.



Tip

The system design phase in this example identified several steps. It is a good approach to developing and testing these steps incrementally by adding them one at a time. This approach makes it much easier to pinpoint problems and debug the program.

incremental development and testing



Check
Point

MyProgrammingLab™

2.25 How would you write the following arithmetic expression?

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

2.17 Character Data Type and Operations



A character data type represents a single character.

char type

In addition to processing numeric values, you can process characters in Java. 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 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**.

char literal



Caution

A string literal must be enclosed in quotation marks (" "). A character literal is a single character enclosed in single quotation marks (' '). Therefore, "A" is a string, but 'A' is a character.

2.17.1 Unicode and ASCII code

encoding

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*.

Unicode

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 the 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.

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**. Hexadecimal numbers are introduced in Appendix F, Number Systems. For example, the English word **welcome** is translated 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 \u03b2 \u03b3 \u03b4",
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 α β γ 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 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:

```

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

Both statements assign character **A** to the `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 ch = 'a';
System.out.println(++ch);
```

`char` increment and decrement

2.17.2 Escape 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 compile 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 the characters.

To overcome this problem, Java uses a special notation to represent special characters, as shown in Table 2.6. This special notation, called an *escape character*, consists of a backslash (`\`) followed by a character or a character sequence. For example, `\t` is an escape character for the Tab character and an escape character such as `\u03b1` is used to represent a Unicode. The symbols in an escape character are interpreted as a whole rather than individually.

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"
```

Note that the symbols `\` and `"` together represent one character.

2.17.3 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
```

TABLE 2.6 Escape Characters

Escape Character	Name	Unicode Code	Decimal Value
<code>\b</code>	Backspace	<code>\u0008</code>	8
<code>\t</code>	Tab	<code>\u0009</code>	9
<code>\n</code>	Linefeed	<code>\u000A</code>	10
<code>\f</code>	Formfeed	<code>\u000C</code>	12
<code>\r</code>	Carriage Return	<code>\u000D</code>	13
<code>\\</code>	Backslash	<code>\u005C</code>	92
<code>\"</code>	Double Quote	<code>\u0022</code>	34

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 `\uFFFF` cannot fit into a byte:

```
byte b = '\uFFFF';
```

To force this assignment, use explicit casting, as follows:

```
byte b = (byte)'\uFFFF';
```

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.

numeric operators on
characters



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

```
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); // j is the Unicode for character c
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'))`.

2.17.4 Case Study: Counting Monetary Units

Suppose you want to develop a program that changes 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 the maximum number of dollars, quarters, dimes, nickels, and pennies, in this order, to result in the minimum number of coins, as shown in the sample run.

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

```

1  import java.util.Scanner;                                import class
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(

```

66 Chapter 2 Elementary Programming

```

10      "Enter an amount, 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, for example, 11.56:

Your amount 11.56 consists of

- 11 dollars
- 2 quarters
- 0 dimes
- 1 nickels
- 1 pennies

[illegible]

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 remaining amount.

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 remaining amount and obtains a new 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 `1002.9999999999999`. 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 2.24).

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).

2.26 Use print statements to find out the ASCII code for `'1'`, `'A'`, `'B'`, `'a'`, and `'b'`. Use print statements to find out the character for the decimal codes `40`, `59`, `79`, `85`, and `90`. Use print statements to find out the character for the hexadecimal code `40`, `5A`, `71`, `72`, and `7A`.



MyProgrammingLab™

2.27 Which of the following are correct literals for characters?

`'1'`, `'\u345dE'`, `'\u3fFa'`, `'\b'`, `'\t'`

2.28 How do you display the characters `\` and `"`?

2.29 Evaluate the following:

```
int i = '1';
int j = '1' + '2' * ('4' - '3') + 'b' / 'a';
int k = 'a';
char c = 90;
```

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

```
char c = 'A';
int 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;
```


2.31 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);
    }
}
```

2.18 The String Type



Key
Point

A string is a sequence of characters.

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 the value "Welcome to Java".

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

String is 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:

```
// 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 augmented += 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".

If **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));
```

concatenate strings and
numbers

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:

```
Scanner input = new Scanner(System.in);
System.out.println("Enter three words separated by spaces: ");
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);
```

```
Enter three words separated by spaces: Welcome to Java ↵ Enter
s1 is Welcome
s2 is to
s3 is Java
```



The `next()` method reads a string that ends with a whitespace character. The characters ' ', `\t`, `\f`, `\r`, or `\n` are known as *whitespace characters*.

whitespace character

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 line: ");
String s = input.nextLine();
System.out.println("The line entered is " + s);
```

```
Enter a line: Welcome to Java ↵ Enter
The line entered is Welcome to Java
```



Important Caution

To avoid input errors, do not use `nextLine()` after `nextByte()`, `nextShort()`, `nextInt()`, `nextLong()`, `nextFloat()`, `nextDouble()`, or `next()`. The reasons will be explained in Section 14.11.3, "How Does **Scanner** Work?"

avoid input errors

2.32 Show the output of the following statements (write a program to verify your results):

```
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);
```



Check
Point

MyProgrammingLab™

2.33 Evaluate the following expressions (write a program to verify your results):

```
1 + "Welcome " + 1 + 1
1 + "Welcome " + (1 + 1)
1 + "Welcome " + ('\u0001' + 1)
1 + "Welcome " + 'a' + 1
```

2.19 Getting Input from Input Dialogs



JOptionPane class

An input dialog box prompts the user to enter an input graphically.

You can obtain input from the console. Alternatively, you can obtain input from an input dialog box by invoking the **JOptionPane.showInputDialog** method, as shown in Figure 2.4.

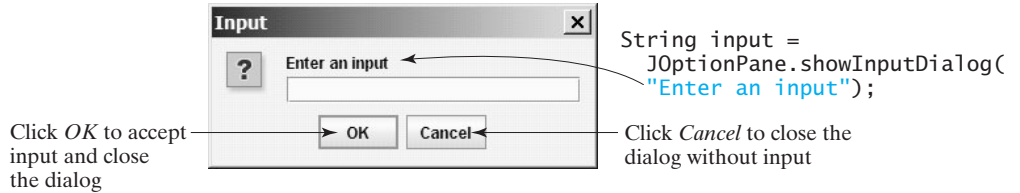


FIGURE 2.4 The input dialog box enables the user to enter a string.

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 close the dialog box. The input is returned from the method as a string.

showInputDialog method

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

One is to use a statement like this one:

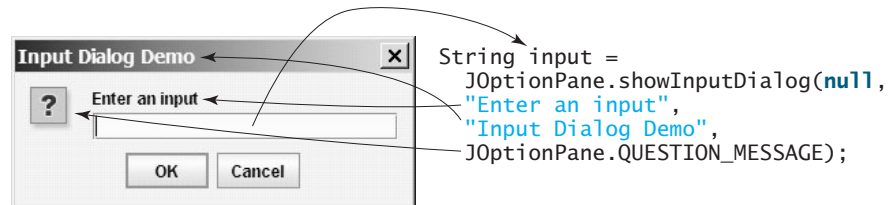
```
JOptionPane.showInputDialog(x);
```

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.



2.19.1 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.

Integer.parseInt method

To convert a string into an **int** value, use the **Integer.parseInt** method, as follows:

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

where **intString** is a numeric string such as **123**.

Double.parseDouble method

To convert a string into a **double** value, use the **Double.parseDouble** method, as follows:

```
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.

2.19.2 Using Input Dialog Boxes

Having learned how to read input from an input dialog box, you can rewrite the program in Listing 2.8, `ComputeLoan.java`, to read from input dialog boxes rather than from the console. Listing 2.11 gives the complete program. Figure 2.5 shows a sample run of the program.

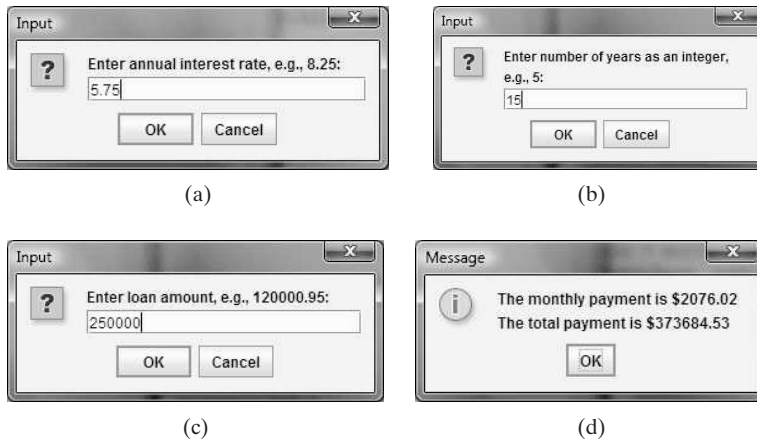


FIGURE 2.5 The program accepts the annual interest rate (a), number of years (b), and loan amount (c), then displays the monthly payment and total payment (d).

LISTING 2.11 `ComputeLoanUsingInputDialog.java`

```

1  import javax.swing.JOptionPane;
2
3  public class ComputeLoanUsingInputDialog {
4      public static void main(String[] args) {
5          // Enter annual interest rate
6          String annualInterestRateString = JOptionPane.showInputDialog(    enter interest rate
7              "Enter annual interest rate, for example, 8.25:");
8
9          // Convert string to double
10         double annualInterestRate =                                     convert string to double
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, for 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     }

```

```

monthlyPayment    30      // Calculate payment
                  31      double monthlyPayment = loanAmount * monthlyInterestRate / (1
totalPayment      32      - 1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12));
                  33      double totalPayment = monthlyPayment * numberOfYears * 12;
                  34
preparing output   35      // Format to keep two digits after the decimal point
                  36      monthlyPayment = (int)(monthlyPayment * 100) / 100.0;
                  37      totalPayment = (int)(totalPayment * 100) / 100.0;
                  38
                  39      // Display results
                  40      String output = "The monthly payment is $" + monthlyPayment +
                  41      "\n\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.

JOptionPane or Scanner?



Pedagogical Note

For obtaining input you can use either `JOptionPane` or `Scanner`—whichever is more convenient. For consistency and simplicity, the examples in this book use `Scanner` for getting input. You can easily revise the examples using `JOptionPane` for getting input.



Check
Point

MyProgrammingLab™

- 2.34** Why do you have to import `JOptionPane` but not the `Math` class?
- 2.35** How do you prompt the user to enter an input using a dialog box?
- 2.36** How do you convert a string to an integer? How do you convert a string to a double?

KEY TERMS

algorithm	34	increment operator (++)	54
assignment operator (=)	42	incremental development	
assignment statement	42	and testing	37
byte type	45	int type	45
casting	56	IPO	39
char type	62	literal	48
constant	43	long type	45
data type	35	narrowing (of types)	56
declare variables	35	operands	46
decrement operator (--)	54	operator	46
double type	45	overflow	45
encoding	62	postdecrement	54
escape character	63	postincrement	54
expression	42	predecrement	54
final keyword	43	preincrement	54
float type	45	primitive data type	35
floating-point number	35	pseudocode	34
identifier	40	requirements specification	58

scope of a variable	42	Unicode	62
short type	45	UNIX epoch	51
supplementary Unicode	62	variable	35
system analysis	58	whitespace character	69
system design	58	widening (of types)	56
underflow	46		

CHAPTER SUMMARY

1. *Identifiers* are names for naming elements such as variables, constants, methods, classes, packages in a program.
2. An identifier is a sequence of characters that consists of letters, digits, underscores (`_`), and dollar signs (`$`). An identifier must start with a letter or an underscore. It cannot start with a digit. An identifier cannot be a reserved word. An identifier can be of any length.
3. *Variables* are used to store data in a program.
4. To declare a variable is to tell the compiler what type of data a variable can hold.
5. In Java, the equal sign (`=`) is used as the *assignment operator*.
6. A variable declared in a method must be assigned a value before it can be used.
7. A *named constant* (or simply a *constant*) represents permanent data that never changes.
8. A named constant is declared by using the keyword `final`.
9. Java provides four integer types (`byte`, `short`, `int`, and `long`) that represent integers of four different sizes.
10. Java provides two *floating-point types* (`float` and `double`) that represent floating-point numbers of two different precisions.
11. Java provides *operators* that perform numeric operations: `+` (addition), `-` (subtraction), `*` (multiplication), `/` (division), and `%` (remainder).
12. Integer arithmetic (`/`) yields an integer result.
13. The numeric operators in a Java expression are applied the same way as in an arithmetic expression.
14. Java provides the augmented assignment operators `+=` (addition assignment), `-=` (subtraction assignment), `*=` (multiplication assignment), `/=` (division assignment), and `%=` (remainder assignment).
15. The *increment operator* (`++`) and the *decrement operator* (`--`) increment or decrement a variable by 1.
16. When evaluating an expression with values of mixed types, Java automatically converts the operands to appropriate types.
17. You can explicitly convert a value from one type to another using the `(type)value` notation.

18. 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*.
19. 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*.
20. Widening a type can be performed automatically without explicit casting. Narrowing a type must be performed explicitly.
21. The character type `char` represents a single character.
22. An *escape character* is a notation for representing a special character. An escape character consists of a backslash (`\`) followed by a character or a character sequence.
23. The characters `' '`, `\t`, `\f`, `\r`, and `\n` are known as the whitespace characters.
24. In computer science, midnight of January 1, 1970, is known as the *UNIX epoch*.

TEST QUESTIONS

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

MyProgrammingLab™

PROGRAMMING EXERCISES

sample runs



Note

You can run all exercises by downloading **exercise9e.zip** from www.cs.armstrong.edu/liang/intro9e/exercise9e.zip and use the command `java -cp exercise9e.zip Exercisei_j` to run `Exercisei_j`. For example, to run Exercise 2.1, use

```
java -cp exercise9e.zip Exercise02_01
```

This will give you an idea how the program runs.

learn from examples



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.

document analysis and design



Pedagogical Note

Instructors may ask you to document your analysis and design for selected exercises. Use your 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.

Sections 2.2–2.12

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

$$\text{fahrenheit} = (9 / 5) * \text{celsius} + 32$$

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 ↵ Enter
43 Celsius is 109.4 Fahrenheit
```



- 2.2** (*Compute the volume of a cylinder*) Write a program that reads in the radius and length of a cylinder and computes the area and 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 ↵ Enter
The area is 95.0331
The volume is 1140.4
```



- 2.3** (*Convert 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: 16.5 ↵ Enter
16.5 feet is 5.0325 meters
```



- 2.4** (*Convert 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 ↵ Enter
55.5 pounds is 25.197 kilograms
```



- *2.5** (*Financial application: calculate 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: 10 15 ↵ Enter
The gratuity is $1.5 and total is $11.5
```



- **2.6** (*Sum 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:

```
Enter a number between 0 and 1000: 999 ↵ Enter
The sum of the digits is 27
```



- *2.7** (*Find 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 Enter
1000000000 minutes is approximately 1902 years and 214 days
```

- *2.8** (*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
```

- 2.9** (*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 v_0 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: 5.5 50.9 4.5 Enter
The average acceleration is 10.0889
```

- 2.10** (*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

$$Q = M * (\text{finalTemperature} - \text{initialTemperature}) * 4184$$

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: 55.5 Enter
Enter the initial temperature: 3.5 Enter
Enter the final temperature: 10.5 Enter
The energy needed is 1625484.0
```

- 2.11** (*Population projection*) Rewrite Exercise 1.11 to prompt the user to enter the number of years and displays the population after the number of years. Here is a sample run of the program:

```
Enter the number of years: 5 ↵ Enter
The population in 5 years is 325932970
```



- 2.12** (*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/s²), and displays the minimum runway length. Here is a sample run:

```
Enter speed and acceleration: 60 3.5 ↵ Enter
The minimum runway length for this airplane is 514.286
```



- **2.13** (*Financial application: compound value*) Suppose you save \$100 each month into a savings account with the annual interest rate 5%. Thus, 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 a monthly saving amount and displays 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.)

```
Enter the monthly saving amount: 100 ↵ Enter
After the sixth month, the account value is $608.81
```



- *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 displays the BMI. Note that one pound is 0.45359237 kilograms and one inch is 0.0254 meters. Here is a sample run:



VideoNote
Compute BMI



```
Enter weight in pounds: 95.5 Enter
Enter height in inches: 50 Enter
BMI is 26.8573
```

- *2.15** (*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

$$s = (side1 + side2 + side3)/2;$$

$$area = \sqrt{s(s - side1)(s - side2)(s - side3)}$$

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
```

- 2.16** (*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

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

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



```
Enter the side: 5.5 Enter
The area of the hexagon is 78.5895
```

- *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:

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

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:



```
Enter the temperature in Fahrenheit: 5.3 Enter
Enter the wind speed in miles per hour: 6 Enter
The wind chill index is -5.56707
```

2.18 (*Print 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	15625

2.19 (*Geometry: distance of two points*) Write a program that prompts the user to enter two points (**x1**, **y1**) and (**x2**, **y2**) and displays their distance between them. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Note that you can use **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 between the two points is 8.764131445842194
```



Sections 2.13–2.16

***2.20** (*Financial application: calculate 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. Here is a sample run:

```
Enter balance and interest rate (e.g., 3 for 3%): 1000 3.5 ↵ Enter
The interest is 2.91667
```



***2.21** (*Financial application: calculate 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:

$$\text{futureInvestmentValue} = \text{investmentAmount} \times (1 + \text{monthlyInterestRate})^{\text{numberOfYears} \times 12}$$

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

Here is a sample run:

```
Enter investment amount: 1000 ↵ Enter
Enter annual interest rate in percentage: 3.25 ↵ Enter
Enter number of years: 1 ↵ Enter
Accumulated value is $1043.34
```



Sections 2.17–2.18

- 2.22** (*Random character*) Write a program that displays a random uppercase letter using the `System.currentTimeMillis()` method.
- 2.23** (*Find the character of an ASCII code*) Write a program that receives an ASCII code (an integer between 0 and 127) 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 is E
```

- *2.24** (*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.
- *2.25** (*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%)
```



```
Enter employee's name: Smith Enter
Enter number of hours worked in a week: 10 Enter
Enter hourly pay rate: 6.75 Enter
Enter federal tax withholding rate: 0.20 Enter
Enter state tax withholding rate: 0.09 Enter

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
```

Section 2.19

- *2.26** (*Use input dialog*) Rewrite Listing 2.10, `ComputeChange.java`, using input and output dialog boxes.
- *2.27** (*Financial application: payroll*) Rewrite Exercise 2.25 using GUI input and output dialog boxes.

SELECTIONS

Objectives

- To declare **boolean** variables and write Boolean expressions using comparison operators (§3.2).
- To implement selection control using one-way **if** statements (§3.3).
- To program using one-way **if** statements (**GuessBirthday**) (§3.4).
- To implement selection control using two-way **if-else** statements (§3.5).
- To implement selection control using nested **if** and multi-way **if** statements (§3.6).
- To avoid common errors in **if** statements (§3.7).
- To generate random numbers using the **Math.random()** method (§3.8).
- To program using selection statements for a variety of examples (**SubtractionQuiz**, **BMI**, **ComputeTax**) (§§3.8–3.10).
- To combine conditions using logical operators (**&&**, **||**, and **!**) (§3.11).
- To program using selection statements with combined conditions (**LeapYear**, **Lottery**) (§§3.12–3.13).
- To implement selection control using **switch** statements (§3.14).
- To write expressions using the conditional operator (§3.15).
- To format output using the **System.out.printf** method (§3.16).
- To examine the rules governing operator precedence and associativity (§3.17).
- To get user confirmation using confirmation dialogs (§3.18).
- To apply common techniques to debug errors (§3.19).



3.1 Introduction



The program can decide which statements to execute based on a condition.

problem

If you enter a negative value for **radius** in Listing 2.2, `ComputeAreaWithConsoleInput.java`, the program displays 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?

selection statements

Like all high-level programming languages, Java provides *selection statements*: 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);
}
```

Boolean expression
Boolean value

Selection statements use conditions that are Boolean expressions. A *Boolean expression* is an expression that evaluates to a *Boolean value*: **true** or **false**. We now introduce Boolean types and comparison operators.

3.2 boolean Data Type



A **boolean** data type declares a variable with the value either **true** or **false**.

boolean data type
comparison operators

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).

TABLE 3.1 Comparison Operators

Java Operator	Mathematics Symbol	Name	Example (radius is 5)	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

compare characters



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 Set, to find the order of characters.

== vs. =



Caution

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

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);
```

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** or **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.

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 1 + 7?”, as shown in the sample run in Listing 3.1. After the student types the answer, the program displays a message to indicate whether it is true or false.

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**.

Boolean variable

Boolean literals



VideoNote

Program addition quiz

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);
6         int number2 = (int)(System.currentTimeMillis() / 7 % 10);
7
8         // Create a Scanner
9         Scanner input = new Scanner(System.in);
10
11         System.out.print(
12             "What is " + number1 + " + " + number2 + "? ";
13
14         int answer = input.nextInt();
15
16         System.out.println(
17             number1 + " + " + number2 + " = " + answer + " is " +
18             (number1 + number2 == answer));
19     }
20 }
```

generate number1
generate number2

show question

display result

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			9	
16				4 + 8 = 9 is false





MyProgrammingLab™

- 3.1** List six comparison operators.
- 3.2** Show the printout of the following statements:

```
System.out.println('a' < 'b');
System.out.println('a' <= 'A');
System.out.println('a' > 'b');
System.out.println('a' >= 'A');
System.out.println('a' == 'a');
System.out.println('a' != 'b');
```

- 3.3** 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;
```

3.3 if Statements



An **if** statement executes the statements if the condition is true.

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 make this minor change.

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

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:

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

The flowchart in Figure 3.1 illustrates how Java executes the syntax of an **if** statement. A *flowchart* is a diagram that describes an algorithm or process, showing the steps as boxes of various kinds, and their order by connecting these with arrows. Process operations are represented in these boxes, and arrows connecting them represent the flow of control. A diamond box is used to denote a Boolean condition and a rectangle box is for representing statements.

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 flowchart of the preceding statement is shown in Figure 3.1b. 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).

why if statement?

if statement

flowchart

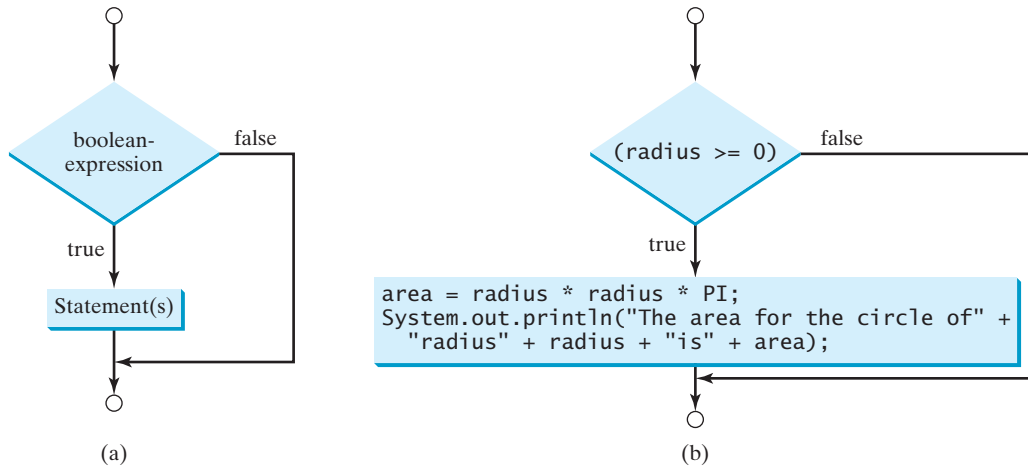


FIGURE 3.1 An **if** statement executes statements if the **boolean-expression** evaluates to **true**.

```
if i > 0 {
    System.out.println("i is positive");
}
```

(a) Wrong

```
if (i > 0) {
    System.out.println("i is positive");
}
```

(b) Correct

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

```
if (i > 0) {
    System.out.println("i is positive");
}
```

(a)

Equivalent

```
if (i > 0)
    System.out.println("i is positive");
```

(b)



Note

Omitting braces makes the code shorter, but it is prone to errors. It is a common mistake to forget the braces when you go back to modify the code that omits the braces.

Omitting braces or not

Listing 3.2 gives a program that prompts the user to enter an integer. If the number is a multiple of 5, the program displays **HiFive**. If the number is divisible by 2, it displays **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();           enter input
8
9          if (number % 5 == 0)                    check 5
10             System.out.println("HiFive");
11
12         if (number % 2 == 0)                    check even
13             System.out.println("HiEven");
14     }
15 }
```



```
Enter an integer: 4 [Enter]
HiEven
```



```
Enter an integer: 30 [Enter]
HiFive
HiEven
```

The program prompts the user to enter an integer (lines 6–7) and displays **HiFive** if it is divisible by **5** (lines 9–10) and **HiEven** if it is divisible by **2** (lines 12–13).



3.4 Write an **if** statement that assigns **1** to **x** if **y** is greater than **0**.

3.5 Write an **if** statement that increases pay by 3% if **score** is greater than **90**.

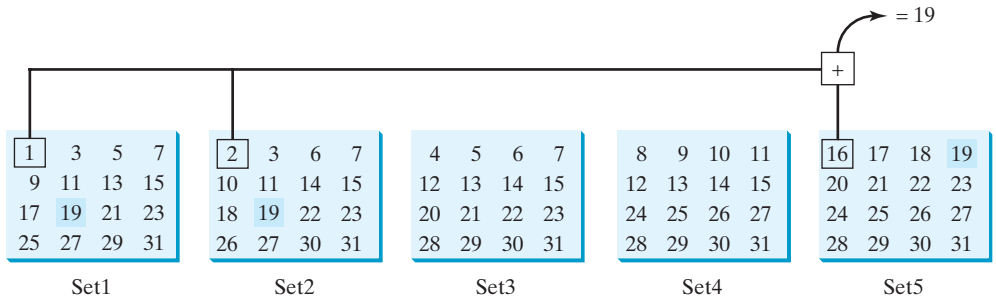
MyProgrammingLab™

3.4 Case Study: Guessing Birthdays



Guessing birthdays is an interesting problem with a simple programming solution.

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 41–44), in Set2 (lines 50–53), in Set3 (lines 59–62), in Set4 (lines 68–71), and in Set5 (lines 77–80). 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
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)
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)
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
in Set4? 73      if (answer == 1)
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
in Set5? 82      if (answer == 1)
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 Enter

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 Enter

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 Enter

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 Enter

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 Enter
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		
85			Your birthday is 19!



This 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 (binary numbers are introduced in Appendix F, Number Systems). A binary number for decimal integers between **1** and **31** has at most five digits, as shown in Figure 3.2a. Let it be $b_5b_4b_3b_2b_1$. Thus, $b_5b_4b_3b_2b_1 = b_50000 + b_4000 + b_300 + b_20 + b_1$, as shown in Figure 3.2b. 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

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
b_1	1	1
+ $b_5b_4b_3b_2b_1$	+ 10011	+ 11111
	19	31

(b)

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**.

3.5 Two-Way **if-else** Statements

An **if-else** statement decides which statements to execute based on whether the condition is true or false.



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-else** statement. The actions that a two-way **if-else** statement specifies differ based on whether the condition is **true** or **false**.

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

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

The flowchart of the statement is shown in Figure 3.3.

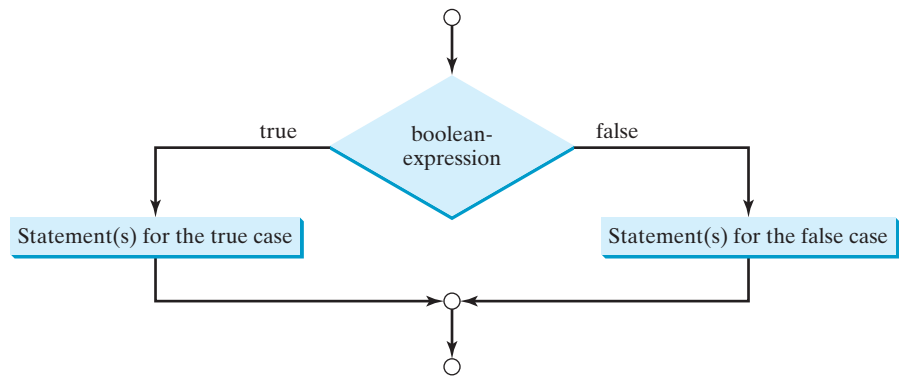


FIGURE 3.3 An **if-else** statement executes statements for the true case if the **Boolean-expression** evaluates to **true**; otherwise, statements for the false case are executed.

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:

two-way if-else statement

```
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");
}
```

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

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.");
```

3.6 Write an **if** statement that increases **pay** by 3% if **score** is greater than **90**, otherwise increases **pay** by 1%.

3.7 What is the printout of the code in (a) and (b) if **number** is **30**? What if **number** is **35**?



MyProgrammingLab™

```
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)

3.6 Nested **if** and Multi-Way **if-else** Statements

An **if** statement can be inside another **if** statement to form a nested **if** statement.



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:

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 3.4a, for instance, assigns a letter grade to the variable **grade** according to the score, with multiple alternatives.

```
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)

FIGURE 3.4 A preferred format for multiple alternatives is shown in (b) using a multi-way **if-else** statement.

The execution of this **if** statement proceeds as shown in Figure 3.5. 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) are 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**.

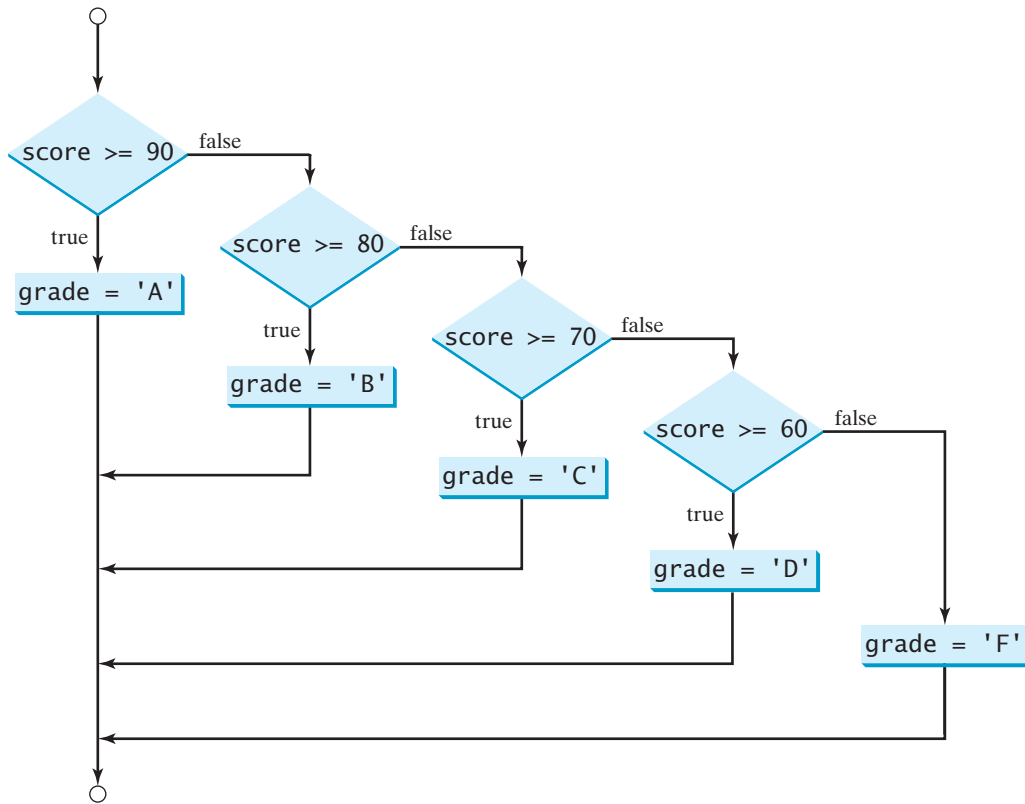


FIGURE 3.5 You can use a multi-way **if-else** statement to assign a grade.

The **if** statement in Figure 3.4a is equivalent to the **if** statement in Figure 3.4b. In fact, Figure 3.4b is the preferred coding style for multiple alternative **if** statements. This style, called *multi-way if-else statements*, avoids deep indentation and makes the program easy to read.

multi-way if statement



MyProgrammingLab™

- 3.8** 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);
  
```

- 3.9** 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);
  
```

3.10 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';

```

3.7 Common Errors in Selection Statements

*Forgetting necessary braces, ending an **if** statement in the wrong place, mistaking **==** for **=**, and dangling **else** clauses are 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 following code in (a) 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 end of an **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);
}

```

(a)

Equivalent

Empty block

```

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

```

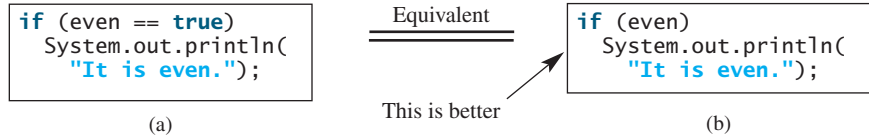
(b)

This mistake is hard to find, because it is neither a compile 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 this 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):



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 the equality of two items in a test condition is a common error. It could lead to the following erroneous statement:

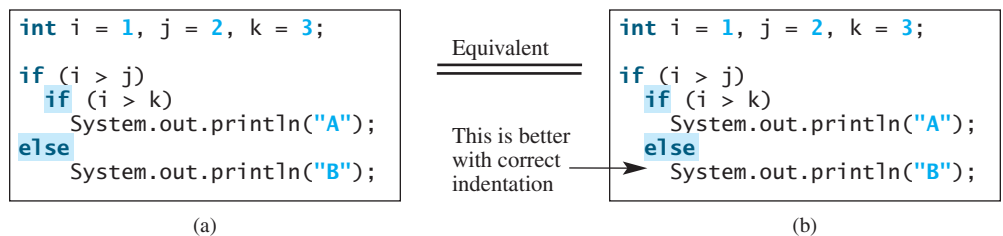
```
if (even = true)
    System.out.println("It is even.");
```

This statement does not have compile 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).

dangling else ambiguity



Since **(i > j)** is false, nothing is displayed from the statements 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 displays **B**.

**Tip**

Often new programmers write the code that assigns a test condition to a **boolean** variable like the code in (a):

```
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).



MyProgrammingLab™

3.11 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)

3.12 Rewrite the following statement using a Boolean expression:

```
if (count % 10 == 0)
    newLine = true;
else
    newLine = false;
```

3.13 Are the following statements correct? Which one is better?

```
if (age < 16)
    System.out.println
        ("Cannot get a driver's license");
if (age >= 16)
    System.out.println
        ("Can get a driver's license");
```

(a)

```
if (age < 16)
    System.out.println
        ("Cannot get a driver's license");
else
    System.out.println
        ("Can get a driver's license");
```

(b)

3.14 What is the output of the following code if **number** is 14, 15, and 30?

```
if (number % 2 == 0)
    System.out.println
        (number + " is even");
if (number % 5 == 0)
    System.out.println
        (number + " is multiple of 5");
```

(a)

```
if (number % 2 == 0)
    System.out.println
        (number + " is even");
else if (number % 5 == 0)
    System.out.println
        (number + " is multiple of 5");
```

(b)

3.8 Generating Random Numbers



You can use `Math.random()` to obtain a random double value between `0.0` and `1.0`, excluding `1.0`.



VideoNote

Program subtraction quiz

`random()` method

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 it 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.

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 $0.0 \leq d < 1.0$. Thus, `(int)(Math.random() * 10)` returns a random single-digit integer (i.e., a number between `0` and `9`).

The program can work as follows:

1. Generate two single-digit integers into `number1` and `number2`.
2. If `number1 < number2`, swap `number1` with `number2`.
3. Prompt the student to answer, “What is `number1 - number2`?”
4. 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);
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();
21
22         // 4. Grade the answer and display the result
23         if (number1 - number2 == answer)
24             System.out.println("You are correct!");
25         else
26             System.out.println("Your answer is wrong\n" + number1 + " - "
27                 + number2 + " is " + (number1 - number2));
28     }
29 }
```

random number

get answer

check the answer



What is $6 - 6$? 0 Enter
You are correct!

What is 9 - 2? 5

Your answer is wrong

9 - 2 is 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).

3.15 Which of the following is a possible output from invoking `Math.random()`?

323.4, 0.5, 34, 1.0, 0.0, 0.234

- 3.16**
- How do you generate a random integer `i` such that $0 \leq i < 20$?
 - How do you generate a random integer `i` such that $10 \leq i < 20$?
 - How do you generate a random integer `i` such that $10 \leq i \leq 50$?



MyProgrammingLab™

3.9 Case Study: Computing Body Mass Index

You can use nested `if` statements to write a program that interprets body mass index.

Body Mass Index (BMI) is a measure of health based on height and weight. It can be calculated by taking your weight in kilograms and dividing it by the square of your height in meters. The interpretation of BMI for people 20 years or older is as follows:



BMI	Interpretation
Below 18.5	Underweight
18.5–24.9	Normal
25.0–29.9	Overweight
Above 30.0	Obese

Write a program that prompts the user to enter a weight in pounds and height in inches and displays 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 ComputeAndInterpretBMI.java

input weight

input height

compute bmi

display output

```
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.println("BMI is " + bmi);
26         if (bmi < 18.5)
27             System.out.println("Underweight");
28         else if (bmi < 25)
29             System.out.println("Normal");
30         else if (bmi < 30)
31             System.out.println("Overweight");
32         else
33             System.out.println("Obese");
34     }
35 }
```



```
Enter weight in pounds: 146 ↵ Enter
Enter height in inches: 70 ↵ Enter
BMI is 20.948603801493316
Normal
```



line#	weight	height	weightInKilograms	heightInMeters	bmi	output
9	146					
13		70				
19			66.22448602			
20				1.778		
21					20.9486	
25						BMI is 20.95
31						Normal

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

3.10 Case Study: Computing Taxes

You can use nested `if` statements to write a program for 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 or qualified widow(er), 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 total tax is \$1,082.50.



VideoNote

Use multi-way `if-else` statements

TABLE 3.2 2009 U.S. Federal Personal Tax Rates

Marginal Tax Rate	Single	Married Filing Jointly or Qualifying 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	\$68,526 – \$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 or qualified widow(er), **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 or qualifying widow(er)
}
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 for computing 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 or qualifying widow(er),
11             + "\n2-married separately, 3-head of household)\n" +
12             "Enter the filing status: ");
13             input status
14
15             // Prompt the user to enter taxable income
16             System.out.print("Enter the taxable income: ");
17             input income
18
19             // Compute tax
20             double tax = 0;
21
22             compute tax
23             if (status == 0) { // Compute tax for single filers
24                 if (income <= 8350)
25                     tax = income * 0.10;
26                 else if (income <= 33950)
27                     tax = 8350 * 0.10 + (income - 8350) * 0.15;
28                 else if (income <= 82250)
29                     tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
30                         (income - 33950) * 0.25;
31                 else if (income <= 171550)
32                     tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
33                         (82250 - 33950) * 0.25 + (income - 82250) * 0.28;
34                 else if (income <= 372950)
35                     tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
36                         (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
37                         (income - 171550) * 0.33;
38                 else
39                     tax = 8350 * 0.10 + (33950 - 8350) * 0.15 +
40                         (82250 - 33950) * 0.25 + (171550 - 82250) * 0.28 +
41                         (372950 - 171550) * 0.33 + (income - 372950) * 0.35;
42             }
43             else if (status == 1) { // Left as exercise
44                 // Compute tax for married file jointly or qualifying widow(er)
45             }
46             else if (status == 2) { // Compute tax for married separately
47                 // Left as exercise
48             }
49             else if (status == 3) { // Compute tax for head of household
50                 // Left as exercise
51             }
52             else {
53                 System.out.println("Error: invalid status");
54                 System.exit(1);
55             }
56
57             // Display the result
58             display output
59             System.out.println("Tax is " + (int)(tax * 100) / 100.0);
60         }
61     }

```



(0-single filer, 1-married jointly or qualifying widow(er),
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	
57				Tax is 117683.5

The program receives the filing status and taxable income. The multi-way **if-else** statements (lines 22, 42, 45, 48, 51) check the filing status and compute the tax based on the filing status.

System.exit(status) (line 53) is defined in the **System** class. Invoking this method terminates the program. The status **0** indicates that the program is terminated normally. A nonzero status code indicates abnormal termination.

System.exit(status)

An initial value of **0** is assigned to **tax** (line 20). A compile 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 compile 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 testing easier, because the errors are likely in the new code you just added.

incremental development and testing

3.17 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;
```



MyProgrammingLab™

3.11 Logical Operators

The logical operators **!**, **&&**, **||**, and **^** can be used to create a compound Boolean expression.



Sometimes, whether a statement is executed is determined by a combination of several conditions. You can use logical operators to combine these conditions to form a compound Boolean expression. *Logical operators*, also known as *Boolean operators*, operate on Boolean values

to create a new Boolean value. Table 3.3 lists the Boolean operators. Table 3.4 defines the not (!) operator, which 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. Note that **p1 ^ p2** is the same as **p1 != p2**.

TABLE 3.3 Boolean Operators

Operator	Name	Description
!	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.
false	true	!(gender == 'M') is true, because (gender == 'M') is false.

TABLE 3.5 Truth Table for Operator &&

p ₁	p ₂	p ₁ && p ₂	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 ||

p ₁	p ₂	p ₁ p ₂	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 \wedge

p ₁	p ₂	p ₁ \wedge p ₂	Example (assume age = 24, gender = 'F')
false	false	false	(age > 34) \wedge (gender == 'F') is true , because (age > 34) is false but (gender == 'F') is true .
false	true	true	
true	false	true	(age > 34) \wedge (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;
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();
11
12        if (number % 2 == 0 && number % 3 == 0)
13            System.out.println(number + " is divisible by 2 and 3.");
14
15        if (number % 2 == 0 || number % 3 == 0)
16            System.out.println(number + " is divisible by 2 or 3.");
17
18        if (number % 2 == 0 ^ number % 3 == 0)
19            System.out.println(number +
20                " is divisible by 2 or 3, but not both.");
21    }
22 }
```

import class

input

and

or

exclusive or

Enter an integer: 4

4 is divisible by 2 or 3.

4 is divisible by 2 or 3, but not both.



Enter an integer: 18

18 is divisible by 2 and 3.

18 is divisible by 2 or 3.



(**number** % 2 == 0 && **number** % 3 == 0) (line 12) checks whether the number is divisible by both 2 and 3. (**number** % 2 == 0 || **number** % 3 == 0) (line 15) checks whether the number is divisible by 2 and/or by 3. (**number** % 2 == 0 ^ **number** % 3 == 0) (line 18) checks whether the number is divisible by 2 or 3, but not both.



Caution

In mathematics, the expression

1 <= numberOfDaysInAMonth <= 31

incompatible operands

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)
```

cannot cast `boolean`**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.

De Morgan's law

**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 the same as
!condition1 || !condition2
!(condition1 || condition2) is the same as
!condition1 && !condition2
```

For example,

```
!(number % 2 == 0 && number % 3 == 0)
```

can be simplified using an equivalent expression:

```
(number % 2 != 0 || number % 3 != 0)
```

As another example,

```
!(number == 2 || number == 3)
```

is better written as

```
number != 2 && number != 3
```

conditional operator
short-circuit operator

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. Java also provides the conditional AND (`&`) and OR (`|`) operators, which are covered in Supplement III.C and III.D for advanced readers.

**Check
Point****MyProgrammingLab™**

3.18 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)
```

3.19 Write a Boolean expression that evaluates to `true` if a number stored in variable `num` is between `1` and `100`.

3.20 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.

3.21 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)
```

3.22 Suppose that **x** is **1**. What is **x** after the evaluation of the following expression?

- a. `(x >= 1) && (x++ > 1)`
- b. `(x > 1) && (x++ > 1)`

3.23 What is the value of the expression `ch >= 'A' && ch <= 'Z'` if **ch** is **'A'**, **'p'**, **'E'**, or **'5'**?

3.24 Suppose, when you run the program, you enter input **2 3 6** 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));
    }
}
```

3.25 Write a Boolean expression that evaluates **true** if **age** is greater than **13** and less than **18**.

3.26 Write a Boolean expression that evaluates **true** if **weight** is greater than **50** pounds or height is greater than **60** inches.

3.27 Write a Boolean expression that evaluates **true** if **weight** is greater than **50** pounds and height is greater than **60** inches.

3.28 Write a Boolean expression that evaluates **true** if either **weight** is greater than **50** pounds or height is greater than **60** inches, but not both.

3.12 Case Study: 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.

You can use the following Boolean expressions to check whether a year is a 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();
9
10         // Check if the year is a leap year
11         boolean isLeapYear =
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);
16     }
17 }

```

input

leap year?

display result



Enter a year: 2012
2008 is a leap year? true



Enter a year: 1900
1900 is a leap year? false



Enter a year: 2002
2002 is a leap year? false

3.13 Case Study: Lottery



The lottery program involves generating random numbers, comparing digits, and using Boolean operators.

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 rules:

1. If the user input matches the lottery number in the exact order, the award is \$10,000.
2. If all the digits in the user input match all the digits in the lottery number, the award is \$3,000.
3. If one digit in the user input matches a digit in the lottery number, 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 number

```

```
6  int lottery = (int)(Math.random() * 100);
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();
12
13 // Get digits from lottery
14 int lotteryDigit1 = lottery / 10;
15 int lotteryDigit2 = lottery % 10;
16
17 // Get digits from guess
18 int guessDigit1 = guess / 10;
19 int guessDigit2 = guess % 10;
20
21 System.out.println("The lottery number is " + lottery);
22
23 // Check the guess
24 if (guess == lottery)
25     System.out.println("Exact match: you win $10,000");
26 else if (guessDigit2 == lotteryDigit1
27         && guessDigit1 == lotteryDigit2)
28     System.out.println("Match all digits: you win $3,000");
29 else if (guessDigit1 == lotteryDigit1
30         || guessDigit1 == lotteryDigit2
31         || guessDigit2 == lotteryDigit1
32         || guessDigit2 == lotteryDigit2)
33     System.out.println("Match one digit: you win $1,000");
34 else
35     System.out.println("Sorry, no match");
36 }
37 }
```

generate a lottery number

enter a guess

exact match?

match all digits?

match one digit?

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



line#	6	11	14	15	18	19	33
variable							
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 and display "Sorry, no match" (lines 34–35).

3.14 switch Statements



A **switch** statement executes statements based on the value of a variable or an expression.

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 simplify coding for multiple conditions. You can write the following **switch** statement to replace the nested **if** statement in Listing 3.6:

```
switch (status) {
    case 0: compute tax for single filers;
            break;
    case 1: compute tax for married jointly or qualifying widow(er);
            break;
    case 2: compute tax for married filing separately;
            break;
    case 3: compute tax for head of household;
            break;
    default: System.out.println("Error: invalid status");
            System.exit(1);
}
```

The flowchart of the preceding **switch** statement is shown in Figure 3.6.

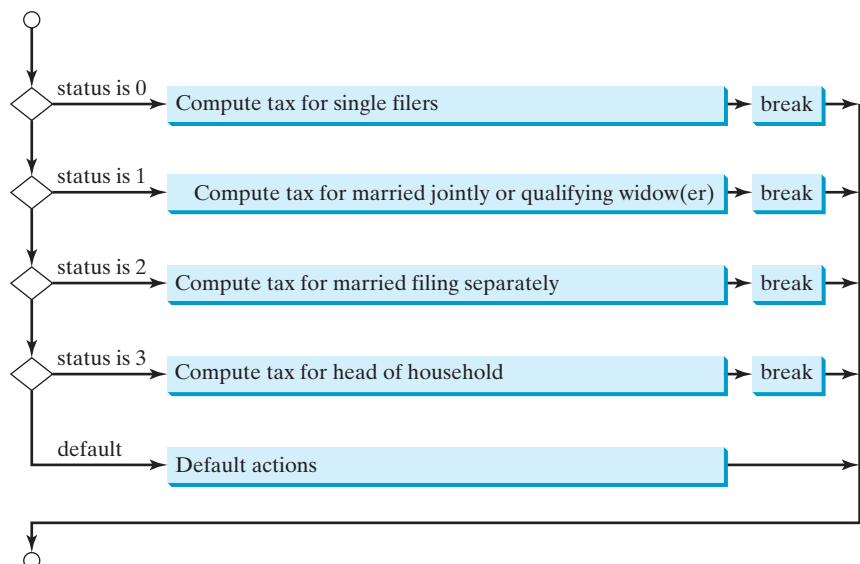


FIGURE 3.6 The **switch** statement checks all cases and executes the statements in the matched case.

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**, **int**, or **String** type and must always be enclosed in parentheses. (Using **String** type in the **switch** expression is new in JDK 7.)
- 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 **default** case, which is optional, can be used to perform actions when none of the specified cases matches the **switch-expression**.
- The keyword **break** is optional. The **break** statement immediately ends the **switch** statement.



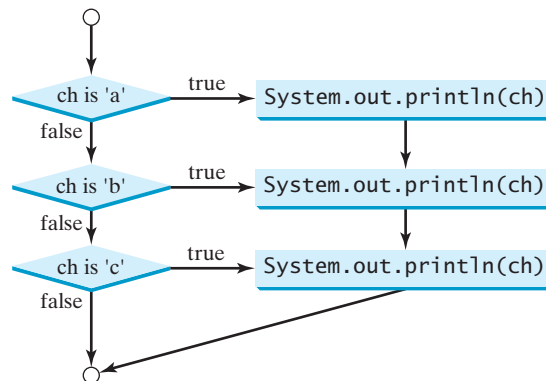
Caution

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 displays the character **a** three times if **ch** is **a**:

without break

fall-through behavior

```
switch (ch) {
    case 'a': System.out.println(ch);
    case 'b': System.out.println(ch);
    case 'c': System.out.println(ch);
}
```



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.

Now let us write a program to find out the Chinese Zodiac sign for a given year. The Chinese Zodiac is based on a twelve-year cycle, with each year represented by an animal—monkey, rooster, dog, pig, rat, ox, tiger, rabbit, dragon, snake, horse, or sheep—in this cycle, as shown in Figure 3.7.

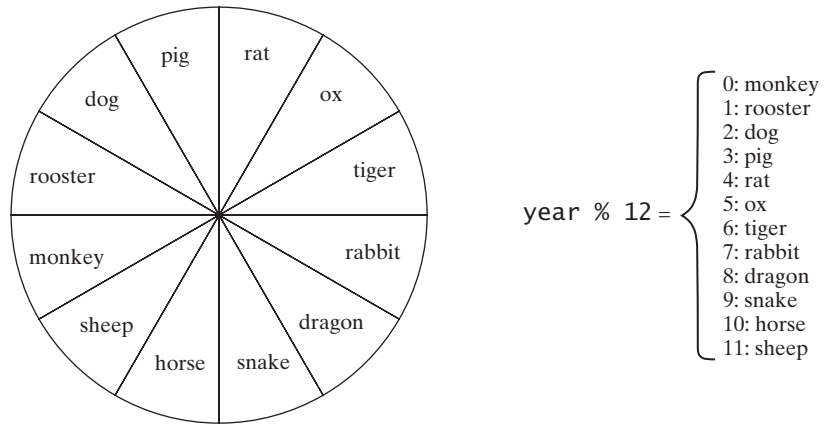


FIGURE 3.7 The Chinese Zodiac is based on a twelve-year cycle.

Note that `year % 12` determines the Zodiac sign. 1900 is the year of the rat because `1900 % 12` is 4. Listing 3.10 gives a program that prompts the user to enter a year and displays the animal for the year.

LISTING 3.10 ChineseZodiac.java

```

1  import java.util.Scanner;
2
3  public class ChineseZodiac {
4      public static void main(String[] args) {
5          Scanner input = new Scanner(System.in);
6
7          System.out.print("Enter a year: ");
8          int year = input.nextInt();
9
10         switch (year % 12) {
11             case 0: System.out.println("monkey"); break;
12             case 1: System.out.println("rooster"); break;
13             case 2: System.out.println("dog"); break;
14             case 3: System.out.println("pig"); break;
15             case 4: System.out.println("rat"); break;
16             case 5: System.out.println("ox"); break;
17             case 6: System.out.println("tiger"); break;
18             case 7: System.out.println("rabbit"); break;
19             case 8: System.out.println("dragon"); break;
20             case 9: System.out.println("snake"); break;
21             case 10: System.out.println("horse"); break;
22             case 11: System.out.println("sheep"); break;
23         }
24     }
25 }

```

enter year

determine Zodiac sign



Enter a year: 1963 → Enter
rabbit

Enter a year: 1877
 ox



3.29 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?



MyProgrammingLab™

3.30 What is **y** after the following **switch** statement is executed? Rewrite the code using the **if-else** statement.

```
x = 3; y = 3;
switch (x + 3) {
    case 6: y = 1;
    default: y += 1;
}
```

3.31 What is **x** after the following **if-else** statement is executed? Use a **switch** statement to rewrite it and draw the flowchart for the new **switch** statement.

```
int x = 1, a = 3;
if (a == 1)
    x += 5;
else if (a == 2)
    x += 10;
else if (a == 3)
    x += 16;
else if (a == 4)
    x += 34;
```

3.32 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.

3.15 Conditional Expressions

A conditional expression evaluates an expression based on a condition.



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 the following 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:

conditional expression

```
boolean-expression ? expression1 : expression2;
```

The result of this conditional expression is **expression1** if **boolean-expression** is true; otherwise the result is **expression2**.

Suppose you want to assign the larger number of 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");
```

As you can see from these examples, conditional expressions enable you to write short and concise code.



Note

The symbols **?** and **:** appear together in a conditional expression. They form a conditional operator called a *ternary operator* because it uses three operands. It is the only ternary operator in Java.



Check
Point

MyProgrammingLab™

- 3.33** Suppose that, when you run the following program, you enter input **2 3 6** 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) ? "sorted" : "not sorted");
    }
}
```

- 3.34** Rewrite the following **if** statements using the conditional operator.

```
if (ages >= 16)
    ticketPrice = 20;
else
    ticketPrice = 10;
```

```
if (count % 10 == 0)
    System.out.print(count + "\n");
else
    System.out.print(count);
```

- 3.35** Rewrite the following conditional expressions using **if-else** statements.

- `score = (x > 10) ? 3 * scale : 4 * scale;`
- `tax = (income > 10000) ? income * 0.2 : income * 0.17 + 1000;`
- `System.out.println((number % 3 == 0) ? i : j);`

3.16 Formatting Console Output



Key
Point

*You can use the **System.out.printf** method to display formatted output on the console.*

Often it is desirable to display numbers in a certain format. For example, the following code computes interest, given the amount and the annual interest rate.

```
double amount = 12618.98;
double interestRate = 0.0013;
```

```
double interest = amount * interestRate;
System.out.println("Interest is " + interest);
```

```
Interest is 16.404674
```



Because the interest amount is currency, it is desirable to display only two digits after the decimal point. To do this, you can write the code as follows:

```
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.println("Interest is "
    + (int)(interest * 100) / 100.0);
```

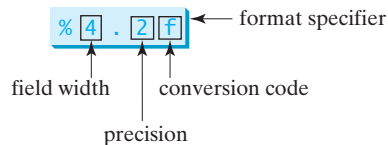
```
Interest is 16.4
```



However, the format is still not correct. There should be two digits after the decimal point: **16.40** rather than **16.4**. You can fix it by using the **printf** method, like this:

printf

```
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.printf("Interest is %4.2f", interest);
```



```
Interest is 16.40
```



The syntax to invoke this method is

```
System.out.printf(format, item1, item2, ..., itemk)
```

where **format** is a string that may consist of substrings and format specifiers.

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 format specifier consists of a percent sign (%) followed by a conversion code. Table 3.8 lists some frequently used simple format specifiers.

format specifier

TABLE 3.8 Frequently Used Format Specifiers

Format 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 format specifiers in order, in number, and in exact type. For example, the format 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 format specifier, as shown in the examples in Table 3.9.

TABLE 3.9 Examples of Specifying Width and Precision

Example	Output
<code>%5c</code>	Output the character and add four spaces before the character item, because the width is 5.
<code>%6b</code>	Output the Boolean value and add one space before the false value and two spaces before the true value.
<code>%5d</code>	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.
<code>%10.2f</code>	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 is <7, add spaces before the number. If the number of digits before the decimal point in the item is >7, the width is automatically increased.
<code>%10.2e</code>	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.
<code>%12s</code>	Output the string with width at least 12 characters. If the string item has fewer than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.

If an item requires more spaces than the specified width, the width is automatically increased. For example, the following code

```
System.out.printf("%3d#%2s#%3.2f\n", 1234, "Java", 51.6653);
```

displays

```
1234#Java#51.67
```

The specified width for `int` item `1234` is `3`, which is smaller than its actual size `4`. The width is automatically increased to `4`. The specified width for string item `Java` is `2`, which is smaller than its actual size `4`. The width is automatically increased to `4`. The specified width for `double` item `51.6653` is `3`, but it needs width `5` to display `51.67`, so the width is automatically increased to `5`.

By default, the output is right justified. You can put the minus sign (-) in the format 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", 5.63);
System.out.printf("%-8d%-8s%-8.1f \n", 1234, "Java", 5.63);
```

display

```

|← 8 →|← 8 →|← 8 →|
□□□□ 1234 □□□□ Java □□□□ 5.6
1234 □□□□ Java □□□□ 5.6 □□□□
```

where the square box (□) denotes a blank space.



Caution

The items must match the format specifiers in exact type. The item for the format 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 format specifier. To output a literal **%** in the format string, use **%%**.

3.36 What are the format specifiers for outputting a Boolean value, a character, a decimal integer, a floating-point number, and a string?

3.37 What is wrong in the following statements?

- `System.out.printf("%5d %d", 1, 2, 3);`
- `System.out.printf("%5d %f", 1);`
- `System.out.printf("%5d %f", 1, 2);`

3.38 Show the output of the following statements.

- `System.out.printf("amount is %f %e\n", 32.32, 32.32);`
- `System.out.printf("amount is %5.4f %5.4e\n", 32.32, 32.32);`
- `System.out.printf("%6b\n", (1 > 2));`
- `System.out.printf("%6s\n", "Java");`
- `System.out.printf("%-6b%s\n", (1 > 2), "Java");`
- `System.out.printf("%6b%-8s\n", (1 > 2), "Java");`



Check
Point

MyProgrammingLab™

3.17 Operator Precedence and Associativity

Operator precedence and associativity determine the order in which operators are evaluated.



Key
Point

Section 2.11 introduced operator precedence involving arithmetic operators. This section discusses operator precedence in more details. Suppose that you have this expression:

```
3 + 4 * 4 > 5 * (4 + 3) - 1 && (4 - 3 > 5)
```

What is its value? What is the execution order of the operators?


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. The logical operators have lower precedence than the relational operators and the relational operators have lower precedence than the arithmetic operators. 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.)

operator precedence

TABLE 3.10 Operator Precedence Chart

Precedence	Operator
	var++ and var-- (Postfix)
	+ , - (Unary plus and minus), ++var and --var (Prefix)
	(type) (Casting)
	! (Not)
	* , / , % (Multiplication, division, and remainder)
	+ , - (Binary addition and subtraction)
	< , <= , > , >= (Comparison)
	== , != (Equality)
	^ (Exclusive OR)
	&& (AND)
	 (OR)
	= , += , -= , *= , /= , %= (Assignment operator)

operator associativity

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

$$a - b + c - d \quad \text{is equivalent to} \quad ((a - b) + c) - d$$

Assignment operators are *right associative*. Therefore, the expression

$$a = b += c = 5 \quad \text{is equivalent to} \quad a = (b += (c = 5))$$

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. Advanced readers may refer to Supplement III.B for more discussions on how an expression is evaluated in Java *behind the scenes*.

behind the scenes

3.39 List the precedence order of the Boolean operators. Evaluate the following expressions:

```
true || true && false
true && true || false
```



MyProgrammingLab™

3.40 True or false? All the binary operators except `=` are left associative.

3.41 Evaluate the following expressions:

```
2 * 2 - 3 > 2 && 4 - 2 > 5
2 * 2 - 3 > 2 || 4 - 2 > 5
```

3.42 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))`?

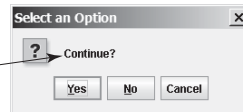
3.18 Confirmation Dialogs

You can use a confirmation dialog to obtain a confirmation from the user.



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 3.11. Figure 3.8 shows a sample run of the program for the day 19.

LISTING 3.11 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
```

```

set3      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
set4      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
set5      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
confirmation dialog 37      // Prompt the user to answer questions
           38      int answer = JOptionPane.showConfirmDialog(null,
           39      "Is your birthday in these numbers?\n" + set1);
           40
in set1?   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
in set2?   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
in set3?   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
in set4?   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
in set5?   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).

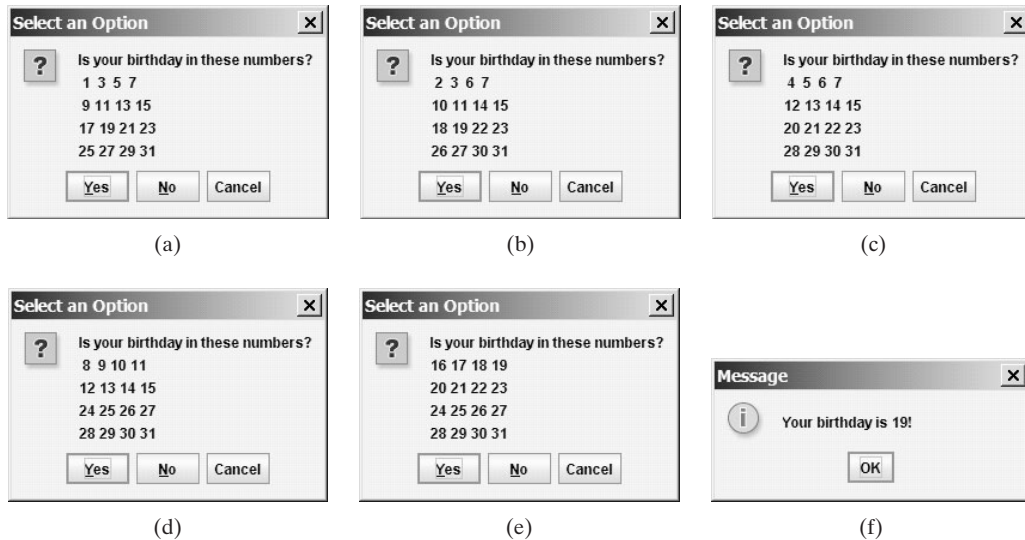


FIGURE 3.8 Click Yes in (a), Yes in (b), No in (c), No in (d), and Yes in (e).

3.43 How do you display a confirmation dialog? What value is returned when invoking `JOptionPane.showConfirmDialog`?



3.19 Debugging

Debugging is the process of finding and fixing errors in a program.



As mentioned in Section 1.11.1, 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 there. Runtime errors are not difficult to find either, because the Java interpreter displays them 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 to debugging is to use a combination of methods to help pinpoint 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. These approaches might work for debugging a short, simple program, but for a large, complex program, the most effective approach is to use a debugger utility.

bugs
debugging
hand-traces

JDK includes a command-line debugger, `jdb`, which is invoked with a class name. `jdb` is itself a Java program, running its own copy of Java interpreter. All the Java IDE tools, such as Eclipse and NetBeans, include integrated debuggers. The debugger utilities let you follow the execution of a program. They vary from one system to another, but they all support most of the following helpful features.

- **Executing a single statement at a time:** The debugger allows you to execute one statement at a time so that you can see the effect of each statement.
- **Tracing into or stepping over a method:** If a method is being executed, you can ask the debugger to enter the method and execute one statement at a time in the method, or you can ask it to step over the entire method. You should step over the entire method if you know that the method works. For example, always step over system-supplied methods, such as `System.out.println`.

- **Setting breakpoints:** You can also set a breakpoint at a specific statement. Your program pauses when it reaches a breakpoint. You can set as many breakpoints as you want. Breakpoints are particularly useful when you know where your program error starts. You can set a breakpoint at that statement and have the program execute until it reaches the breakpoint.
- **Displaying variables:** The debugger lets you select several variables and display their values. As you trace through a program, the content of a variable is continuously updated.
- **Displaying call stacks:** The debugger lets you trace all of the method calls. This feature is helpful when you need to see a large picture of the program-execution flow.
- **Modifying variables:** Some debuggers enable you to modify the value of a variable when debugging. This is convenient when you want to test a program with different samples but do not want to leave the debugger.

debugging in IDE



Tip

If you use an IDE such as Eclipse or NetBeans, please refer to *Learning Java Effectively with Eclipse/NetBeans* in Supplements II.C and II.E on the Companion Website. The supplement shows you how to use a debugger to trace programs and how debugging can help in learning Java effectively.

KEY TERMS

Boolean expression	82	flowchart	84
boolean data type	82	format specifier	113
Boolean value	82	operator associativity	116
conditional operator	104	operator precedence	116
dangling else ambiguity	94	selection statement	82
debugging	119	short-circuit operator	104
fall-through behavior	109		

CHAPTER SUMMARY

1. A **boolean** type variable can store 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 of actions. There are several types of selection statements: **if** statements, two-way **if-else** statements, nested **if** statements, multi-way **if-else** 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**, **int**, or **String**.
8. The keyword **break** is optional in a **switch** statement, but it is normally used at the end of each case in order to skip the remainder of the **switch** statement. If the **break** statement is not present, the next **case** statement will be executed.
9. The operators in expressions are evaluated in the order determined by the rules of parentheses, *operator precedence*, and *operator associativity*.
10. Parentheses can be used to force the order of evaluation to occur in any sequence.
11. Operators with higher precedence are evaluated earlier. For operators of the same precedence, their associativity determines the order of evaluation.
12. All binary operators except assignment operators are left-associative; assignment operators are right-associative.

TEST QUESTIONS

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

PROGRAMMING EXERCISES

MyProgrammingLab™



Pedagogical Note

For each exercise, carefully analyze the problem requirements and design strategies for solving the problem before coding.

think before coding



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: solve quadratic equations) The two roots of a quadratic equation $ax^2 + bx + c = 0$ can be obtained using the following formula:

$$r_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \quad \text{and} \quad r_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$b^2 - 4ac$ 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 that 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 (*Game: add 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.

Sections 3.3–3.8

***3.3** (*Algebra: solve 2×2 linear equations*) You can use Cramer's rule to solve the following 2×2 system of linear equation:

$$\begin{array}{l} ax + by = e \\ cx + dy = f \end{array} \quad x = \frac{ed - bf}{ad - bc} \quad y = \frac{af - ec}{ad - bc}$$

Write a program that prompts the user to enter a, b, c, d, e , and f and displays the result. If $ad - bc$ 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: learn 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** (*Find future dates*) Write a program that prompts the user to enter an integer for today's day of the week (Sunday is 0, Monday is 1, . . . , and Saturday is 6). Also prompt the user to enter the number of days after today for a future day and display the future day of the week. Here is a sample run:



Enter today's day: 1
Enter the number of days elapsed since today: 3
Today is Monday and the future day is Thursday

```

Enter today's day: 0
Enter the number of days elapsed since today: 31
Today is Sunday and the future day is Wednesday
    
```



- *3.6** (*Health application: BMI*) Revise Listing 3.5, `ComputeAndInterpretBMI.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. Here is a sample run:

```

Enter weight in pounds: 140
Enter feet: 5
Enter inches: 10
BMI is 20.087702275404553
Normal
    
```



- 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.



VideoNote

- *3.8** (*Sort 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 $num1 \leq num2 \leq num3$.

Sort three integers

- **3.9** (*Business: check ISBN-10*) An **ISBN-10** (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 as X according to the ISBN-10 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. Here are sample runs:

```

Enter the first 9 digits of an ISBN as integer: 013601267
The ISBN-10 number is 0136012671
    
```



```

Enter the first 9 digits of an ISBN as integer: 013031997
The ISBN-10 number is 013031997X
    
```



- 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** (*Find 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 **2012**, the program should display that February 2012 had 29 days. If the user entered month **3** and year **2015**, the program should display that March 2015 had 31 days.

- 3.12** (*Check 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: compute 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: heads or tails*) Write a program that lets the user guess whether the flip of a coin results in heads or tails. 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 rules:

1. If the user input matches the lottery number in the exact order, the award is \$10,000.
2. If all the digits in the user input match all the digits in the lottery number, the award is \$3,000.
3. If one digit in the user input matches a digit in the lottery number, 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 Enter
The computer is scissor. You are rock. You won
```



```
scissor (0), rock (1), paper (2): 2 Enter
The computer is paper. You are paper too. It is a draw
```

- *3.18** (*Use the input dialog box*) Rewrite Listing 3.8, `LeapYear.java`, using the input dialog box.

- **3.19** (*Compute 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 every pair of two edges is greater than the remaining edge.

- *3.20** (*Science: wind-chill temperature*) Programming Exercise 2.17 gives a formula to compute the wind-chill temperature. The formula is valid for temperatures in the range between -58°F 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, it displays a message indicating whether the temperature and/or wind speed is invalid.

Comprehensive

- **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

$$h = \left(q + \frac{26(m+1)}{10} + k + \frac{k}{4} + \frac{j}{4} + 5j \right) \% 7$$

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., $\frac{\text{year}}{100}$).
- **k** is the year of the century (i.e., $\text{year} \% 100$).

Note that the division in the formula performs an integer division. 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., 2012): 2015 Enter
Enter month: 1-12: 1 Enter
Enter the day of the month: 1-31: 25 Enter
Day of the week is Sunday
```



```
Enter year: (e.g., 2012): 2012 Enter
Enter month: 1-12: 5 Enter
Enter the day of the month: 1-31: 12 Enter
Day of the week is Saturday
```



(*Hint: 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 3.9a.



VideoNote
Check point location

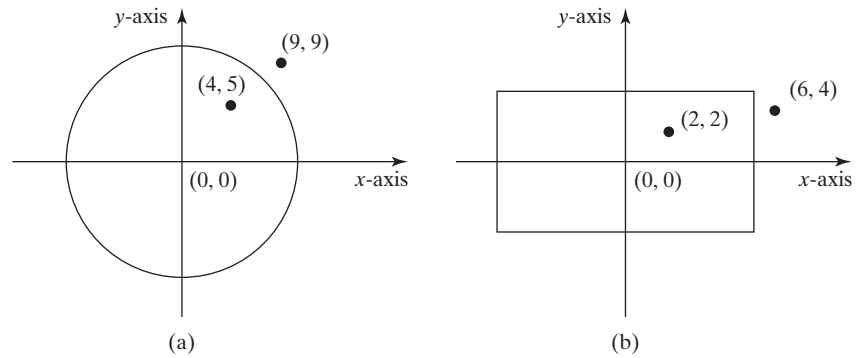


FIGURE 3.9 (a) Points inside and outside of the circle. (b) Points inside and outside of the rectangle.

(Hint: A point is in the circle if its distance to (0, 0) is less than or equal to 10. The formula for computing the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Test your program to cover all cases.) Two sample runs are shown below.



Enter a point with two coordinates: 4 5
Point (4.0, 5.0) is in the circle



Enter a point with two coordinates: 9 9
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 rectangle, as shown in Figure 3.9b. (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 5.0 / 2. Test your program to cover all cases.) Here are two sample runs.



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: pick 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** (Geometry: intersecting point) Two points on line 1 are given as (x1, y1) and (x2, y2) and on line 2 as (x3, y3) and (x4, y4), as shown in Figure 3.10a–b.

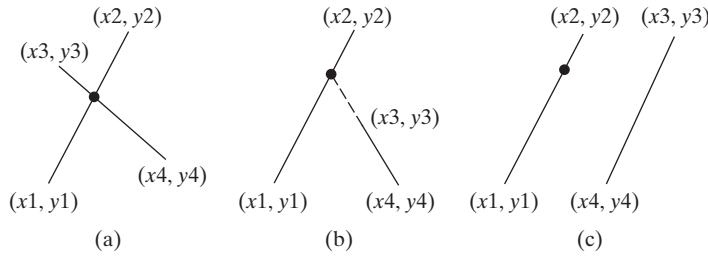


FIGURE 3.10 Two lines intersect in (a and b) and two lines are parallel in (c).

The intersecting point of the two lines can be found by solving the following linear equation:

$$(y_1 - y_2)x - (x_1 - x_2)y = (y_1 - y_2)x_1 - (x_1 - x_2)y_1$$

$$(y_3 - y_4)x - (x_3 - x_4)y = (y_3 - y_4)x_3 - (x_3 - x_4)y_3$$

This linear equation can be solved using Cramer's rule (see Exercise 3.3). If the equation has no solutions, the two lines are parallel (Figure 3.10c). Write a program that prompts the user to enter four points and displays the intersecting point. Here are sample runs:

```
Enter x1, y1, x2, y2, x3, y3, x4, y4: 2 2 5 -1.0 4.0 2.0 -1.0 -2.0
The intersecting point is at (2.88889, 1.1111)
```



```
Enter x1, y1, x2, y2, x3, y3, x4, y4: 2 2 7 6.0 4.0 2.0 -1.0 -2.0
The two lines are parallel
```

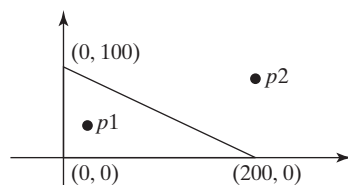


3.26 (Use the `&&`, `||` and `^` operators) Write a program that prompts the user to enter an integer and determines whether it is divisible by 5 and 6, whether it is divisible by 5 or 6, and whether it is divisible by 5 or 6, but not both. Here is a sample run of this program:

```
Enter an integer: 10
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 3.11. Test your program to cover all cases.

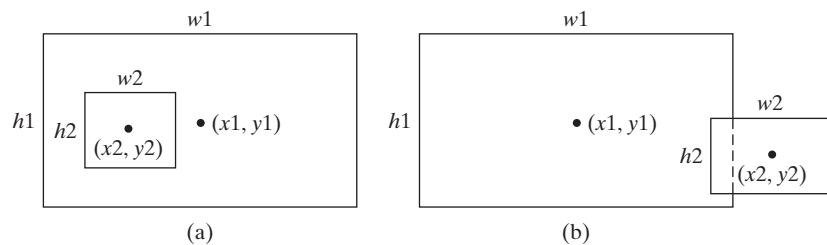


FIGURE 3.11 (a) A rectangle is inside another one. (b) A rectangle overlaps another one.

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
Enter r2's center x-, y-coordinates, width, and height: 3 4 4.5 5 Enter
r2 overlaps r1
```



```
Enter r1's center x-, y-coordinates, width, and height: 1 2 3 3 Enter
Enter r2's center x-, y-coordinates, width, and height: 40 45 3 2 Enter
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 3.12. (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 $\leq r1 + r2$. Test your program to cover all cases.)

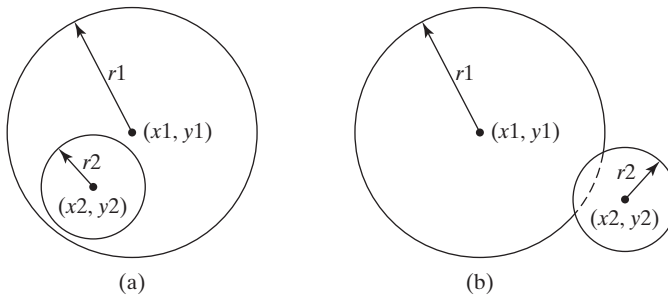


FIGURE 3.12 (a) A circle is inside another circle. (b) A circle overlaps another circle.

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
```



***3.30** (*Current time*) Revise Programming Exercise 2.8 to display the hour using a 12-hour clock. Here is a sample run:

```
Enter the time zone offset to GMT: -5 ↵ Enter
The current time is 4:50:34 AM
```



***3.31** (*Financials: currency exchange*) Write a program that prompts the user to enter the exchange rate from currency in U.S. dollars to Chinese RMB. Prompt the user to enter **0** to convert from U.S. dollars to Chinese RMB and **1** to convert from Chinese RMB and U.S. dollars. Prompt the user to enter the amount in U.S. dollars or Chinese RMB to convert it to Chinese RMB or U.S. dollars, respectively. Here are the sample runs:

```
Enter the exchange rate from dollars to RMB: 6.81 ↵ Enter
Enter 0 to convert dollars to RMB and 1 vice versa: 0 ↵ Enter
Enter the dollar amount: 100 ↵ Enter
$100.0 is 681.0 yuan
```





```
Enter the exchange rate from dollars to RMB: 6.81 ↵ Enter
Enter 0 to convert dollars to RMB and 1 vice versa: 1 ↵ Enter
Enter the RMB amount: 10000 ↵ Enter
10000.0 yuan is $1468.43
```



```
Enter the exchange rate from dollars to RMB: 6.81 ↵ Enter
Enter 0 to convert dollars to RMB and 1 vice versa: 5 ↵ Enter
Incorrect input
```

***3.32** (*Geometry: point position*) Given a directed line from point $p_0(x_0, y_0)$ to $p_1(x_1, y_1)$, you can use the following condition to decide whether a point $p_2(x_2, y_2)$ is on the left of the line, on the right, or on the same line (see Figure 3.13):

$$(x_1 - x_0) * (y_2 - y_0) - (x_2 - x_0) * (y_1 - y_0) \begin{cases} > 0 & p_2 \text{ is on the left side of the line} \\ = 0 & p_2 \text{ is on the same line} \\ < 0 & p_2 \text{ is on the right side of the line} \end{cases}$$

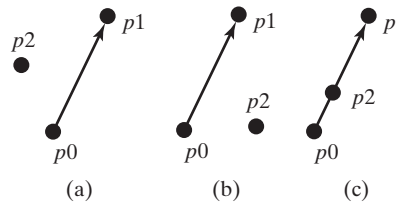


FIGURE 3.13 (a) p_2 is on the left of the line. (b) p_2 is on the right of the line. (c) p_2 is on the same line.

Write a program that prompts the user to enter the three points for p_0 , p_1 , and p_2 and displays whether p_2 is on the left of the line from p_0 to p_1 , to the right, or on the same line. Here are some sample runs:



```
Enter three points for p0, p1, and p2: 4.4 2 6.5 9.5 -5 4 ↵ Enter
p2 is on the left side of the line
```



```
Enter three points for p0, p1, and p2: 1 1 5 5 2 2 ↵ Enter
p2 is on the same line
```



```
Enter three points for p0, p1, and p2: 3.4 2 6.5 9.5 5 2.5 ↵ Enter
p2 is on the right side of the line
```

***3.33** (*Financial: compare costs*) Suppose you shop for rice in two different packages. You would like to write a program to compare the cost. The program prompts the user to enter the weight and price of the each package and displays the one with the better price. Here is a sample run:

Enter weight and price for package 1: 50 24.59

Enter weight and price for package 2: 25 11.99

Package 1 has a better price.



***3.34** (*Geometry: point on line segment*) Exercise 3.32 shows how to test whether a point is on an unbounded line. Revise Exercise 3.32 to test whether a point is on a line segment. Write a program that prompts the user to enter the three points for p0, p1, and p2 and displays whether p2 is on the line segment from p0 to p1. Here are some sample runs:

Enter three points for p0, p1, and p2: 1 1 2.5 2.5 1.5 1.5

(1.5, 1.5) is on the line segment from (1.0, 1.0) to (2.5, 2.5)



Enter three points for p0, p1, and p2: 1 1 2 2 3.5 3.5

(3.5, 3.5) is not on the line segment from (1.0, 1.0) to (2.0, 2.0)



***3.35** (*Decimal to hex*) Write a program that prompts the user to enter an integer between 0 and 15 and displays its corresponding hex number. Here are some sample runs:

Enter a decimal value (0 to 15): 11

The hex value is B



Enter a decimal value (0 to 15): 5

The hex value is 5



Enter a decimal value (0 to 15): 31

Invalid input



This page intentionally left blank

LOOPS

Objectives

- To write programs for executing statements repeatedly using a **while** loop (§4.2).
- To follow the loop design strategy to develop loops (§§4.2.1–4.2.3).
- 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 (§4.2.5).
- 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).
- To write a program that displays prime numbers (§4.10).
- To control a loop with a confirmation dialog (§4.11).



4.1 Introduction



Key
Point

A loop can be used to tell a program to execute statements repeatedly.

problem

Suppose that you need to display 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!");
 }

loop

So, how do you solve this problem?

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 display 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 display 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



Key
Point

A while loop executes statements repeatedly while the condition is true.

The syntax for the **while** loop is:

while loop

```
while (loop-continuation-condition) {
    // Loop body
    Statement(s);
}
```

loop body

iteration

loop-continuation-
condition

Figure 4.1a shows the **while**-loop flowchart. 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* (or *repetition*) 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.

The loop for displaying **Welcome to Java!** a hundred times introduced in the preceding section is an example of a **while** loop. Its flowchart is shown in Figure 4.1b. The

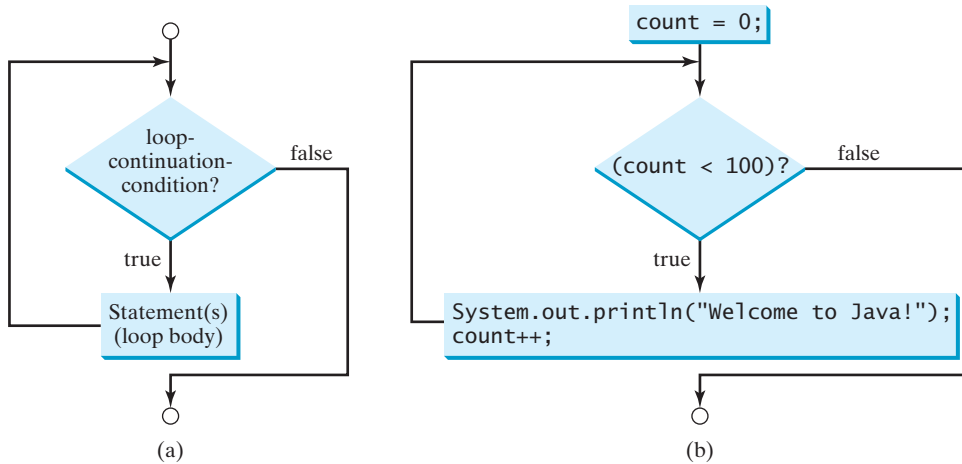


FIGURE 4.1 The **while** loop repeatedly executes the statements in the loop body when the **loop-continuation-condition** evaluates to **true**.

loop-continuation-condition is `count < 100` and the loop body contains the following two statements:

```

int count = 0;
while (count < 100) {
    System.out.println("Welcome to Java!");
    count++;
}
  
```

loop-continuation-condition

} loop body

In this example, you know exactly how many times the loop body needs to be executed because the 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 is incremented to `2`, `3`, and up to `10`. When `i` is `10`, `i < 10` is **false**, so the loop exits. Therefore, 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`.

infinite loop



Note

Make sure that the **loop-continuation-condition** eventually becomes **false** so that the loop will terminate. A common programming error involves *infinite loops* (i. e., the loop runs forever). If your program takes an unusually long time to run and does not stop, it may have an infinite loop. If you are running the program from the command window, press `CTRL+C` to stop it.

off-by-one error



Caution

Programmers often make the mistake of executing 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`.

```
int count = 0;
while (count <= 100) {
    System.out.println("Welcome to Java!");
    count++;
}
```

Recall that Listing 3.1, `AdditionQuiz.java`, gives a program that prompts the user to enter an answer for a question on addition of two single digits. Using a loop, you can now rewrite the program to let the user repeatedly enter a new answer until it is correct, as shown in Listing 4.1.

LISTING 4.1 RepeatAdditionQuiz.java

generate number1
generate number2

show question

get first answer

check answer

read an answer

```
1  import java.util.Scanner;
2
3  public class RepeatAdditionQuiz {
4      public static void main(String[] args) {
5          int number1 = (int)(Math.random() % 10);
6          int number2 = (int)(Math.random() % 10);
7
8          // Create a Scanner
9          Scanner input = new Scanner(System.in);
10
11         System.out.print(
12             "What is " + number1 + " + " + number2 + "? ";
13         int answer = input.nextInt();
14
15         while (number1 + number2 != answer) {
16             System.out.print("Wrong answer. Try again. What is "
17                 + number1 + " + " + number2 + "? ");
18             answer = input.nextInt();
19         }
20
21         System.out.println("You got it!");
22     }
23 }
```



```
What is 5 + 9? 12 Enter
Wrong answer. Try again. What is 5 + 9? 34 Enter
Wrong answer. Try again. What is 5 + 9? 14 Enter
You got it!
```

The loop in lines 15–19 repeatedly prompts the user to enter an **answer** when **number1 + number2 != answer** is **true**. Once **number1 + number2 != answer** is **false**, the loop exits.

4.2.1 Case Study: Guessing Numbers

The problem is to guess what 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:



VideoNote
Guess a number



```

Guess a magic number between 0 and 100
Enter your guess: 50 Enter
Your guess is too high
Enter your guess: 25 Enter
Your guess is too low
Enter your guess: 42 Enter
Your guess is too high
Enter your guess: 39 Enter
Yes, the number is 39

```

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 4.2.

code incrementally

LISTING 4.2 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();
14
15         if (guess == number)
16             System.out.println("Yes, the number is " + number);
17         else if (guess > number)
18             System.out.println("Your guess is too high");
19         else

```

generate a number

enter a guess

correct guess?

too high?


```

too low?      20      System.out.println("Your guess is too low");
               21      }
               22      }

```

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 4.3.

LISTING 4.3 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
generate a number 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         int guess = -1;
12         while (guess != number) {
13             // Prompt the user to guess the number
14             System.out.print("\nEnter your guess: ");
enter a 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 }

```

too high?
too low?

	line#	number	guess	output
	6	39		
iteration 1	11		-1	
	15		50	
	20			Your guess is too high
iteration 2	15		25	
	22			Your guess is too low
iteration 3	15		42	
	20			Your guess is too high
iteration 4	15		39	
	18			Yes, the number is 39



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 Case Study: Multiple Subtraction Quiz

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



VideoNote
Multiple subtraction quiz

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.

Listing 4.4 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 4.4 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();
9          String output = " "; // output string is initially empty
10         Scanner input = new Scanner(System.in);
11
12         while (count < NUMBER_OF_QUESTIONS) {
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(
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) {
31                 System.out.println("You are correct!");
32                 correctCount++; // Increase the correct answer count
33             }
34             else
35                 System.out.println("Your answer is wrong.\n" + number1
36                     + " - " + number2 + " should be " + (number1 - number2));
37
38             // Increase the question count
39             count++;
40
41             output += "\n" + number1 + "-" + number2 + "=" + answer +
42                 ((number1 - number2 == answer) ? " correct" : " wrong");
43         }
44
45         long endTime = System.currentTimeMillis();
46         long testTime = endTime - startTime;
47
48         System.out.println("Correct count is " + correctCount +
49             "\nTest time is " + testTime / 1000 + " seconds\n" + output);
50     }
51 }

```

get start time

loop

display a question

grade an answer

increase correct count

increase control variable

prepare output

end loop

get end time

test time

display result



```

What is 9 - 2? 7 ↵ Enter
You are correct!

What is 3 - 0? 3 ↵ Enter
You are correct!

What is 3 - 2? 1 ↵ Enter
You are correct!

What is 7 - 4? 4 ↵ Enter
Your answer is wrong.

7 - 4 should be 3
What is 7 - 5? 4 ↵ Enter

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 input. A loop that uses a sentinel value to control its execution is called a *sentinel-controlled loop*.

sentinel value

sentinel-controlled loop

Listing 4.5 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 4.5 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 integer (the input ends if it is 0): ";
12         int data = input.nextInt();

```

input

```

13
14 // Keep reading data until the input is 0
15 int sum = 0;
loop 16 while (data != 0) {
17     sum += data;
18
19     // Read the next data
20     System.out.print(
21         "Enter an integer (the input ends if it is 0): ";
22     data = input.nextInt();
23 }
24
25 System.out.println("The sum is " + sum);
26 }
27 }

```

end of loop

display result



```

Enter an integer (the input ends if it is 0): 2 Enter
Enter an integer (the input ends if it is 0): 3 Enter
Enter an integer (the input ends if it is 0): 4 Enter
Enter an integer (the input ends if it 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. Because 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 $1 + 0.9 + 0.8 + \dots + 0.1$:

```

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 okay 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 can 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
```

This command is called *input redirection*. The program takes the input from the file **input.txt** rather than having the user 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**.

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
```

Try running the program to see what contents are in **output.txt**.

- 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
```



MyProgrammingLab™

- 4.2** What is wrong if **guess** is initialized to **0** in line 11 in Listing 4.3?
- 4.3** How many times are the following loop bodies repeated? What is the printout of each 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 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;

        while (number != 0) {
            number = input.nextInt();
            if (number > max)
                max = number;
        }

        System.out.println("max is " + max);
        System.out.println("number " + number);
    }
}
```

4.5 What is the output of the following code? Explain the reason.

```
int x = 80000000;

while (x > 0)
    x++;

System.out.println("x is " + x);
```

4.3 The do-while Loop



*A **do-while** loop is the same as a **while** loop except that it executes the loop body first and then checks the loop continuation condition.*

The **do-while** loop is a variation of the **while** loop. Its syntax is:

```
do {
    // Loop body;
    Statement(s);
} while (loop-continuation-condition);
```

Its execution flowchart is shown in Figure 4.2.

The loop body is executed first, and 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. You can write a loop using either the **while** loop or the **do-while** loop. Sometimes one is a more convenient choice than the other. For example, you can rewrite the **while** loop in Listing 4.5 using a **do-while** loop, as shown in Listing 4.6.

LISTING 4.6 TestDoWhile.java

```
1 import java.util.Scanner;
2
3 public class TestDoWhile {
```

do-while loop

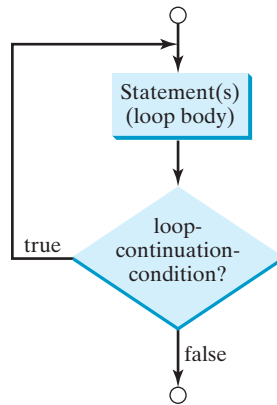


FIGURE 4.2 The **do-while** loop executes the loop body first, then checks the **loop-continuation-condition** to determine whether to continue or terminate the loop.

```

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 {
14         // Read the next data
15         System.out.print(
16             "Enter an integer (the input ends if it is 0): ";
17         data = input.nextInt();
18
19         sum += data;
20     } while (data != 0);
21
22     System.out.println("The sum is " + sum);
23 }
24 }
```

loop

end loop

```

Enter an integer (the input ends if it is 0): 3 ↵ Enter
Enter an integer (the input ends if it is 0): 5 ↵ Enter
Enter an integer (the input ends if it is 0): 6 ↵ Enter
Enter an integer (the input ends if it is 0): 0 ↵ Enter
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.



MyProgrammingLab™

4.6 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);
    }
}
```

4.7 What are the differences between a **while** loop and a **do-while** loop? Convert the following **while** loop into a **do-while** loop.

```
Scanner input = new Scanner(System.in);
int sum = 0;
System.out.println("Enter an integer " +
    "(the input ends if it is 0)");
int number = input.nextInt();
while (number != 0) {
    sum += number;
    System.out.println("Enter an integer " +
        "(the input ends if it is 0)");
    number = input.nextInt();
}
```

4.4 The for Loop



A **for** loop has a concise syntax for writing loops.

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 as:

```
for (i = initialValue; i < endValue; i++) {
    // Loop body
    ...
}
```

In general, the syntax of a **for** loop is:

```
for (initial-action; loop-continuation-condition;
    action-after-each-iteration) {
    // Loop body;
    Statement(s);
}
```

for loop

The flowchart of the **for** loop is shown in Figure 4.3a.

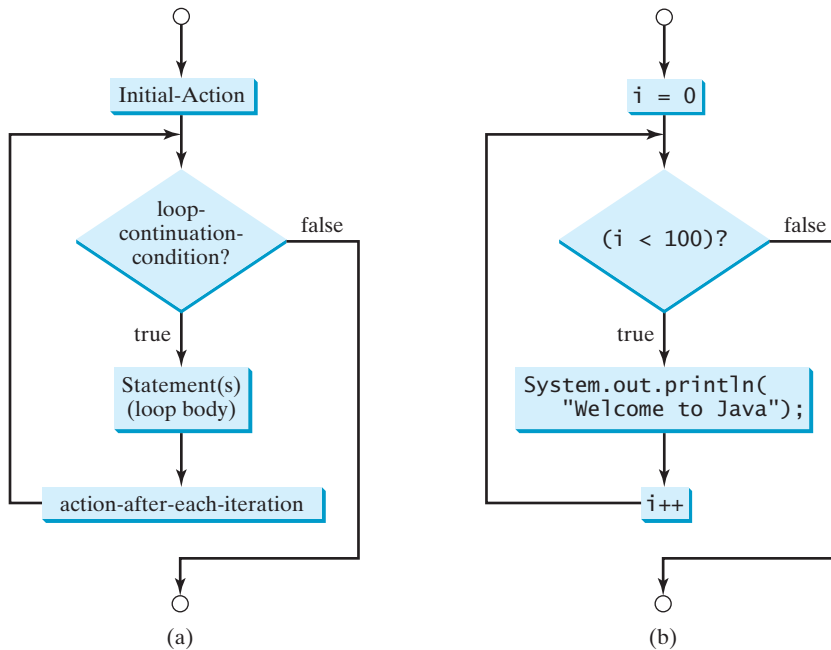


FIGURE 4.3 A **for** loop performs an initial action once, then repeatedly executes the statements in the loop body, and performs an action after an iteration when the **loop-continuation-condition** evaluates to **true**.

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 flowchart of the statement is shown in Figure 4.3b. 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.

The **action-after-each-iteration**, **i++**, is a statement that adjusts the control variable. This statement is executed after each iteration and 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.


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.

**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.

**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
}
```

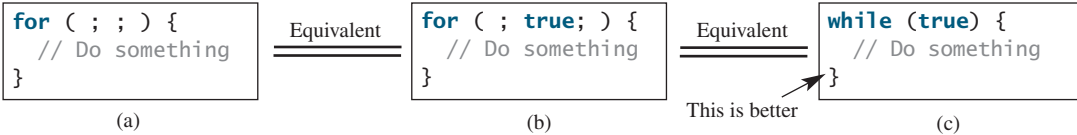
The **action-after-each-iteration** in a **for** loop can be a list of zero or more comma-separated statements. For example:

```
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).





4.8 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)

4.9 What are the three parts of a **for** loop control? Write a **for** loop that prints the numbers from **1** to **100**.

4.10 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, 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);
    }
}
```

4.11 What does the following statement do?

```
for ( ; ; ) {
    // Do something
}
```

4.12 If a variable is declared in the **for** loop control, can it be used after the loop exits?

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 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)

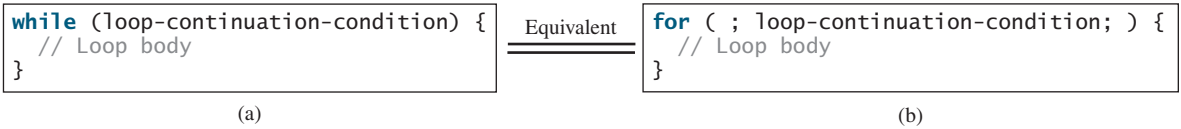
4.5 Which Loop to Use?

Key
Point

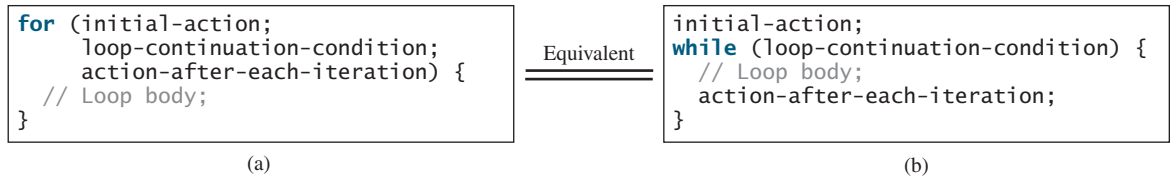
pretest loop
posttest loop

You can use a **for** loop, a **while** loop, or a **do-while** loop, whichever is convenient.

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).



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 Checkpoint Question 4.23 for such a case).

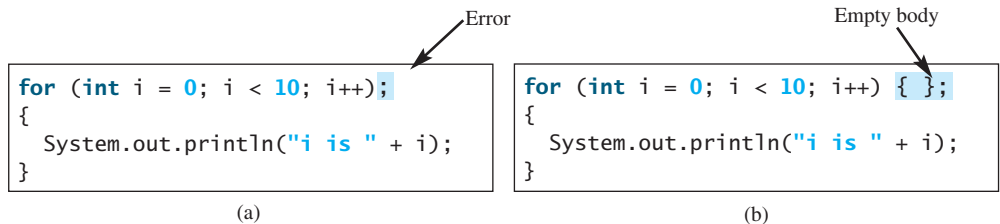


Use the loop statement that is most intuitive and comfortable for you. In general, a **for** loop may be used if the number of repetitions is known in advance, as, for example, when you need to display a message a hundred times. A **while** loop may be used if the number of repetitions is not fixed, 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.

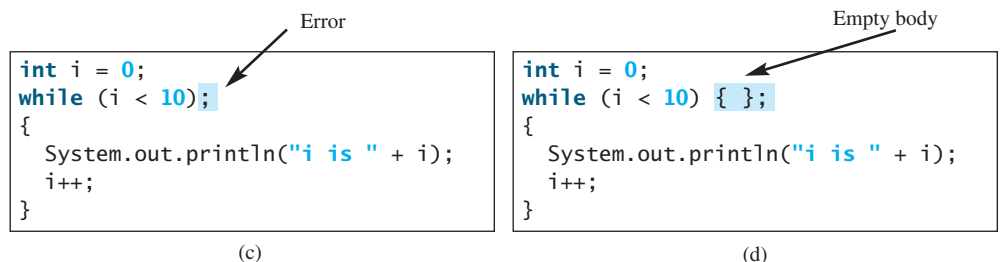


Caution

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. Both are incorrect.



Similarly, the loop in (c) is also wrong. (c) is equivalent to (d). Both are incorrect.



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

4.15 Can you convert a **for** loop to a **while** loop? List the advantages of using **for** loops.

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 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 }
```

4.18 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)

4.6 Nested Loops



A loop can be nested inside another loop.

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 4.7 presents a program that uses nested **for** loops to display a multiplication table.

LISTING 4.7 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");
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        // Display 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	72	81

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 dashed (–) 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**.

**Note**

Be aware that a nested loop may take a long time to run. Consider the following loop nested in three levels:

```
for (int i = 0; i < 10000; i++)
    for (int j = 0; j < 10000; j++)
        for (int k = 0; k < 10000; k++)
            Perform an action
```

The action is performed one trillion times. If it takes 1 microsecond to perform the action, the total time to run the loop would be more than 277 hours. Note that 1 microsecond is one millionth (10^{-6}) of a second.

4.19 How many times is the **println** statement executed?

```
for (int i = 0; i < 10; i++)
    for (int j = 0; j < i; j++)
        System.out.println(i * j)
```



MyProgrammingLab™

4.20 Show the output of the following programs. (*Hint: 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)

4.7 Minimizing Numeric Errors



Using floating-point numbers in the loop continuation condition may cause numeric errors.



VideoNote

Minimize numeric errors

Numeric errors involving floating-point numbers are inevitable. This section discusses how to minimize such errors through an example.

Listing 4.8 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 4.8 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 }
```

loop



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.

double precision

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 holds 64 bits, whereas a **float** variable holds 32 bits.

```

// 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;
```

numeric error

However, you will be stunned to see that the result is actually **49.50000000000003**. What went wrong? If you display **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:

```

double currentValue = 0.01;

for (int count = 0; count < 100; count++) {
```

```

sum += currentValue;
currentValue += 0.01;
}

```

After this loop, `sum` is `50.50000000000003`. This loop adds the numbers from smallest to biggest. What happens if you add numbers from biggest to smallest (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 `50.49999999999995`. Adding from biggest to smallest is less accurate than adding from smallest to biggest. 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 + 0.000000001` is `100000000.0`. To obtain more accurate results, carefully select the order of computation. Adding smaller numbers before bigger numbers is one way to minimize errors.

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 Case Study: Finding the Greatest Common Divisor

The greatest common divisor (gcd) of the two integers `4` and `2` is `2`. The greatest common divisor of the 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:

```

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 4.9 presents the program that prompts the user to enter two positive integers and finds their greatest common divisor.

LISTING 4.9 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: ");
input      11         int n1 = input.nextInt();
input      12         System.out.print("Enter second integer: ");
13         int n2 = input.nextInt();
14
gcd         15         int gcd = 1; // Initial gcd is 1
16         int k = 2; // Possible gcd
17         while (k <= n1 && k <= n2) {
check divisor 18             if (n1 % k == 0 && n2 % k == 0)
19                 gcd = k; // Update gcd
20             k++;
21         }
22
output      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 would you write this program? Would 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 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:

```

for (int k = 2; k <= n1 && k <= n2; k++) {
    if (n1 % k == 0 && n2 % k == 0)
        gcd = k;
}

```

A problem often has multiple solutions, and the gcd problem can be solved in many ways. Programming Exercise 4.14 suggests another solution. A more efficient solution is to use the classic Euclidean algorithm (see www.cut-the-knot.org/blue/Euclid.shtml for more information).

You might think that a divisor for a number **n1** cannot be greater than **n1 / 2** and would attempt to improve the program using the following loop:

```

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 Checkpoint Question 4.21 for the answer.

4.8.2 Case Study: Predicting 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**. Thus, the tuition for each year can be computed as follows:

```
double tuition = 10000;   int year = 0;   // Year 0
tuition = tuition * 1.07;  year++;         // Year 1
tuition = tuition * 1.07;  year++;         // Year 2
tuition = tuition * 1.07;  year++;         // Year 3
...
```

Keep computing the 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 0
int year = 0;
while (tuition < 20000) {
    tuition = tuition * 1.07;
    year++;
}
```

The complete program is shown in Listing 4.10.

LISTING 4.10 FutureTuition.java

```
1 public class FutureTuition {
2     public static void main(String[] args) {
3         double tuition = 10000;   // Year 0
4         int year = 0;
5         while (tuition < 20000) {
6             tuition = tuition * 1.07;
7             year++;
8         }
9
10        System.out.println("Tuition will be doubled in "
11            + year + " years");
12        System.out.printf("Tuition will be %.2f in %1d years",
13            tuition, year);
14    }
15 }
```

loop
next year's tuition

```
Tuition will be doubled in 11 years
Tuition will be $21048.52 in 11 years
```

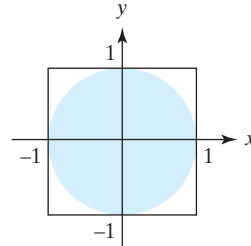


The **while** loop (lines 5–8) is used to repeatedly compute the tuition for a new year. The loop terminates when the tuition is greater than or equal to **20000**.

4.8.3 Case Study: 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 π .

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**. Therefore, 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 $\text{circleArea} / \text{squareArea} = \pi / 4$.

Write a program that randomly generates 1,000,000 points in the square and let **numberOfHits** denote the number of points that fall in the circle. Thus, **numberOfHits** is approximately $1000000 * (\pi / 4)$. π can be approximated as $4 * \text{numberOfHits} / 1000000$. The complete program is shown in Listing 4.11.

LISTING 4.11 MonteCarloSimulation.java

```

1  public class MonteCarloSimulation {
2      public static void main(String[] args) {
3          final int NUMBER_OF_TRIALS = 1000000;
4          int numberOfHits = 0;
5
6          for (int i = 0; i < NUMBER_OF_TRIALS; i++) {
7              generate random points
7              double x = Math.random() * 2.0 - 1;
8              double y = Math.random() * 2.0 - 1;
9              check inside circle
9              if (x * x + y * y <= 1)
10                 numberOfHits++;
11         }
12
13         estimate pi
13         double pi = 4.0 * numberOfHits / NUMBER_OF_TRIALS;
14         System.out.println("PI is " + pi);
15     }
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 $4 * \text{numberOfHits} / \text{NUMBER_OF_TRIALS}$ (line 13).

4.21 Will the program work if **n1** and **n2** are replaced by **n1 / 2** and **n2 / 2** in line 17 in Listing 4.9?



MyProgrammingLab™

4.9 Keywords **break** and **continue**

The **break** and **continue** keywords provide additional controls in a loop.



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 students' understanding of 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 4.12 presents a program to demonstrate the effect of using **break** in a loop.

break statement

LISTING 4.12 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;
11         }
12
13         System.out.println("The number is " + number);
14         System.out.println("The sum is " + sum);
15     }
16 }
```

break

```

The number is 14
The sum is 105
```



The program in Listing 4.12 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 and 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 4.13 presents a program to demonstrate the effect of using **continue** in a loop.

continue statement

LISTING 4.13 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 }

```

continue



The sum is 189

The program in Listing 4.13 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 Checkpoint Question 4.24). In general, though, using `break` and `continue` is appropriate if it simplifies coding and makes programs easier to read.

Suppose you need to write a program to find the smallest factor other than 1 for an integer `n` (assume `n >= 2`). You can write a simple and intuitive code using the `break` statement as follows:

```

int factor = 2;
while (factor <= n) {
    if (n % factor == 0)
        break;
    factor++;
}

```

```
System.out.println("The smallest factor other than 1 for "
    + n + " is " + factor);
```

You may rewrite the code without using **break** as follows:

```
boolean found = false;
int factor = 2;
while (factor <= n && !found) {
    if (n % factor == 0)
        found = true;
    else
        factor++;
}
System.out.println("The smallest factor other than 1 for "
    + n + " is " + factor);
```

Obviously, the **break** statement makes this program simpler and easier to read in this case. 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

4.22 What is the keyword **break** for? What is the keyword **continue** for? Will the following programs terminate? If so, give the output.



MyProgrammingLab™

```
int balance = 10;
while (true) {
    if (balance < 9)
        break;
    balance = balance - 9;
}

System.out.println("Balance is "
    + balance);
```

(a)

```
int balance = 10;
while (true) {
    if (balance < 9)
        continue;
    balance = balance - 9;
}

System.out.println("Balance is "
    + balance);
```

(b)

4.23 The **for** loop on the left is converted into the **while** loop on the right. What is wrong? Correct it.

```
for (int i = 0; i < 4; i++) {
    if (i % 3 == 0) continue;
    sum += i;
}
```

Converted
Wrong conversion

```
int i = 0;
while (i < 4) {
    if (i % 3 == 0) continue;
    sum += i;
    i++;
}
```


- 4.24** Rewrite the programs **TestBreak** and **TestContinue** in Listings 4.12 and 4.13 without using **break** and **continue**.
- 4.25** After the **break** statement in (a) is executed in the following loop, which statement is executed? Show the output. After the **continue** statement in (b) 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);
}
```

(a)

```
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);
}
```

(b)

4.10 Case Study: Displaying Prime Numbers



Key
Point

This section presents a program that displays the first fifty prime numbers in five lines, each containing ten 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.
- Display each prime number, and display 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 {
        Display 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, and so on up to **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) {                                count prime numbers
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++) {        check primeness
17                if (number % divisor == 0) { // If true, number is not prime
18                    isPrime = false; // Set isPrime to false
19                    break; // Exit the for loop                                exit loop
20                }
21            }
22
23            // Display the prime number and increase the count
24            if (isPrime) {                                                    display if prime
25                count++; // Increase the count
26
27                if (count % NUMBER_OF_PRIMES_PER_LINE == 0) {
28                    // Display 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
```

subproblem

This is a complex program for novice programmers. The key to developing a programmatic solution for this problem, and for 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.

To determine whether a number is prime, check whether it is divisible by a number between **2** and **number/2** inclusive (lines 16–21). 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.

4.1.1 Controlling a Loop with a Confirmation Dialog



Key Point

You can use a confirmation dialog to prompt the user to confirm whether to continue or exit a loop.

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:

```
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 4.5, `SentinelValue.java`, using a confirmation dialog box. A sample run is shown in Figure 4.4.

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 integer: ");
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 }

```

confirmation option
check option
input dialog
confirmation dialog
message dialog

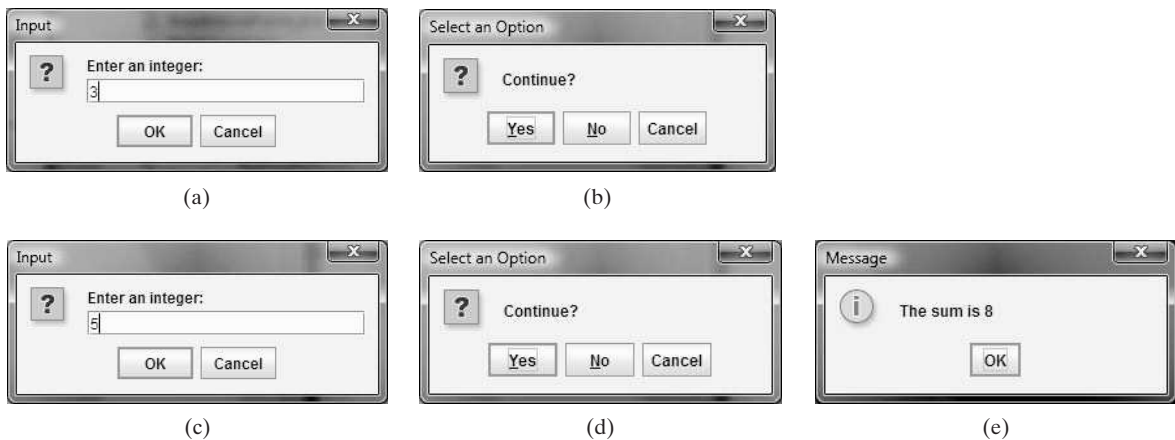


FIGURE 4.4 The user enters 3 in (a), clicks Yes in (b), enters 5 in (c), clicks No in (d), and the result is shown in (e).

The 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).

The **showConfirmDialog** method (line 17) returns an integer **JOptionPane.YES_OPTION**, **JOptionPane.NO_OPTION**, or **JOptionPane.CANCEL_OPTION**, if the user clicks *Yes*, *No*, or *Cancel*. The return value is assigned to the variable **option** (line 17). If this value is **JOptionPane.YES_OPTION**, the loop continues (line 9).

KEY TERMS

break statement	159	loop body	134
continue statement	159	nested loop	152
do-while loop	144	off-by-one error	136
for loop	147	output redirection	143
infinite loop	136	posttest loop	150
input redirection	143	pretest loop	150
iteration	134	sentinel value	141
loop	134	while loop	134

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. The **while** loop and the **do-while** loop often are used when the number of repetitions is not predetermined.
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 a *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.

TEST QUESTIONS

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

PROGRAMMING EXERCISES



Pedagogical Note

Read each problem several times until you understand it. Think how to solve the problem before starting to write code. Translate your logic into a program.

A problem often can be solved in many different ways. Students are encouraged to explore various solutions.

Sections 4.2–4.7

- *4.1** (*Count positive and negative numbers and compute 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 integer, the input ends if it is 0: 1 2 -1 3 0 Enter
The number of positives is 3
The number of negatives is 1
The total is 5
The average is 1.25
```



```
Enter an integer, the input ends if it is 0: 0 Enter
No numbers are entered except 0
```



- 4.2** (*Repeat additions*) Listing 4.4, `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 and pounds to kilograms*) Write a program that displays the following two tables side by side (note that 1 kilogram is 2.2 pounds and that 1 pound is .453 kilograms):

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 and that 1 kilometer is .621 miles):

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: compute 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** (*Find 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** (*Find 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** (*Find numbers divisible by 5 and 6*) Write a program that displays all the numbers from 100 to 1,000, ten per line, that are divisible by 5 and 6. Numbers are separated by exactly one space.
- 4.11** (*Find 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. Numbers are separated by exactly one space.
- 4.12** (*Find 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** (*Find 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.

Sections 4.8–4.10

- *4.14** (*Compute the greatest common divisor*) Another solution for Listing 4.9 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.
- *4.15** (*Display the ASCII character table*) Write a program that prints the characters in the ASCII character table from **!** to **~**. Display ten characters per line. The ASCII table is shown in Appendix B. Characters are separated by exactly one space.
- *4.16** (*Find 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** (*Display 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 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** (Display four patterns using loops) Use nested loops that display the following patterns in four separate programs:

Pattern A	Pattern B	Pattern C	Pattern D
1	1 2 3 4 5 6	1	1 2 3 4 5 6
1 2	1 2 3 4 5	2 1	1 2 3 4 5
1 2 3	1 2 3 4	3 2 1	1 2 3 4
1 2 3 4	1 2 3	4 3 2 1	1 2 3
1 2 3 4 5	1 2	5 4 3 2 1	1 2
1 2 3 4 5 6	1	6 5 4 3 2 1	1

****4.19** (Display 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
    
```

***4.20** (Display prime numbers between 2 and 1,000) Modify Listing 4.14 to display all the prime numbers between 2 and 1,000, inclusive. Display eight prime numbers per line. Numbers are separated by exactly one space.

Comprehensive

****4.21** (Financial application: compare 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: 10000 
Number of Years: 5 
Interest Rate    Monthly Payment    Total Payment
5.000%          188.71              11322.74
5.125%          189.28              11357.13
5.250%          189.85              11391.59
...
7.875%          202.17              12129.97
8.000%          202.76              12165.83
    
```



For the formula to compute monthly payment, see Listing 2.8, ComputeLoan.java.



VideoNote
Display loan schedule

****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:



```
Loan Amount: 10000  ↵ Enter
Number of Years: 1  ↵ Enter
Annual Interest Rate: 7  ↵ Enter

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 display 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** (*Obtain 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 computes the results of the summation of the preceding series from left to right and from right to left with **n = 50000**.

***4.24** (*Sum a series*) Write a program to sum the following series:

$$\frac{1}{3} + \frac{3}{5} + \frac{5}{7} + \frac{7}{9} + \frac{9}{11} + \frac{11}{13} + \dots + \frac{95}{97} + \frac{97}{99}$$



VideoNote
Sum a series

****4.25** (Compute π) You can approximate π by using the following series:

$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots + \frac{(-1)^{i+1}}{2i-1} \right)$$

Write a program that displays the π value for $i = 10000, 20000, \dots$, and **100000**.

****4.26** (Compute 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 $i = 10000, 20000, \dots$, and **100000**. (Hint: Because $i! = i \times (i-1) \times \dots \times 2 \times 1$, then

$$\frac{1}{i!} \text{ is } \frac{1}{i(i-1)!}$$

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 $i = 2, 3, 4, \dots$)

****4.27** (Display leap years) Write a program that displays all the leap years, ten per line, in the twenty-first century (from 2001 to 2100), separated by exactly one space.

****4.28** (Display 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 **2013**, and **2** for Tuesday, January 1, 2013, your program should display the following output:

```
January 1, 2013 is Tuesday
...
December 1, 2013 is Sunday
```

****4.29** (Display 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 **2013**, and **2** for Tuesday, January 1, 2013, your program should display the calendar for each month in the year, as follows:

January 2013						
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 2013						
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				

- *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: compute 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 perfect number because $6 = 3 + 2 + 1$. The next is $28 = 14 + 7 + 4 + 2 + 1$. There are four perfect numbers less than 10,000. 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 to compute 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 3.9.
- **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: find 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 number of sales you have to generate in order to make \$30,000.

- 4.40** (*Simulation: heads or tails*) 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 3 5 2 5 5 5 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
The largest number is 5
The occurrence count of the largest number is 4
```



***4.42** (Financial application: find 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.

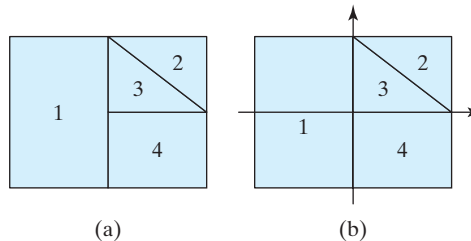
***4.43** (Simulation: clock countdown) Write a program that prompts the user to enter the number of seconds, displays a message at every second, and terminates when the time expires. Here is a sample run:



```
Enter the number of seconds: 3
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 1,000,000 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 for a point to fall into an odd-numbered region.)



***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
...
...

The total number of all combinations is 21
```

***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 0000000000000101
```

```
Enter an integer: -5
The bits are 1111111111111011
```



(Hint: You need to use the bitwise right shift operator (\gg) and the bitwise AND operator ($\&$), which are covered in Appendix G, Bitwise Operations.)

****4.47** (Statistics: compute 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:

$$\text{mean} = \frac{\sum_{i=1}^n x_i}{n} = \frac{x_1 + x_2 + \dots + x_n}{n} \quad \text{deviation} = \sqrt{\frac{\sum_{i=1}^n x_i^2 - \frac{\left(\sum_{i=1}^n x_i\right)^2}{n}}{n - 1}}$$

Here is a sample run:

```
Enter ten numbers: 1 2 3 4.5 5.6 6 7 8 9 10
The mean is 5.61
The standard deviation is 2.99794
```



This page intentionally left blank

METHODS

Objectives

- To define methods with formal parameters (§5.2).
- To invoke methods with actual parameters (i.e., arguments) (§5.2).
- To define methods with a return value (§5.3).
- To define 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 hexadecimal (§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



Methods can be used to define reusable code and organize and simplify code.

problem

Suppose that you need to find the sum of integers from **1** to **10**, from **20** to **37**, and from **35** to **49**, respectively. You may write the code as follows:

```
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 <= 37; i++)
    sum += i;
System.out.println("Sum from 20 to 37 is " + sum);

sum = 0;
for (int i = 35; i <= 49; i++)
    sum += i;
System.out.println("Sum from 35 to 49 is " + sum);
```

why methods?

You may have observed that computing these sums from **1** to **10**, from **20** to **37**, and from **35** to **49** 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? We can do so by defining a method and invoking it.

The preceding code can be simplified as follows:

define sum method

```
1 public static int sum(int i1, int i2) {
2     int result = 0;
3     for (int i = i1; i <= i2; i++)
4         result += i;
5
6     return result;
7 }
```

main method
invoke sum

```
8
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 37 is " + sum(20, 37));
12     System.out.println("Sum from 35 to 49 is " + sum(35, 49));
13 }
```

method

Lines 1–7 define the method named **sum** with two parameters **i1** and **i2**. The statements in the **main** method invoke **sum(1, 10)** to compute the sum from **1** to **10**, **sum(20, 37)** to compute the sum from **20** to **37**, and **sum(35, 49)** to compute the sum from **35** to **49**.

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**, **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



A method definition consists of its method name, parameters, return value type, and body.

The syntax for defining a method is:

```
modifier returnType methodName(list of parameters) {
    // Method body;
}
```

Let's look at a method defined to find the larger between two integers. 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.

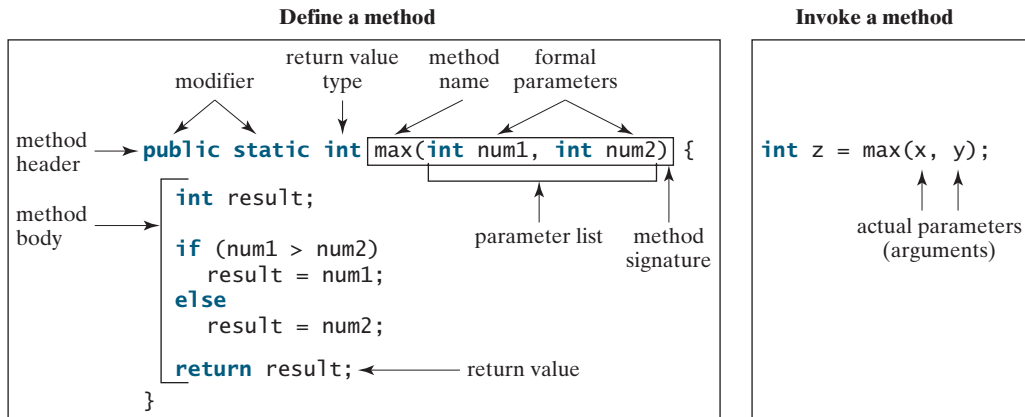


FIGURE 5.1 A method definition consists of a method header and a method body.

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.

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 called a *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 method's type, order, and number of the parameters. The method name and the parameter list together constitute the *method signature*. Parameters are optional; that is, a method doesn't have to contain any parameters. For example, the **Math.random()** method has no parameters.

The method body contains a collection of statements that implement the method. 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

Some programming languages refer to methods as *procedures* and *functions*. In those languages, a value-returning method is called a *function* and a void method is called a *procedure*.



Caution

In the method header, you need to declare each parameter separately. For instance, **max(int num1, int num2)** is correct, but **max(int num1, num2)** is wrong.

method header
modifier

value-returning method
void method
formal parameter
parameter
actual parameter
argument
parameter list
method signature

define vs. declare

**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.

5.3 Calling a Method

**Key Point**

Calling a method executes the code in the method.

In a method definition, you define what the method is to do. To execute the 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 a 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 a 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 it 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.

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 of " + i +
8              " and " + j + " is " + k);
9      }
10
11     /** Return the max of 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 }
```

**VideoNote**

Define/invoke `max` method

main method

invoke `max`

define method

The maximum of 5 and 2 is 5



	line#	i	j	k	num1	num2	result
	4	5					
	5		2				
Invoking max	12				5	2	
	13						undefined
	16						5
	6			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 to start the program.

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, and 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 **main** method). This process is illustrated in Figure 5.2.

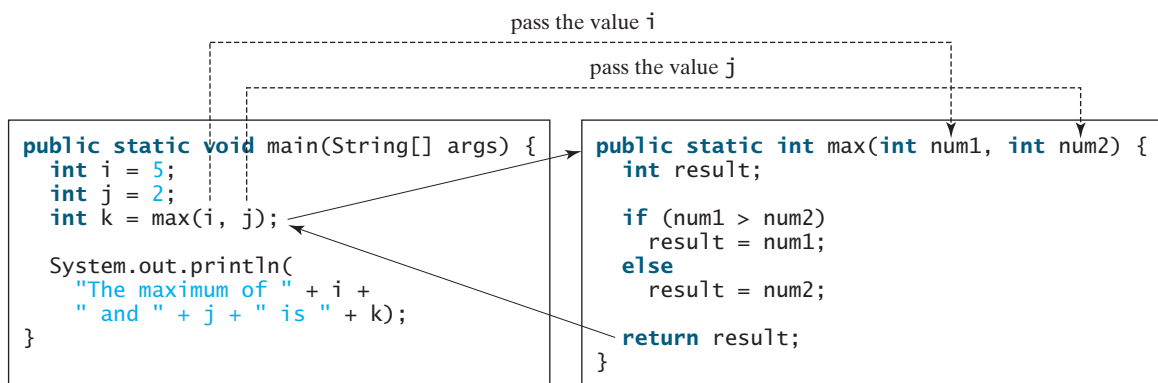
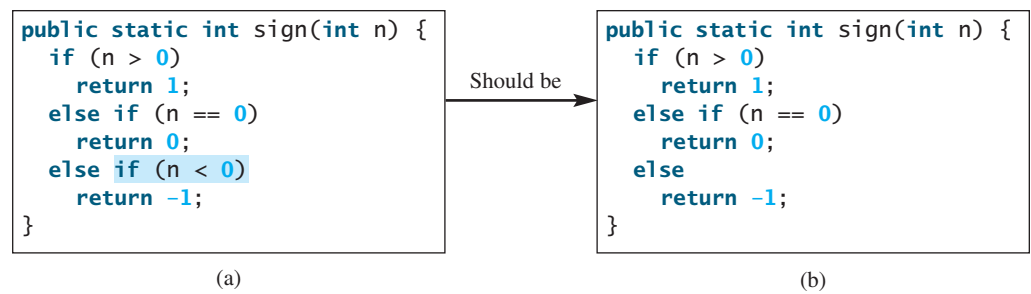


FIGURE 5.2 When the **max** method is invoked, the flow of control transfers to it. Once the **max** method is finished, it returns control back to the caller.



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 that this method might not return a value.



To fix this problem, delete `if (n < 0)` in (a), so the compiler will see a `return` statement to be reached regardless of how the `if` statement is evaluated.



Note

Methods enable code sharing and reuse. The `max` method can be invoked from any class, not just `TestMax`. If you create a new class, you can invoke the `max` method using `ClassName.methodName` (i.e., `TestMax.max`).

reusing method

activation record

call stack

Each time a method is invoked, the system creates an *activation record* (also called an *activation frame*) that stores parameters and variables for the method and places the activation record in an area of memory known as a *call stack*. A call stack is also known as an *execution stack*, *runtime stack*, or *machine stack*, and it is often shortened to just “the stack.” When a method calls another method, the caller’s activation record is kept intact, and a new activation record is created for the new method called. When a method finishes its work and returns to its caller, its activation record is removed from the call stack.

A call stack stores the activation records in a last-in, first-out fashion: The activation record for the method that is invoked last is removed first from the stack. For example, suppose method `m1` calls method `m2`, and then `m3`. The runtime system pushes `m1`’s activation record into the stack, then `m2`’s, and then `m3`’s. After `m3` is finished, its activation record is removed from the stack. After `m2` is finished, its activation record is removed from the stack. After `m1` is finished, its activation record is removed from the stack.

Understanding call stacks helps you to comprehend how methods are invoked. The variables defined in the `main` method in Listing 5.1 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 `max` method. Their values are passed through method invocation. Figure 5.3 illustrates the activation records for method calls in the stack.

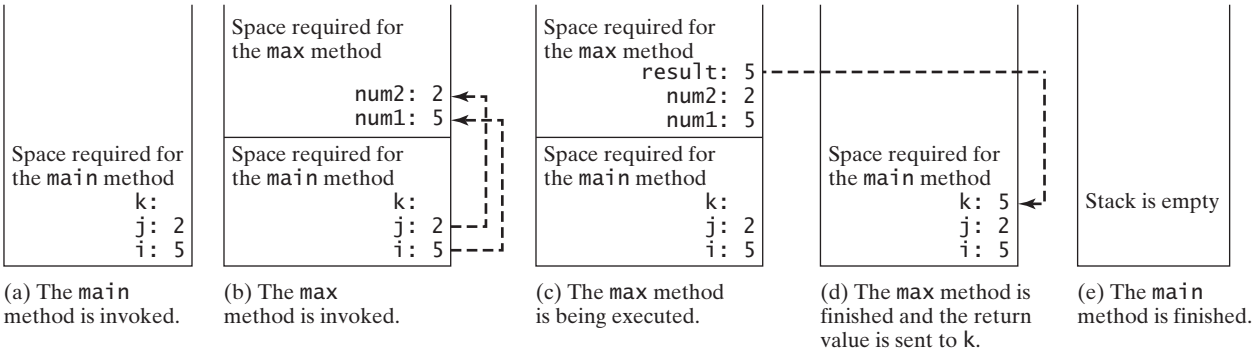


FIGURE 5.3 When the `max` method is invoked, the flow of control transfers to the `max` method. Once the `max` method is finished, it returns control back to the caller.

5.4 void Method Example

A **void** method does not return a value.

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.



VideoNote

Use void method

LISTING 5.2 TestVoidMethod.java

```

1  public class TestVoidMethod {
2      public static void main(String[] args) {
3          System.out.print("The grade is ");
4          printGrade(78.5);
5
6          System.out.print("The grade is ");
7          printGrade(59.5);
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 }
```

main method

invoke printGrade

printGrade method

```
The grade is C
The grade is F
```



The **printGrade** method is a **void** method because it does not return any value. A call to a **void** method must be a statement. Therefore, 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 value-returning method, let's 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. value-returned

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  }
```

main method

invoke getGrade

getGrade method

```

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 }
```



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, so it must be invoked as a statement.

return in void method



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.

```

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');
    }
}
```



- 5.1** What are the benefits of using a method?
- 5.2** How do you define a method? How do you invoke a method?
- 5.3** How do 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 cannot be a statement by itself.
- 5.5** What is the **return** type of a **main** method?
- 5.6** 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;
}
```

- 5.7** Define the terms parameter, argument, and method signature.
- 5.8** Write method headers (not the bodies) for the following methods:
- Compute a sales commission, given the sales amount and the commission rate.
 - Display the calendar for a month, given the month and year.
 - Compute a square root of a number.
 - Test whether a number is even, and returning **true** if it is.
 - Display a message a specified number of times.
 - Compute the monthly payment, given the loan amount, number of years, and annual interest rate.
 - Find the corresponding uppercase letter, given a lowercase letter.
- 5.9** 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 }
```

- 5.10** Reformat the following program according to the programming style and documentation guidelines proposed in Section 1.10, 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;
    }
}
```


5.5 Passing Parameters by Values



The arguments are passed by value to parameters when invoking a method.

parameter order association

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 of 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:

```
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 argument, `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.

pass-by-value

When you invoke a method with an argument, 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). The parameter `n` is incremented by `1` in the method (line 10), but `x` is not changed no matter what the method does.

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             + " and 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             + " and n2 is " + n2);
31     }
32 }

```

false swap

```

Before invoking the swap method, num1 is 1 and num2 is 2
Inside the swap method
Before swapping, n1 is 1 and n2 is 2
After swapping, n1 is 2 and 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.

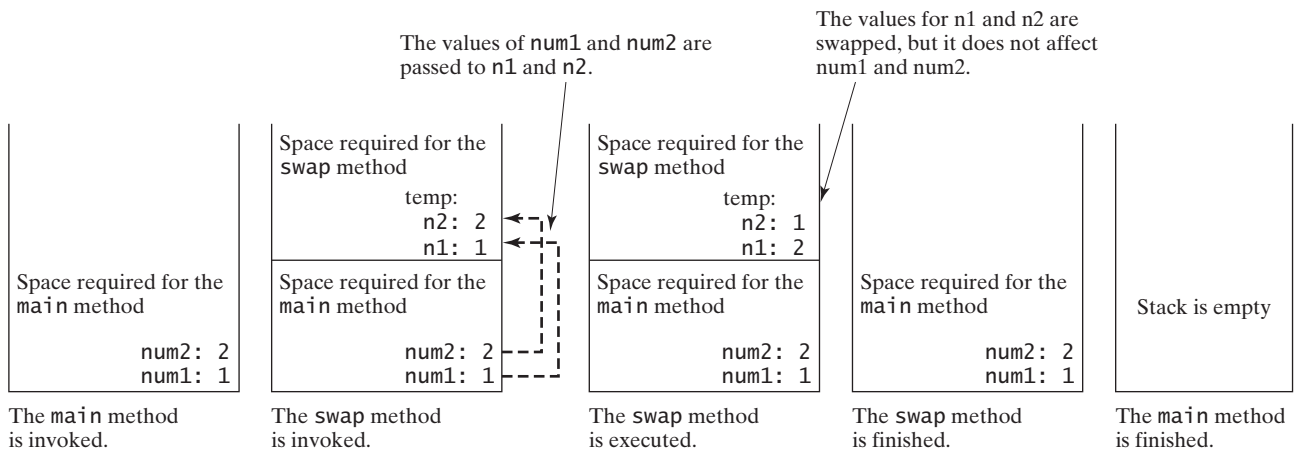


FIGURE 5.4 The values of the variables are passed to the method's parameters.



Note

For simplicity, Java programmers often say *passing x to y*, which actually means *passing the value of argument x to parameter y*.



5.11 How is an argument passed to a method? Can the argument have the same name as its parameter?

MyProgrammingLab™

5.12 Identify and correct the errors in the following program:

```
1 public class Test {
2     public static void main(String[] args) {
3         nPrintln(5, "Welcome to Java!");
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 }
```

5.13 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;
    }
}
```

(a)

```
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();
        }
    }
}
```

(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)

5.14 For (a) in the preceding question, show the contents of the activation records in the call stack just before the method `max` is invoked, just as `max` is entered, just before `max` is returned, and right after `max` is returned.

5.6 Modularizing Code

Modularizing makes the code easy to maintain and debug and enables the code to be reused.

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.

Listing 4.9 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.



VideoNote

Modularize code

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));
17  }
18
19  /** Return the gcd of two integers */
20  public static int gcd(int n1, int n2) {
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
31  }
32  }

```

invoke gcd

compute gcd

return gcd



```

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 the 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

```

invoke printPrimeNumbers

printPrimeNumbers method

invoke isPrime

```

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 }

```

isPrime method

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: determining whether a number is a prime and printing the prime numbers. 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 Case Study: Converting Decimals to Hexadecimals

This section presents a program that converts a decimal number to a hexadecimal number.

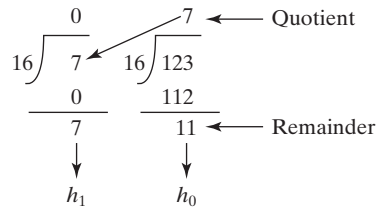


Hexadecimals are often used in computer systems programming (see Appendix F for an introduction to number systems). To convert a decimal number d to a hexadecimal number is to find the hexadecimal digits $h_n, h_{n-1}, h_{n-2}, \dots, h_2, h_1$, and h_0 such that

$$\begin{aligned}
 d = & h_n \times 16^n + h_{n-1} \times 16^{n-1} + h_{n-2} \times 16^{n-2} + \dots \\
 & + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0
 \end{aligned}$$

These hexadecimal digits 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 . The hexadecimal digits include the decimal digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9, plus A, which is the decimal value 10; B, which is the decimal value 11; C, which is 12; D, which is 13; E, which is 14; and F, which is 15.

For example, the decimal number **123** is **7B** in hexadecimal. The conversion is done as follows. Divide **123** by **16**. The remainder is **11** (**B** in hexadecimal) and the quotient is **7**. Continue to divide **7** by **16**. The remainder is **7** and the quotient is **0**. Therefore **7B** is the hexadecimal number for **123**.



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();
12
13         System.out.println("The hex number for 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 }

```

input decimal

decimal to hex

get a hex char

get a letter

```
Enter a decimal number: 1234 [Enter]
The hex number for decimal 1234 is 4D2
```



	line#	decimal	hex	hexValue	toHexChar(hexValue)
iteration 1	19	1234	""		
	22			2	
	23		"2"		2
iteration 2	24	77			
	22			13	
	23		"D2"		D
iteration 3	24	4			
	22			4	
	23		"4D2"		4
	24	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 `decimalToHex` 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 that 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 - 10 + 'A')` (line 35). For instance, if `hexValue` is 11, `(char)(hexValue - 10 + 'A')` returns B.

- 5.15 What is the return value from invoking `toHexChar(5)`? What is the return value from invoking `toHexChar(15)`?
- 5.16 What is the return value from invoking `decimalToHex(245)`? What is the return value from invoking `decimalToHex(3245)`?



MyProgrammingLab™

5.8 Overloading Methods

Overloading methods enables you to define the methods with the same name as long as their signatures are different.



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;
}
```


method overloading

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 to use based on the method signature.

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

overloaded max

overloaded max

overloaded max

```

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 of 3 and 4 is "
6              + max(3, 4));
7
8          // Invoke the max method with the double parameters
9          System.out.println("The maximum of 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 of 3.0, 5.4, and 10.14 is "
14             + max(3.0, 5.4, 10.14));
15     }
16
17     /** Return the max of two int values */
18     public static int max(int num1, int num2) {
19         if (num1 > num2)
20             return num1;
21         else
22             return num2;
23     }
24
25     /** Find the max of two double values */
26     public static double max(double num1, double num2) {
27         if (num1 > num2)
28             return num1;
29         else
30             return num2;
31     }
32
33     /** Return the max of three double values */
34     public static double max(double num1, double num2, double num3) {
35         return max(max(num1, num2), num3);
36     }
37 }

```



```

The maximum of 3 and 4 is 4
The maximum of 3.0 and 5.4 is 5.4
The maximum of 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 question 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)`.



Tip

Overloading methods can make programs clearer and more readable. Methods that perform the same function with different types of parameters should be given the same name.



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)`. Because neither is more specific than the other, the invocation is ambiguous, resulting in a compile error.

5.17 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?

5.18 What is wrong in the following program?

```
public class Test {
    public static void method(int x) {
    }
}
```



MyProgrammingLab™

```

    public static int method(int y) {
        return y;
    }
}

```

5.19 Given two method definitions,

```

public static double m(double x, double y)

public static double m(int x, double y)

```

tell which of the two methods is invoked for:

- `double z = m(4, 5);`
- `double z = m(4, 5.4);`
- `double z = m(4.5, 5.4);`

5.9 The Scope of Variables



The scope of a variable is the part of the program where the variable can be referenced.

scope of a variable
local variable

Section 2.5 introduced the scope of a variable. This section discusses the scope of variables in more details. A variable defined inside a method is referred to as 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 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. However, 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.

```

    public static void method1() {
        .
        .
        for (int i = 1; i < 10; i++) {
            .
            .
            int j;
            .
            .
        }
    }

```

The scope of i →

The scope of j →

FIGURE 5.5 A variable declared in the initial action part of a **for**-loop header has its scope in the entire loop.

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.

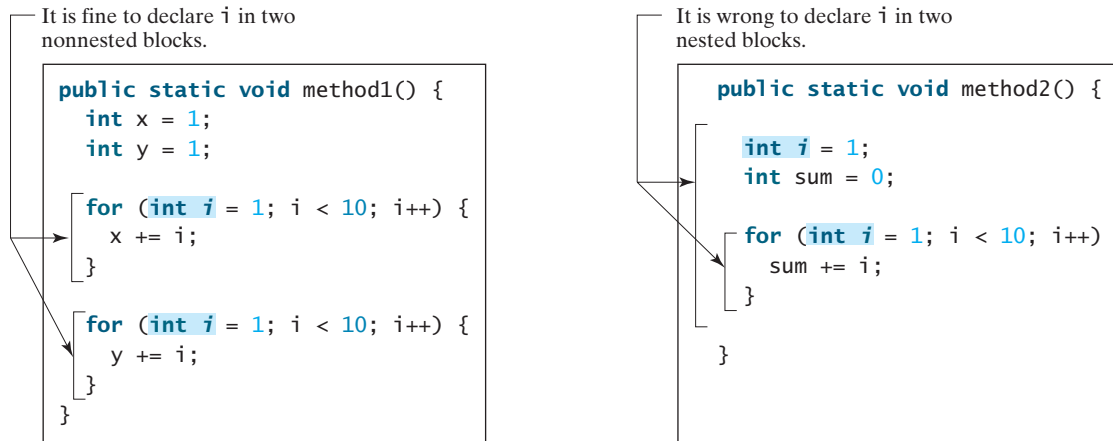


FIGURE 5.6 A variable can be declared multiple times in nonnested blocks, but only once in nested blocks.



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.

5.20 What is a local variable?

5.21 What is the scope of a local variable?



MyProgrammingLab™

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 a^b in Section 2.9.3, Exponent Operations, and the `Math.random()` method in Section 3.8, Generating Random Numbers. This section introduces other useful methods in the **Math** class. They can be categorized as *trigonometric methods*, *exponent methods*, and *service methods*. Service methods include the rounding, min, max, absolute, and random methods. In addition to 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 radians */
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 $-\pi/2$ and $\pi/2$. One degree is equal to $\pi/180$ in radians, 90 degrees is equal to $\pi/2$ in radians, and 30 degrees is equal to $\pi/6$ in radians.

For example,

```

Math.toDegrees(Math.PI / 2) returns 90.0
Math.toRadians(30) returns 0.5236 (same as  $\pi/6$ )
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 0.523598333 (same as  $\pi/6$ )

```

5.10.2 Exponent Methods

There are five methods related to exponents in the `Math` class:

```

/** Return e raised to the power of x ( $e^x$ ) */
public static double exp(double x)

/** Return the natural logarithm of x ( $\ln(x) = \log_e(x)$ ) */
public static double log(double x)

/** Return the base 10 logarithm of x ( $\log_{10}(x)$ ) */
public static double log10(double x)

/** Return a raised to the power of b ( $a^b$ ) */
public static double pow(double a, double b)

/** Return the square root of x  $\sqrt{x}$  for  $x \geq 0$  */
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 // Returns int
Math.round(-2.6) returns -3 // Returns long
Math.round(-2.4) returns -2 // Returns long
```

5.10.4 The **min**, **max**, and **abs** Methods

The **min** and **max** methods are overloaded to return the minimum and maximum numbers of 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**, or **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`



Tip

You can view the complete documentation for the `Math` class online at download.oracle.com/javase/7/docs/api/, as shown in Figure 5.7.



Note

Not all classes need a `main` method. The `Math` class and the `JOptionPane` class do not have `main` methods. These classes contain methods for other classes to use.

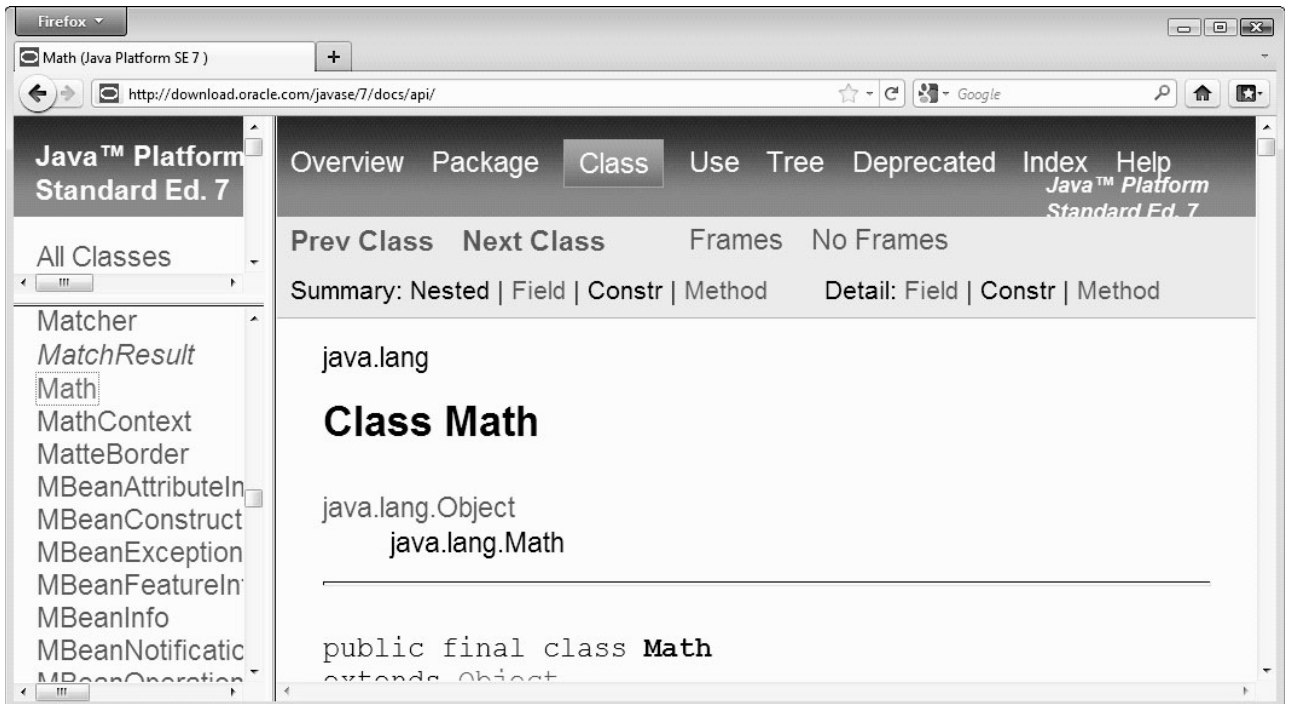
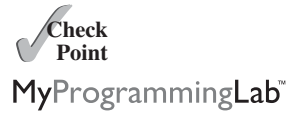


FIGURE 5.7 You can view the documentation for the Java API online.

- 5.22** True or false? The argument for trigonometric methods is an angle in radians.
- 5.23** Write an expression that obtains a random integer between 34 and 55. Write an expression that obtains a random integer between 0 and 999. Write an expression that obtains a random number between 5.5 and 55.5. Write an expression that obtains a random lowercase letter.
- 5.24** Evaluate the following method calls:

- | | |
|---|--|
| a. <code>Math.sqrt(4)</code> | j. <code>Math.floor(-2.5)</code> |
| b. <code>Math.sin(2 * Math.PI)</code> | k. <code>Math.round(-2.5f)</code> |
| c. <code>Math.cos(2 * Math.PI)</code> | l. <code>Math.round(-2.5)</code> |
| d. <code>Math.pow(2, 2)</code> | m. <code>Math rint(2.5)</code> |
| e. <code>Math.log(Math.E)</code> | n. <code>Math.ceil(2.5)</code> |
| f. <code>Math.exp(1)</code> | o. <code>Math.floor(2.5)</code> |
| g. <code>Math.max(2, Math.min(3, 4))</code> | p. <code>Math.round(2.5f)</code> |
| h. <code>Math.rint(-2.5)</code> | q. <code>Math.round(2.5)</code> |
| i. <code>Math.ceil(-2.5)</code> | r. <code>Math.round(Math.abs(-2.5))</code> |



5.11 Case Study: Generating Random Characters

A character is coded using an integer. Generating a random character is to generate an integer.



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 of generating random characters.

As introduced in Section 2.17, 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's 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'
```

Thus, a random integer between `(int)'a'` and `(int)'z'` is

```
(int)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))
```

As discussed in Section 2.17.3, 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. Therefore, 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))
```


Hence, 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. Listing 5.10 creates a class named `RandomCharacter` 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) {
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 }

```

getRandomCharacter

getRandomLowercaseLetter()

getRandomUppercaseLetter()

getRandomDigitCharacter()

getRandomCharacter()

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 }

```

constants

lower-case letter

```
gmjsohezfkgtazqgmswfc1rao
pnrunulnwmaztlfjedmpchcif
1alqdgivxkxpbzulrmqmbhikr
lbnrjlsopfxahssqhwuuljvbe
xbhdotzhpehbqmuwsfktwsoli
cbuwkzgxpmtzihgatslvwbz
bfesoklwbhnooygiigzduqni
```



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

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 also 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.



Key
Point



VideoNote

Stepwise refinement

method abstraction

information hiding

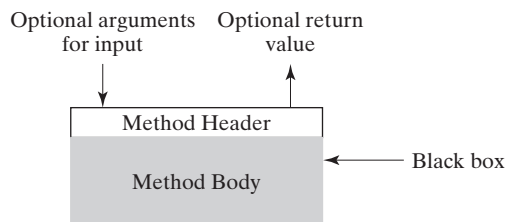


FIGURE 5.8 The method body can be thought of as a black box that contains the detailed implementation for the method.

You have already used the `System.out.print` method to display a string 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.

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.

divide and conquer
stepwise refinement



```
Enter full year (e.g., 2012): 2012 ↵ Enter
Enter month as a number between 1 and 12: 3 ↵ Enter

      March 2012
-----
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
```

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 5.9a).

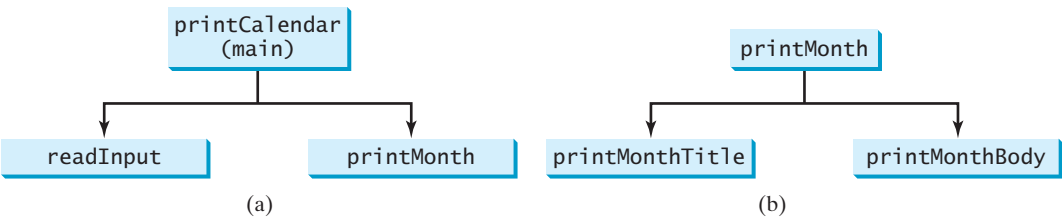


FIGURE 5.9 The structure chart shows that the `printCalendar` problem is divided into two subproblems, `readInput` and `printMonth` in (a), and that `printMonth` is divided into two smaller subproblems, `printMonthTitle` and `printMonthBody` in (b).

You can use `Scanner` to read input for the year and the month. 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 5.9b. The month title consists of three lines: month and year, a dashed 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 5.10a).

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`),

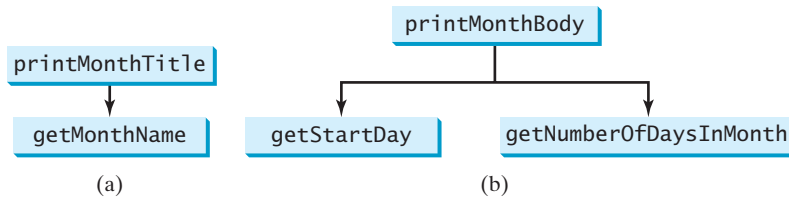


FIGURE 5.10 (a) To `printMonthTitle`, you need `getMonthName`. (b) The `printMonthBody` problem is refined into several smaller problems.

as shown in Figure 5.10b. For example, December 2013 has 31 days, and December 1, 2013, is a Sunday.

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 an alternative approach. Assume you know that the start day for January 1, 1800, was a Wednesday ($\text{START_DAY_FOR_JAN_1_1800} = 3$). You could compute the total number of days (`totalNumberOfDays`) between January 1, 1800, and the first date of the calendar month. The start day for the calendar month is $(\text{totalNumberOfDays} + \text{startDay1800}) \% 7$, since every week has seven days. Thus, the `getStartDay` problem can be further refined as `getTotalNumberOfDays`, as shown in Figure 5.11a.

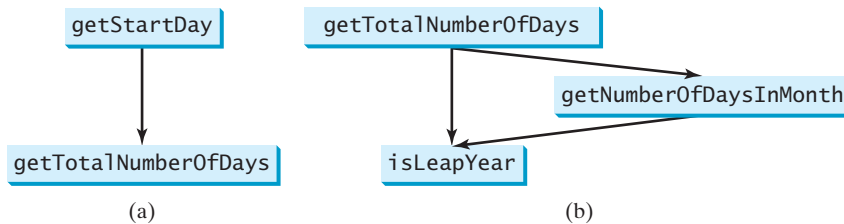


FIGURE 5.11 (a) To `getStartDay`, you need `getTotalNumberOfDays`. (b) The `getTotalNumberOfDays` problem is refined into two smaller problems.

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. Thus, `getTotalNumberOfDays` can be further refined into two subproblems: `isLeapYear` and `getNumberOfDaysInMonth`, as shown in Figure 5.11b. The complete structure chart is shown in Figure 5.12.

5.12.2 Top-Down and/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*—a simple but incomplete version of a method—can be used for the methods waiting to be implemented. 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,

top-down approach
stub

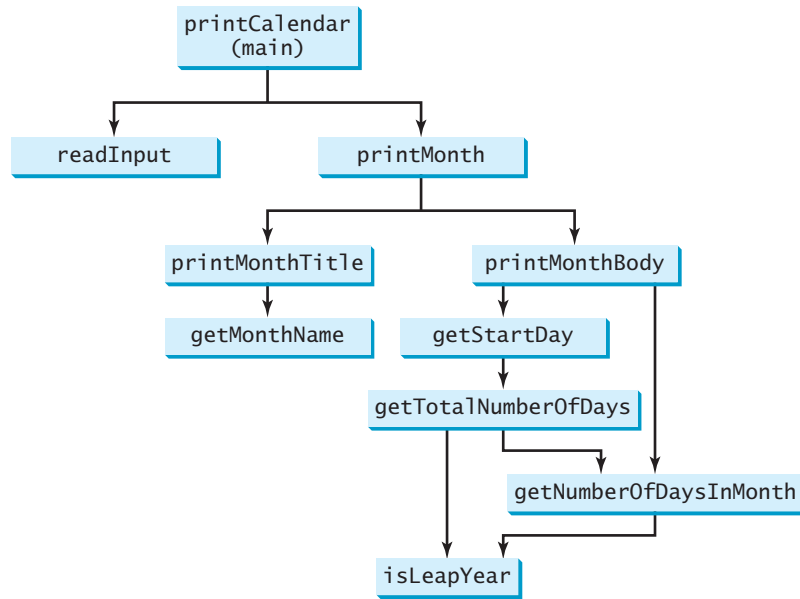


FIGURE 5.12 The structure chart shows the hierarchical relationship of the subproblems in the program.

let **printMonth** display the year and the month in the stub. Thus, your program may begin like this:

```

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., 2012): ");
        int year = input.nextInt();

        // Prompt the user to enter month
        System.out.print("Enter month as a 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 getStartDay 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 isLeapYear 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, known as the *driver*, to test it. The top-down and bottom-up approaches are equally good: Both approaches implement methods incrementally, help to isolate programming errors, and make debugging easy. They can be used together.

bottom-up approach
driver

5.12.3 Implementation Details

The `isLeapYear(int year)` method can be implemented using the following code from Section 3.12:

```
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., 2012): ");
10         int year = input.nextInt();
11
12         // Prompt the user to enter month
13         System.out.print("Enter month as a 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) {
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., March 2012 */
30     public static void printMonthTitle(int year, int month) {
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) {
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 */

```

printMonth

printMonthTitle

getMonthName

```

59 public static void printMonthBody(int year, int month) {           printMonthBody
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("%4d", 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) {               getStartDay
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) {      getTotalNumberOfDays
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             total = total + 366;
99         else
100            total = total + 365;
101
102     // Add days from Jan to the month prior to the calendar month
103     for (int i = 1; i < month; i++)
104         total = total + getNumberOfDaysInMonth(year, i);
105
106     return total;
107 }
108
109 /** Get the number of days in a month */
110 public static int getNumberOfDaysInMonth(int year, int month) {    getNumberOfDaysInMonth
111     if (month == 1 || month == 3 || month == 5 || month == 7 ||
112         month == 8 || month == 10 || month == 12)
113         return 31;
114
115     if (month == 4 || month == 6 || month == 9 || month == 11)
116         return 30;
117
118     if (month == 2) return isLeapYear(year) ? 29 : 28;

```



```
119
120     return 0; // If month is incorrect
121 }
122
123 /** Determine if it is a leap year */
isLeapYear 124 public static boolean isLeapYear(int year) {
125     return year % 400 == 0 || (year % 4 == 0 && year % 100 != 0);
126 }
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 print months before 1800.

5.12.4 Benefits of Stepwise Refinement

Stepwise refinement breaks a large problem into smaller manageable subproblems. Each subproblem can be implemented using a method. This approach makes the program easier to write, reuse, debug, test, modify, and maintain.

Simpler Program

The print calendar program is long. Rather than writing a long sequence of statements in one method, stepwise refinement breaks it into smaller methods. This simplifies the program and makes the whole program easier to read and understand.

Reusing Methods

Stepwise refinement promotes code reuse within a program. The isLeapYear method is defined once and invoked from the getTotalNumberOfDays and getNumberOfDayInMonth methods. This reduces redundant code.

Easier Developing, Debugging, and Testing

Since each subproblem is solved in a method, a method can be developed, debugged, and tested individually. This isolates the errors and makes developing, debugging, and testing easier.

When implementing a large program, use the top-down and/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.

incremental development and testing

Better Facilitating Teamwork

When a large problem is divided into subprograms, subproblems can be assigned to different programmers. This makes it easier for programmers to work in teams.

KEY TERMS

actual parameter	179	method overloading	194
ambiguous invocation	195	method signature	179
argument	179	modifier	179
divide and conquer	203	parameter	179
formal parameter (i.e., parameter)	179	pass-by-value	186
information hiding	203	scope of a variable	196
method	178	stepwise refinement	203
method abstraction	203	stub	205

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 a method's parameters. The method name and the parameter list together constitute the *method signature*. Parameters are optional; that is, a method doesn't need to contain any 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. A method can be overloaded. This means that two methods can have the same name, as long as their method parameter lists differ.
10. 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.
11. *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*.
12. 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.
13. When implementing a large program, use the top-down and/or bottom-up coding approach. Do not write the entire program at once. This approach may seem to take more time for coding (because you are repeatedly compiling and running the program), but it actually saves time and makes debugging easier.

TEST QUESTIONS

Do the test questions for this chapter online at www.cs.armstrong.edu/liang/intro9e/test.html.

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. Therefore, the first few numbers are 1, 5, 12, 22, \dots . Write a method with the following header that returns a pentagonal number:

```
public static int getPentagonalNumber(int n)
```

Write a test program that uses this method to display the first 100 pentagonal numbers with 10 numbers on each line.

- *5.2** (*Sum 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 methods with the following headers

```
// Return the reversal of an integer, i.e., reverse(456) returns 654
public static int reverse(int number)
```

```
// Return true if number is a 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** (*Display an integer reversed*) Write a method with the following header 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** (*Sort three numbers*) Write a method with the following header to display three numbers in increasing order:

```
public static void displaySortedNumbers(
    double num1, double num2, double num3)
```

Write a test program that prompts the user to enter three numbers and invokes the method to display them in increasing order.

- *5.6** (*Display patterns*) Write a method to display a pattern as follows:

```

      1
     2 1
    3 2 1
   ...
  n n-1 ... 3 2 1
```



VideoNote

Reverse an integer

The method header is

```
public static void displayPattern(int n)
```

- *5.7** (*Financial application: compute 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 Programming Exercise 2.21.

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 ↵ Enter  
Annual interest rate: 9 ↵ Enter  
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:

```
/** Convert from Celsius to Fahrenheit */  
public static double celsiusToFahrenheit(double celsius)  
  
/** Convert from Fahrenheit to Celsius */  
public static double fahrenheitToCelsius(double fahrenheit)
```

The formula for the conversion is:

```
fahrenheit = (9.0 / 5) * celsius + 32  
celsius = (5.0 / 9) * (fahrenheit - 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:

```
/** Convert from feet to meters */  
public static double footToMeter(double foot)
```

```
/** Convert from meters to feet */
public static double meterToFoot(double meter)
```

The formula for the conversion is:

```
meter = 0.305 * foot
foot = 3.279 * meter
```

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	0.610	25.0	81.967
...			
9.0	2.745	60.0	196.721
10.0	3.050	65.0	213.115

5.10 (Use 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: compute commissions) Write a method that computes the commission, using the scheme in Programming 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 (Display 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**. Characters are separated by exactly one space.

***5.13** (Sum 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** (Estimate π) π can be computed using the following series:

$$m(i) = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \dots + \frac{(-1)^{i+1}}{2i-1} \right)$$

Write a method that returns **m(i)** for a given **i** and write a test program that displays the following table:

i	m(i)
1	4.0000
101	3.1515
201	3.1466
301	3.1449
401	3.1441
501	3.1436
601	3.1433
701	3.1430
801	3.1428
901	3.1427



VideoNote
Estimate π

***5.15** (Financial application: print 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 the following statuses:

Taxable Income	Single	Married Joint or Qualifying Widow(er)	Married Separate	Head of a House
50000	8688	6665	8688	7352
50050	8700	6673	8700	7365
...				
59950	11175	8158	11175	9840
60000	11188	8165	11188	9852

***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 2020.

Sections 5.10–5.11

***5.17** (Display 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 prompts the user to enter **n** and displays an n -by- n matrix. Here is a sample run:

```
Enter n: 3
0 1 0
0 0 0
1 1 1
```



- 5.18** (Use 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** (The `MyTriangle` class) Create a class named `MyTriangle` that contains the following two methods:

```
/** Return true if the sum of any two sides is
 * greater than the third side. */
public static boolean isValid(
    double side1, double side2, double side3)

/** Return 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 Programming Exercise 2.15.

- 5.20** (Use 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** (Geometry: great circle distance) The great circle distance is the distance between two points on the surface of a sphere. Let (x_1, y_1) and (x_2, y_2) be the geographical latitude and longitude of two points. The great circle distance between the two points can be computed using the following formula:

$$d = \text{radius} \times \arccos(\sin(x_1) \times \sin(x_2) + \cos(x_1) \times \cos(x_2) \times \cos(y_1 - y_2))$$

Write a program that prompts the user to enter the latitude and longitude of two points on the earth in degrees and displays its great circle distance. The average earth radius is 6,371.01 km. Note that you need to convert the degrees into radians using the `Math.toRadians` method since the Java trigonometric methods use radians. The latitude and longitude degrees in the formula are for North and West. Use negative to indicate South and East degrees. Here is a sample run:



```
Enter point 1 (latitude and longitude) in degrees:
39.55 -116.25 Enter
Enter point 2 (latitude and longitude) in degrees:
41.5 87.37 Enter
The distance between the two points is 10691.79183231593 km
```

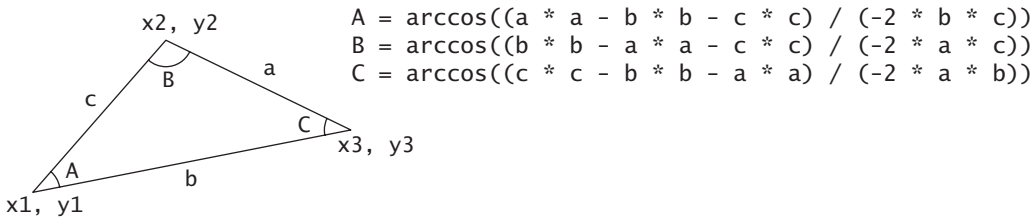
- **5.22** (Math: approximate the square root) There are several techniques for implementing the `sqrt` method in the `Math` class. One such technique is known as the *Babylonian method*. It approximates the square root of a number, `n`, by repeatedly performing a calculation using the following formula:

$$\text{nextGuess} = (\text{lastGuess} + n / \text{lastGuess}) / 2$$

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 `n`. If not, `nextGuess` becomes `lastGuess` and the approximation process continues. Implement the following method that returns the square root of `n`.

```
public static double sqrt(long n)
```

- *5.23** (Geometry: display angles) Write a program that prompts the user to enter three points of a triangle and displays the angles in degrees. Round the value to keep two digits after the decimal point. The formula to compute angles A, B, and C are as follows:



Here is a sample run of the program:

```
Enter three points: 1 1 6.5 1 6.5 2.5
The three angles are 15.26 90.0 74.74
```



Sections 5.10–5.12

- **5.24** (Display current date and time) Listing 2.6, `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 the year, month, and day.
- **5.25** (Convert 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, as are 313 and 757. Write a program that displays the first 100 palindromic prime numbers. Display 10 numbers per line, separated by exactly one space, 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, separated by exactly one space, 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 $2^p - 1$ for some positive integer p . Write a program that finds all Mersenne primes with $p \leq 31$ and displays the output as follows:

```
p      2^p - 1
2      3
3      7
5      31
...
```

- **5.29** (*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 1,000. Display the output as follows:

```
(3, 5)
(5, 7)
...
```

- **5.30** (*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

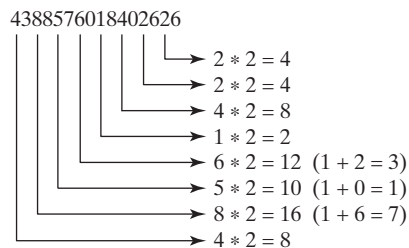


****5.31** (Financial: credit card number validation) Credit card numbers follow certain patterns. A credit card number must have between 13 and 16 digits. It must start with:

- 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. 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. 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)
```

Here are sample runs of the program:



```
Enter a credit card number as a long integer:
4388576018410707 ↵ Enter
4388576018410707 is valid
```



```
Enter a credit card number as a long integer:
4388576018402626 ↵ Enter
4388576018402626 is invalid
```

****5.32** (Game: chance of winning at craps) Revise Exercise 5.30 to run it 10,000 times and display the number of winning games.

****5.33** (Current date and time) Invoking **System.currentTimeMillis()** returns the elapsed 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, 2012 10:34:23
```

****5.34** (*Print calendar*) Programming 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 the area. Here is a sample run:

```
Enter the side: 5.5  Enter
The area of the pentagon is 52.04444136781625
```



***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. Here is a sample run:

```
Enter the number of sides: 5  Enter
Enter the side: 6.5  Enter
The area of the polygon is 72.69017017488385
```



5.37 (*Format an integer*) Write a method with the following header to format the integer with the specified width.

```
public static String format(int number, int width)
```

The method returns a string for the number with one or more prefix 0s. The size of the string is the width. For example, `format(34, 4)` returns `0034` and `format(34, 5)` returns `00034`. If the number is longer than the width, the method returns the string representation for the number. For example, `format(34, 1)` returns `34`.

Write a test program that prompts the user to enter a number and its width and displays a string returned by invoking `format(number, width)`.

***5.38** (*Generate 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.

5.39 (*Geometry: point position*) Programming Exercise 3.32 shows how to test whether a point is on the left side of a directed line, on the right, or on the same line. Write the methods with the following headers:

```
/** Return true if point (x2, y2) is on the left side of the
 * directed line from (x0, y0) to (x1, y1) */
public static boolean leftOfTheLine(double x0, double y0,
    double x1, double y1, double x2, double y2)

/** Return true if point (x2, y2) is on the same
 * line from (x0, y0) to (x1, y1) */
public static boolean onTheSameLine(double x0, double y0,
    double x1, double y1, double x2, double y2)

/** Return true if point (x2, y2) is on the
 * line segment from (x0, y0) to (x1, y1) */
public static boolean onTheLineSegment(double x0, double y0,
    double x1, double y1, double x2, double y2)
```

Write a program that prompts the user to enter the three points for `p0`, `p1`, and `p2` and displays whether `p2` is on the left of the line from `p0` to `p1`, right, the same line, or on the line segment. Here are some sample runs:



Enter three points for p0, p1, and p2: 1 1 2 2 1.5 1.5 ↵ Enter
(1.5, 1.5) is on the line segment from (1.0, 1.0) to (2.0, 2.0)



Enter three points for p0, p1, and p2: 1 1 2 2 3 3 ↵ Enter
(3.0, 3.0) is on the same line from (1.0, 1.0) to (2.0, 2.0)



Enter three points for p0, p1, and p2: 1 1 2 2 1 1.5 ↵ Enter
(1.0, 1.5) is on the left side of the line
from (1.0, 1.0) to (2.0, 2.0)



Enter three points for p0, p1, and p2: 1 1 2 2 1 -1 ↵ Enter
(1.0, -1.0) is on the right side of the line
from (1.0, 1.0) to (2.0, 2.0)