Programming in Java

Online module to accompany *Invitation to Computer Science, 7th Edition* ISBN-10: 1305075773; ISBN-13: 9781305075771 (Cengage Learning, 2016).

1 Introduction to Java

Hundreds of high-level programming languages have been developed; a fraction of these have become viable, commercially successful languages. There are a half-dozen or so languages that can illustrate some of the concepts of a high-level programming language, but this module uses Java for this purpose.

Our intent here is not to make you an expert Java programmer—any more than our purpose in Chapter 4 was to make you an expert circuit designer. Indeed, there is much about the language that we will not even discuss. You will, however, get a sense of what programming in a high-level language is like and perhaps see why some people think it is one of the most fascinating of human endeavors.

1.1 *A Simple Java Program*

Figure 1 shows a simple but complete Java program. Even if you know nothing about the Java language, it is not hard to get the general drift of what the program is doing.

Someone running this program (the user) could have the following dialogue with the program, where boldface indicates what the user types:

```
Enter your speed in mph: 58
Enter your distance in miles: 657.5
At 58 mph, it will take 11.336206896551724 hours
to travel 657.5 miles.
```

To aid our discussion of how the program works, Figure 2 shows the same program with a number in front of each line. The numbers are there for reference purposes only; they are *not* part of the program. Lines 1–3 in the program of Figure 2 are Java **comments**. Anything appearing on a line after the double slash symbol (//) is ignored by the compiler, just as anything following the double dash (--) is treated as a comment in the assembly language programs of Chapter 6. Although the computer ignores comments, they are important to include in a program because they give information to the human readers of the code. Every high-level language has some facility for including comments, because understanding code that someone else has written (or understanding your own code after some period of time has passed) is very difficult without the notes and explanations that comments provide. Comments are one way to *document* a computer program to make it more understandable.

FIGURE 1

A Simple Java Program

```
//Computes and outputs travel time
//for a given speed and distance
//Written by J. Q. Programmer, 6/15/16

import java.util.*;
public class TravelPlanner
{
   public static void main(String[] args)
   {
      int speed;          //rate of travel
      double distance;    //miles to travel
      double time;        //time needed for this travel
      Scanner inp = new Scanner(System.in);    //to read input

      System.out.print("Enter your speed in mph: ");
      speed = inp.nextInt();
      System.out.print("Enter your distance in miles: ");
      distance = inp.nextDouble();

      time = distance/speed;

      System.out.println("At " + speed + " mph, it will take "
         + time + " hours ");
      System.out.println("to travel " + distance + " miles.");
   }
}
```

FIGURE 2

The Program of Figure 1 (line numbers added for reference)

```
1. //Computes and outputs travel time
2. //for a given speed and distance
3. //Written by J. Q. Programmer, 06/15/16
4.
5. import java.util.*;
6. public class TravelPlanner
7. {
8.    public static void main(String[] args)
9.    {
10.       int speed;          //rate of travel
11.       double distance;    //miles to travel
12.       double time;        //time needed for this travel
13.       Scanner inp = new Scanner(System.in);  //to read input
14.
15.       System.out.print("Enter your speed in mph: ");
16.       speed = inp.nextInt();
17.       System.out.print("Enter your distance in miles: ");
18.       distance = inp.nextDouble();
19.
20.       time = distance/speed;
21.
22.       System.out.println("At " + speed + " mph, it will take "
23.          + time + " hours ");
24.       System.out.println("to travel " + distance + " miles.");
25.    }
26. }
```

Java Is Born

Java was developed at Sun Microsystems, Inc., but its birth as a full-fledged programming language was almost an accident. In early 1991, Sun created a team of top-notch software developers and gave them free rein to do whatever creative thing they wanted. The somewhat secret "Green team" isolated itself and set to work mapping out a strategy. Its focus was on the consumer electronics market. Televisions, VCRs, stereo systems, laser disc players, and video game machines all operated on different CPUs. Over the next 18 months the team worked to develop the graphical user interface (GUI), a programming language, an operating system, and a hardware architecture for a handheld remote-control device called the *7 that would allow various electronic devices to communicate over a network. In contrast to the high-end workstations that were a Sun hallmark, the *7 was designed to be small, inexpensive, easy to use, reliable, and equipped with software that could function over the multiple hardware platforms of the consumer electronics market.

Armed with this technology, Sun went looking for a business market but found none. In 1993, Mosaic—the first graphical Internet browser—was created at the National Center for Supercomputing Applications, and the World Wide Web began to emerge. This development sent the Sun group in a new direction where their experience with platform independence, reliability, security, and GUIs paid off: They wrote a Web browser.

The programming language component of the *7 was named Oak, for a tree outside language developer James Gosling's window. Later renamed Java, the language was used to code the Web browser. The Web browser was released in 1995, and the first version of the Java programming language itself was released in 1996. Java gained market share among programming languages at quite a phenomenal rate. Sun released several versions of Java, each succeeding version with increased capabilities and features. Oracle Corporation bought out Sun Microsystems in 2010, but continues to support Java language development.

The comments in lines 1–3 of Figure 2 describe what the program does plus tell who wrote the program and when. These three comment lines together make up the program's **prologue comment** (the introductory comment that comes first). A prologue comment is always a good idea; it's almost like the headline in a newspaper, giving the big picture up front.

Blank lines in Java programs are ignored and are used, like comments, to make the program more readable by human beings. In our example program, we've used blank lines (lines 4, 14, 19, 21) to separate sections of the program, visually indicating groups of statements that are related.

Line 5 is an **import** statement asking the linker to include object code from a Java library or **package**. Line 6 is a **class header**, which announces that a **class** is about to be defined. The class is named *TravelPlanner*, and the curly braces at lines 7 and 26 mark the beginning and end of this class definition. All Java code (except for comments and import statements) must be either a class header or inside a class definition. The word "public" in line 6 denotes that the *TravelPlanner* class is available to any other program that might want to make use of it.

We will have much more to say about classes later. For now, just think of a class as a collection of sections of code called **methods** that are able to perform various related services. In the *TravelPlanner* class, there is only one method, the **main method**. The service it performs is to compute and write out the time to travel a given distance at a given speed. Line 8,

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

is the header for the main method. It is not necessary to understand this somewhat obscure code; just remember that every Java program must have a main method and that all main methods start out exactly this way. The curly braces at lines 9 and 25 enclose the main **method body**, which is the heart of the sample program. Lines 10–12 are declarations that name and describe the items of data that are used within the main method. Descriptive names—*speed*, *distance*, and *time*—are used for these quantities to help document their purpose in the program, and comments provide further clarification. Line 10 describes an integer quantity (type "int") called *speed*. Lines 11 and 12 declare *distance* and *time* as real number quantities (type "double"). A real number quantity is one containing a decimal point, such as 28.3, 102.0, or –17.5. Line 13 declares *inp* as an object of the *Scanner* class that will be used to collect input from the user; we'll explain this in more detail later.

Lines 15–18 prompt the user to enter values for speed and distance and store those values in *speed* and *distance*. Line 20 computes the time required to travel this distance at this speed. Finally, lines 22–24 print the two lines of output to the user's screen. The values of *speed*, *time*, and *distance* are inserted in appropriate places among the strings of text shown in double quotes.

You may have noticed that most of the statements in this program end with a semicolon. A semicolon must appear at the end of every executable Java instruction, which means everywhere except at the end of a comment or at the end of a class header such as

```
public class TravelPlanner
```

or a method header such as

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

Java, along with every other programming language, has specific rules of **syntax**—the correct form for each component of the language. Having a semicolon at the end of every executable statement is a Java syntax rule. Any violation of the syntax rules generates an error message from the compiler, because the compiler does not recognize or know how to translate the offending code. In the case of a missing semicolon, the compiler cannot tell where the instruction ends. The syntax rules for a programming language are often defined by a formal grammar, much as correct English syntax is defined by rules of grammar.

Java is a **free-format language,** which means that it does not matter where things are placed on a line. For example, we could have written

```
         time   =
    distance        /
                      speed;
```

although this is clearly harder to read. The free-format characteristic explains why a semicolon is needed to mark the end of an instruction, which might be spread over several lines.

1.2 *Creating and Running a Java Program*

Creating and running a Java program is basically a three-step process. The first step is to type the program into a text editor. When you are finished, you save the file using the same name as the class, with the file extension *.java*. So the file for Figure 1 is named

```
TravelPlanner.java
```

As the second step, the program in the *.java* file must be compiled using a Java compiler for your computer; for our example program, the result is a file called

```
TravelPlanner.class
```

that contains low-level code called **bytecode**, which is not yet object code. The third step operates on the *.class* file; it finishes the translation to object code and links, loads, and executes your program. Depending on your system, you may have to type operating system commands for the last two steps.

Another approach is to do all of your work in an **Integrated Development Environment**, or **IDE**. The IDE lets the programmer perform a number of tasks within the shell of a single application program, rather than having to use a separate program for each task. A modern programming IDE provides a text editor, a file manager, a compiler, a linker and loader, and tools for debugging, all within this one piece of software. The IDE usually has a GUI (graphical user interface) with menu choices for the different tasks. This can significantly speed up program development.

This Java exercise is just a beginning. In the rest of this chapter, we'll examine the features of the language that will enable you to write your own Java programs to carry out more sophisticated tasks.

JAVA COMPILERS

You can download a free Java command-line compiler from

http://www.oracle.com/technetwork/java/index.html

Look for the Java SE Development Kit (JDK). There are versions for Linux and Windows. Apple Java came pre-installed on Mac OS X systems through OS X 10.6, but not on later versions of the operating system.

There are also a number of Java GUI IDEs available, such as the free Dr. Java compiler that runs on top of the JDK and can be downloaded from

www.cs.rice.edu/~javaplt/drjava/

2 Virtual Data Storage

One of the improvements we seek in a high-level language is freedom from having to manage data movement within memory. Assembly language does not require us to give the actual memory address of the storage location to be used for each item, as in machine language. However, we still have to move values, one by one, back and forth between memory and the arithmetic logic unit (ALU) as simple modifications are made, such as setting the value of A to the sum of the values of B and C. We want the computer to let us use data values by name in any appropriate computation without thinking about where they are stored or what is currently in some register in the ALU. In fact, we do not even want to know that there *is* such a thing as an ALU, where data are moved to be operated on. Instead, we want the virtual machine to manage the details when we request that a computation be performed. A high-level language allows this, and it also allows the names for data items to be more meaningful than in assembly language.

Names in a programming language are called **identifiers**. Each language has its own specific rules for what a legal identifier can look like. In Java, an identifier can be any combination of letters, digits, and the underscore symbol (_), as long as it does not begin with a digit. An additional restriction is that an identifier cannot be one of the few **keywords**, such as “class,” “public,” “int,” and so forth, that have a special meaning in Java and that you would not be likely to use anyway. The three integers *B*, *C*, and *A* in our assembly language program can therefore have more descriptive names, such as *subTotal*, *tax*, and *finalTotal*. The use of descriptive identifiers is one of the greatest aids to human understanding of a program. Identifiers can be almost arbitrarily long, so be sure to use a meaningful identifier such as *finalTotal* instead of something like *A*; the improved readability is well worth the extra typing time. Java is a **case-sensitive** language, which means that uppercase letters are distinguished from lowercase letters. Thus, *FinalTotal*, *Finaltotal*, and *finalTotal* are three different identifiers.

CAPITALIZATION OF IDENTIFIERS

There are two standard capitalization patterns for identifiers, particularly “multiple word” identifiers:

camel case: First word begins with a lowercase letter, additional words begin with uppercase letters (*finalTotal*)

Pascal case: All words begin with an uppercase letter (*FinalTotal*)

The code in this chapter uses the following convention for creating identifiers (examples included):

Simple variables – camel case: *speed*, *time*, *finalTotal*

Named constants - all uppercase: *PI, FREEZING_POINT*

Method names – camel case: *myMethod*, *getInput*

Class names – Pascal case: *MyClass*

Object names – camel case: *myObject*

The underscore character is not used except for named constants. Occasionally, however, we’ll use single capital letters for identifiers in quick code fragments.

Data that a program uses can come in two varieties. Some quantities are fixed throughout the duration of the program, and their values are known ahead of time. These quantities are called **constants**. An example of a constant is the integer value 2. Another is an approximation to π, say 3.1416. The integer 2 is a constant that we don’t have to name by an identifier, nor do we have to build the value 2 in memory manually by the equivalent of a .DATA pseudo-op. We can just use the symbol “2” in any program statement. When “2” is first encountered in a program statement, the binary representation of the integer 2 is automatically generated and stored in a memory location. Likewise, we can use "3.1416" for the real number value 3.1416, but if we are really using this number as an approximation to π, it is more informative to use the identifier *PI*.

Some quantities used in a program have values that change as the program executes, or values that are not known ahead of time but must be obtained from the computer user (or from a data file previously prepared by the user) as the program runs. These quantities are called **variables**. For example, in a program doing computations with circles (where we might use the constant *PI*), we might need to obtain from the user or a data file the radius of the circle. This variable can be given the identifier *radius*.

Identifiers for variables serve the same purpose in program statements as pronouns do in ordinary English statements. The English statement "He will be home today" has specific meaning only when we plug in the value for which "He" stands. Similarly, a program statement such as

```
time = distance/speed;
```

becomes an actual computation only when numeric values have been stored in the memory locations referenced by the *distance* and *speed* identifiers.

We know that all data are represented internally in binary form. In Chapter 4 we noted that any one sequence of binary digits can be interpreted as a whole number, a negative number, a real number (one containing a decimal point, such as –17.5 or 28.342), or as a letter of the alphabet. Java requires the following information about each variable in the program:

- What identifier we want to use for it (its name)
- What **data type** it represents (e.g., an integer or a letter of the alphabet)

The data type determines how many bytes will be needed to store the variable—that is, how many memory cells are to be considered as one **memory location** referenced by one identifier—and also how the string of bits in that memory location is to be interpreted. Java provides several "primitive" data types that represent a single unit of information, as shown in Figure 3.

The way to give this information within a Java program is to declare each variable. A **variable declaration** consists of a data type followed by a list of one or more identifiers of that type. Our sample program used three declaration statements:

```
int speed;            //rate of travel
double distance;      //miles to travel
double time;          //time needed for this travel
```

but these could have been combined into two:

```
int speed;                //rate of travel
double distance, time;    //miles to travel and time
                          //needed for this travel
```

Where do the variable declarations go? Although the only requirement is that a variable must be declared before it can be used, all variable declarations are

FIGURE 3

Some of the Java Primitive Data Types

int	an integer quantity
double	a real number
char	a character (a single keyboard character, such as 'a')

usually collected together at the top of the main method, as in our sample program. This also gives the reader of the code quick information about the data that the program will be using.

What about the constant *PI*? We want to assign the fixed value 3.1416 to the *PI* identifier. Constant declarations are just like variable declarations, with the addition of the keyword **final** and the assignment of the fixed value to the constant identifier.

```
final double PI = 3.1416;
```

Many programmers use all uppercase letters to denote constant identifiers, but the compiler identifies a constant quantity only by the presence of **final** in the declaration. Once a quantity has been declared as a constant, any attempt later in the program to change its value generates an error message from the compiler.

In addition to variables of a primitive data type that hold only one unit of information, it is possible to declare a whole collection of related variables at one time. This allows storage to be set aside as needed to contain each of the values in this collection. For example, suppose we want to record the number of hits on a Web site for each month of the year. The value for each month is a single integer. We want a collection of 12 such integers, ordered in a particular way. An **array** groups together a collection of memory locations, all storing data of the same type. The following statement declares an array:

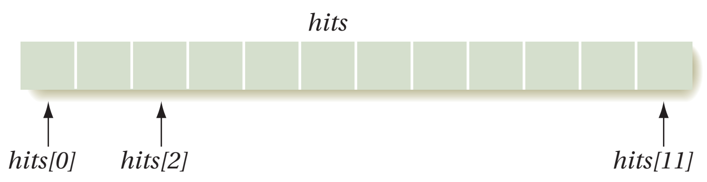
```
int[] hits = new int[12];
```

The left side of the equals sign says that *hits* is an array of integers; the right side of the equals sign actually generates (new) memory locations for 12 integer quantities. The 12 individual array elements are numbered from *hits[0]* to *hits[11]*. (Notice that a Java array counts from 0 up to 11, instead of from 1 up to 12.) Thus, we use *hits[0]* to refer to the first entry in *hits*, which represents the number of visits to the Web site during the first month of the year, January. Continuing this numbering scheme, *hits[2]* refers to the number of visits during March, and *hits[11]* to the number of visits during December. In this way we use one declaration to cause 12 separate (but related) integer storage locations to be set up. Figure 4 illustrates this array.

Here is an example of the power of a high-level language. In assembly language, we can name only individual memory locations—that is, individual items of data—but in Java we can also assign a name to an entire collection of related data items. An array thus allows us to talk about an entire table of values, or the individual elements making up that table. If we are writing Java programs to implement the data cleanup algorithms of Chapter 3, we can use an array of integers to store the 10 data items.

FIGURE 4

A 12-Element Array hits



PRACTICE PROBLEMS

1. Which of the following are legitimate Java identifiers?

   `martinBradley    C3P_OH    Amy3    3Right    double`

2. Write a declaration for a Java program that uses one integer quantity called *number*.
3. Write a Java statement that declares a type *double* constant called *TAX_RATE* that has the value 5.5.
4. Using the *hits* array of Figure 4, how do you reference the number of hits on the Web page for August?

3 Statement Types

Now that we can reserve memory for data items simply by naming what we want to store and describing its data type, we will examine additional kinds of programming instructions (statements) that Java provides. These statements enable us to manipulate the data items and do something useful with them. The instructions in Java, or indeed in any high-level language, are designed as components for algorithmic problem solving, rather than as one-to-one translations of the underlying machine language instruction set of the computer. Thus, they allow the programmer to work at a higher level of abstraction. In this section we examine three types of high-level programming language statements. They are consistent with the pseudocode operations described in Chapter 2 (see Figure 2.9).

Input/output statements make up one type of statement. An **input statement** collects a specific value from the user for a variable within the program. In our TravelPlanner program, we need input statements to get the values of the speed and distance that are to be used in the computation. An **output statement** writes a message or the value of a program variable to the user's screen. Once the TravelPlanner program computes the time required to travel the given distance at the given speed, the output statement displays that value on the screen.

Another type of statement is the **assignment statement**, which assigns a value to a program variable. This is similar to what an input statement does, except that the value is not collected directly from the user, but is computed by the program. In pseudocode we called this a "computation operation."

Control statements, the third type of statement, affect the order in which instructions are executed. A program executes one instruction or program statement at a time. Without directions to the contrary, instructions are executed sequentially, from first to last in the program. (In Chapter 2 we called this a straight-line algorithm.) Imagine beside each program statement a light bulb that lights up while that statement is being executed; you would see a ripple of lights from the top to the bottom of the program. Sometimes, however, we want to interrupt this sequential progression and jump around in the program (which is accomplished by the instructions JUMP, JUMPGT, and so on, in assembly language). The progression of lights, which may no longer be sequential, illustrates the **flow of control** in the program—that is, the path through the program that is traced by following the currently executing statement. Control statements direct this flow of control.

3.1 *Input/Output Statements*

Remember that the job of an input statement is to collect from the user specific values for variables in the program. In pseudocode, to get the value for *speed* in the TravelPlanner program, we would say something like

```
Get value for $speed$
```

The core Java language does not provide a convenient way to collect user data entered at the keyboard. However, a Java class, called the *Scanner* class, has been written that provides an easy way to do this. Although we still have no definition of what a "class" is, other than that it contains sections of code called methods, the object code for useful classes is stored in code libraries. In order to access the *Scanner* class code, we need to use an import statement. The *Scanner* class is found in the Java "utility" library, so our TravelPlanner program begins with

```
import java.util.*;
```

The * is a "wild card" designation, so the above statement asks the linker to include object code for all of the classes in the utility library, which includes the *Scanner* class. The utility library also includes other classes that we don't need for the TravelPlanner program, but the extra object code does no harm and keeps our import statement simple and easy to remember. The object code gets linked into the object code for our program, so we use the *Scanner* class code without ever seeing it. All we need to know is the services it provides and how to use those services properly. This is in the same spirit of abstraction that led to the development of high-level languages in the first place.

The *Scanner* class actually represents a new data type (not one of the primitive data types such as *int* or *double*). Before we can make use of the *Scanner* class methods, we must declare a variable (an object) of the *Scanner* data type. The following statement in the TravelPlanner program

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

declares *inp* as an object of the *Scanner* class. There is nothing special about the identifier *inp*; it suggests "input," but any legal Java identifier could be used here. *System.in* indicates that the source of the input will be the keyboard. The *inp* object now has access to these useful Scanner methods

`nextInt()`	read a value of type int
`nextDouble()`	read a value of type double

Finally, in the statement

```
speed = inp.nextInt();
```

the *inp* object uses the *nextInt()* method to read input from the keyboard and store it in the previously declared *int* variable *speed*. (Recall that all variables must be declared before they can be used.)

Similarly, the statement to read in the value for distance that the user enters at the keyboard is

```
distance = inp.nextDouble();
```

You cannot use

```
SomeVariable = inp.nextDouble();
```

if *SomeVariable* has been declared as type *int*. The Java compiler will give you an error message about "incompatible types."

The value of the time can be computed and stored in the memory location referenced by *time*. A pseudocode operation for producing output would be something like

```
Print the value of time
```

This could be done in Java by the following statement:

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

System.out is a predefined object of a class with a *println* method that writes output to the screen; in the above statement, the object is using that *println* method. But we don't want the program to simply print a number with no explanation; we want some words to make the output meaningful.

The general form of the output statement is

```
System.out.println(string);   or   System.out.print(string);
```

The difference between *System.out.println* and *System.out.print* is that after the *println* statement, the screen cursor moves to the next line where any subsequent output will appear, whereas after a *print* statement, the cursor remains on the same line. The string in the output statement could be empty, as follows:

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

This just prints a blank line, which is useful for formatting the output to make it easier to read. The string can also be a **literal string** (enclosed in double quotes). Literal strings are printed out exactly as is. For example,

```
System.out.println("Here's your answer.");
```

prints

```
Here’s your answer.
```

A string can also be composed of items joined by +, the **concatenation operator**. The items can be literal strings, numbers, or variables. Items that are not themselves literal strings are converted to strings for the purposes of writing them out. For example,

```
System.out.println("Give me" + 5);
```

prints the line

```
Give me5
```

on the screen. If we want a space between “me” and “5”, then we make that space part of the literal string, as in

```
System.out.println("Give me " + 5);
```

If *number* is an integer variable with current value 5, then the same output is produced by

```
System.out.println("Give me " + number);
```

The concatenation operator is also helpful when trying to write out a long literal string; whereas a single Java statement can be spread over multiple lines, a line break cannot occur in the middle of a literal string. The solution is to make two smaller substrings and concatenate them, as in

```
System.out.println("Oh for a sturdy ship to sail, "
   + "and a star to steer her by.");
```

Literal strings and variables can be concatenated together in all sorts of combinations, as long as the quotation marks and + signs appear in the right places. Consider again the output statements in the TravelPlanner program:

```
System.out.println("At " + speed + " mph, it will take "
    + time + " hours ");
System.out.println("to travel " + distance + " miles.");
```

Let’s back up a bit and note that we also need to print some text information before the input statement, to alert the user that the program expects some input. A statement such as

```
System.out.print("Enter your speed in mph: ");
```

acts as a user **prompt**. Without a prompt, the user may be unaware that the program is waiting for some input; instead, it may simply seem to the user that the program is “hung up.”

Assembling all of these bits and pieces, we can see that

```
   System.out.print("Enter your speed in mph: ");
   speed = inp.nextInt();
   System.out.print("Enter your distance in miles: ");
   distance = inp.nextDouble();
```

is a series of prompt, input, prompt, input statements to get the data, and then

```
System.out.println("At " + speed + " mph, it will take "
   + time + " hours ");
System.out.println("to travel " + distance + " miles.");
```

writes out the computed value of *time* along with the associated input values in an informative message. In the middle, we need a program statement to compute the value of *time*. We can do this with a single assignment statement; the assignment statement is explained in the next section.

In our sample execution of the TravelPlanner program, we got the following output:

```
   At 58 mph, it will take 11.336206896551724 hours
   to travel 657.5 miles.
```

This is fairly ridiculous output—it does not make sense to display the result to 15 decimal digits. Exercise 11 at the end of this module tells you how decimal output can be formatted to a specified number of decimal places.

PRACTICE PROBLEMS

1. Write two statements that prompt the user to enter an integer value and store that value in a (previously declared) variable called *quantity*.
2. A program has computed a value for the variable *average* that represents the average high temperature in San Diego for the month of May. Write an appropriate output statement.
3. What appears on the screen after execution of the following statement?

```
System.out.println("This is" + "goodbye"
  + ", Steve");
```

3.2 *The Assignment Statement*

As we said earlier, an assignment statement assigns a value to a program variable. This is accomplished by evaluating some expression and then writing the resulting value in the memory location referenced by the program variable. The general pseudocode operation

```
Set the value of "variable" to "arithmetic expression"
```

has as its Java equivalent

```
variable = expression;
```

The expression on the right is evaluated, and the result is then written into the memory location named on the left. For example, suppose that *A*, *B*, and *C* have all been declared as integer variables in some program. The assignment statements

```
B = 2;
C = 5;
```

result in *B* taking on the value 2 and *C* taking on the value 5. After execution of

```
A = B + C;
```

A has the value that is the sum of the current values of *B* and *C*. Assignment is a destructive operation, so whatever *A*'s previous value was, it is gone. Note that this one assignment statement says to add the values of *B* and *C* and assign the result to *A*. This one high-level language statement is equivalent to three assembly language statements needed to do this same task (LOAD B, ADD C, STORE A). A high-level language program thus packs more power per line than an assembly language program. To state it another way, whereas a single assembly language instruction is equivalent to a single machine language instruction, a single Java instruction is usually equivalent to many assembly language instructions or machine language instructions, and it allows us to think at a higher level of problem solving.

In the assignment statement, the expression on the right is evaluated first. Only then is the value of the variable on the left changed. This means that an assignment statement like

```
A = A + 1;
```

makes sense. If *A* has the value 7 before this statement is executed, then the expression evaluates to

```
7 + 1 or 8
```

and 8 then becomes the new value of *A*. (Here it becomes obvious that the assignment instruction symbol $=$ is not the same as the mathematical equals sign $=$, because $A = A + 1$ does not make sense mathematically.)

All four basic arithmetic operations can be done in Java, denoted by

+	Addition
-	Subtraction
*	Multiplication
/	Division

For the most part, this is standard mathematical notation rather than the somewhat verbose assembly language op code mnemonics such as SUBTRACT. The reason a special symbol is used for multiplication is that × would be confused with *x*, an identifier, · (a multiplication dot) doesn't appear on the keyboard, and juxtaposition (writing *AB* for *A***B*) would look like a single identifier named *AB*.

We do have to pay some attention to data types. In particular, division has one peculiarity. If at least one of the two values being divided is a real number, then division behaves as we expect. Thus,

```
7.0/2  7/2.0  7.0/2.0
```

all result in the value 3.5. However, if the two values being divided are both integers, the result is an integer value; if the division doesn't "come out even," the integer value is obtained by truncating the answer to an integer quotient. Thus,

```
7/2
```

results in the value 3. Think of grade-school long division of integers:

$$\begin{array}{r} 3 \\ 2\overline{)7} \\ \underline{6} \\ 1 \end{array}$$

Here the quotient is 3 and the remainder is 1. Java also provides an operation, with the symbol %, to obtain the integer remainder. Using this operation,

```
7 % 2
```

results in the value 1.

If the values are stored in type *int* variables, the result is the same. For example,

```
int numerator;
int denominator;
int quotient;
numerator = 7;
denominator = 2;
quotient = numerator/denominator;
System.out.println("The result of " + numerator + "/"
   + denominator + " is " + quotient);
```

produces the output

```
The result of 7/2 is 3
```

As soon as an arithmetic operation involves one or more real (decimal) numbers, any integers are converted to their real number equivalent, and the calculations are done with real numbers.

Data types also play a role in assignment statements. Suppose the expression in an assignment statement evaluates to a real number, and your program tries to assign it to an identifier that has been declared as an integer. The Java compiler gives you an error message stating that you have incompatible types. (We mentioned that this same problem occurs if you try to use the *nextDouble()* input method from the *Scanner* class and assign the result to an integer variable.) In fact, the error message goes on to say that you need to do an "explicit cast" to convert *double* to *int*. Java is saying that if you want to throw away the noninteger part of a decimal number by storing it in an integer, you're going to have to write code to do that. However, you can assign an integer value to a type *double* variable (or input an integer value to a type *double* variable). Java does this **type casting** (changing of data type) automatically. This type cast would merely change the integer 3, for example, to its real number equivalent 3.0.

This explains why we declared *distance* to be type *double* in the TravelPlanner program. The user can enter an integer value for distance, and Java will type cast it to a real number. But if we had declared both *speed* and *distance* to be integers, then the division to compute *time* would only produce integer answers.

You should assign only an expression that has a character value to a variable that has been declared to be type *char*. Suppose that *Letter* is a variable of type *char*. Then

```
Letter = 'm';
```

is a legitimate assignment statement, giving *Letter* the value of the character 'm'. Note that single quotation marks are used here, as opposed to the double quotation marks that enclose a literal string. The assignment

```
Letter = '4';
```

is also acceptable; the single quotes around the 4 mean that it is being treated as just another character on the keyboard, not as the integer 4.

Java requires that all variables have a value before they are used. It is a good idea to get into the habit of **initializing variables** as soon as they are declared, using an assignment statement. For example, you can declare and then initialize a variable by

```
int count;
count = 0;
```

but Java also allows you to combine these two statements into one:

```
int count = 0;
```

This statement is equivalent to the assembly language statement

```
COUNT: .DATA 0
```

that reserves a memory location, assigns it the identifier COUNT, and fills it with the value zero.

PRACTICE PROBLEMS

1. *newNumber* and *next* are integer variables in a Java program. Write a statement to assign the value of *newNumber* to *next*.
2. What is the value of *Average* after the following statements are executed? (*Note*: *total* and *number* are type *int*, and *average* is type *double*.)

```
total = 277;
number = 5;
average = total/number;
```

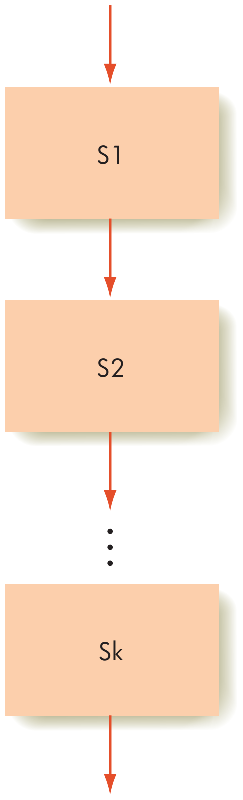
3.3 *Control Statements*

We mentioned earlier that sequential flow of control is the default; that is, a program executes instructions sequentially from first to last. The flowchart in Figure 5 illustrates this, where S1, S2, . . ., Sk are program instructions (program statements).

As stated in Chapter 2, no matter how complicated the task to be done, only three types of control mechanisms are needed:

1. **Sequential**: Instructions are executed in order.

FIGURE 5
Sequential Flow of Control



2. **Conditional**: Which instruction executes next depends on some condition.
3. **Looping**: A group of instructions may be executed many times.

Sequential flow of control, the default, is what occurs if the program does not contain any instances of the other two control structures. In the TravelPlanner program, for instance, instructions are executed sequentially, beginning with the input statements, next the computation, and finally the output statement.

In Chapter 2 we introduced pseudocode notation for conditional operations and looping. In Chapter 6 we learned how to write somewhat laborious assembly language code to implement conditional operations and looping. Now we'll see how Java provides instructions that directly carry out these control structure mechanisms—more evidence of the power of high-level language instructions. We can think in a pseudocode algorithm design mode, as we did in Chapter 2, and then translate that pseudocode directly into Java code.

Conditional flow of control begins with the evaluation of a **Boolean condition**, also called a **Boolean expression**, which can be either true or false. We discussed these "true/false conditions" in Chapter 2, and we also encountered Boolean expressions in Chapter 4, where they were used to design circuits. A Boolean condition often involves comparing the values of two expressions and determining whether they are equal, whether the first is greater than the second, and so on. Again assuming that *A*, *B*, and *C* are integer variables in a program, the following are legitimate Boolean conditions:

`A == 0`	(Does *A* currently have the value 0?)
`B < (A + C)`	(Is the current value of *B* less than the sum of the current values of *A* and *C*?)
`A != B`	(Does *A* currently have a different value than *B*?)

If the current values of *A*, *B*, and *C* are 2, 5, and 7, respectively, then the first condition is false (*A* does not have the value zero), the second condition is true (5 is less than 2 plus 7), and the third condition is true (*A* and *B* do not have equal values).

Comparisons need not be numeric. They can also be made between variables of type *char*, where the "ordering" is the usual alphabetic ordering. If *initial* is a value of type *char* with a current value of 'D', then

```
initial == 'F'
```

is false because *initial* does not have the value 'F', and

```
initial < 'P'
```

is true because 'D' precedes 'P' in the alphabet (or, more precisely, because the binary code for 'D' is numerically less than the binary code for 'P'). Note that the comparisons are case sensitive, so 'F' is not equal to 'f', but 'F' is less than 'f'.

Figure 6 shows the comparison operators available in Java. Note the use of the two equality signs to test whether two expressions have the same value. The single equality sign is used in an assignment statement, the double equality sign in a comparison.

Boolean conditions can be built up using the Boolean operators AND, OR, and NOT. Truth tables for these operators were given in Chapter 4

FIGURE 6

Java Comparison Operators

COMPARISON	SYMBOL	EXAMPLE	EXAMPLE RESULT
the same value as	==	2 == 5	false
less than	<	2 < 5	true
less than or equal to	<=	5 <= 5	true
greater than	>	2 > 5	false
greater than or equal to	>=	2 >= 5	false
not the same value as	!=	2 != 5	true

(Figures 4.12–4.14). The only new thing is the symbols Java uses for these operators, shown in Figure 7.

A conditional statement relies on the value of a Boolean condition (true or false) to decide which programming statement to execute next. If the condition is true, one statement is executed next, but if the condition is false, a different statement is executed next. Control is therefore no longer in a straight-line (sequential) flow, but may hop to one place or to another. Figure 8 illustrates this situation. If the condition is true, the statement S1 is executed (and statement S2 is not); if the condition is false, the statement S2 is executed (and statement S1 is not). In either case, the flow of control then continues on to statement S3. We saw this same scenario when we discussed pseudocode conditional statements in Chapter 2 (Figure 2.4).

The Java instruction that carries out conditional flow of control is called an **if-else** statement. It has the following form (note that the words *if* and *else* are lowercase, and that the Boolean condition must be in parentheses):

```
if (Boolean condition)
        S1;
else
        S2;
```

Below is a simple if-else statement, where we assume that *A*, *B*, and *C* are integer variables:

```
if (B < (A + C))
     A = 2*A;
else
     A = 3*A;
```

Suppose that when this statement is reached, the values of *A*, *B*, and *C* are 2, 5, and 7, respectively. As we noted before, the condition $B < (A + C)$ is then true, so the statement

```
A = 2*A;
```

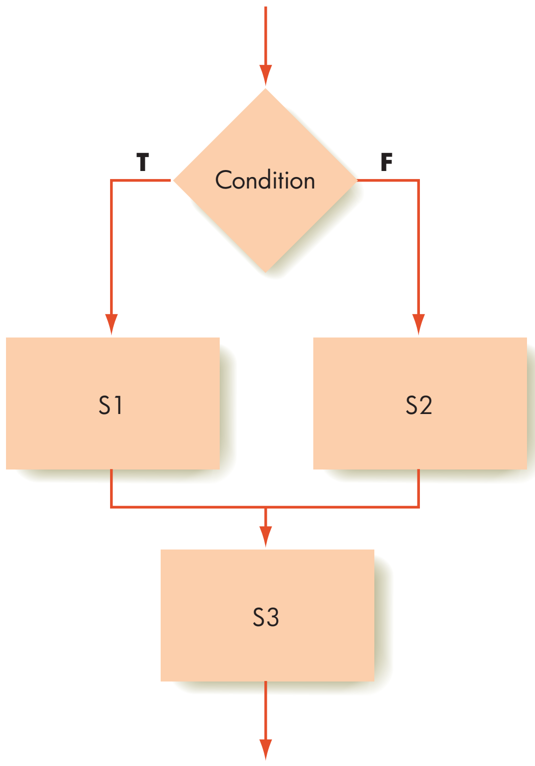
FIGURE 7

Java Boolean Operators

OPERATOR	SYMBOL	EXAMPLE	EXAMPLE RESULT
AND	&&	(2 < 5) && (2 > 7)	false
OR	||	(2 < 5) || (2 > 7)	true
NOT	!	!(2 == 5)	true

FIGURE 8

Conditional Flow of Control (if-else)



is executed, and the value of *A* is changed to 4. However, suppose that when this statement is reached, the values of *A*, *B*, and *C* are 2, 10, and 7, respectively. Then the condition $B < (A + C)$ is false, the statement

```
    A = 3*A;
```

is executed, and the value of *A* is changed to 6.

A variation on the if-else statement is to allow an "empty else" case. Here we want to do something if the condition is true, but if the condition is false, we want to do nothing. Figure 9 illustrates the empty else case. If the condition is true, statement S1 is executed, and after that the flow of control continues on to statement S3, but if the condition is false, nothing happens except to move the flow of control directly on to statement S3.

This *if* variation of the if-else statement can be accomplished by omitting the word *else*. This form of the instruction therefore looks like

```
    if (Boolean condition)
              S1;
```

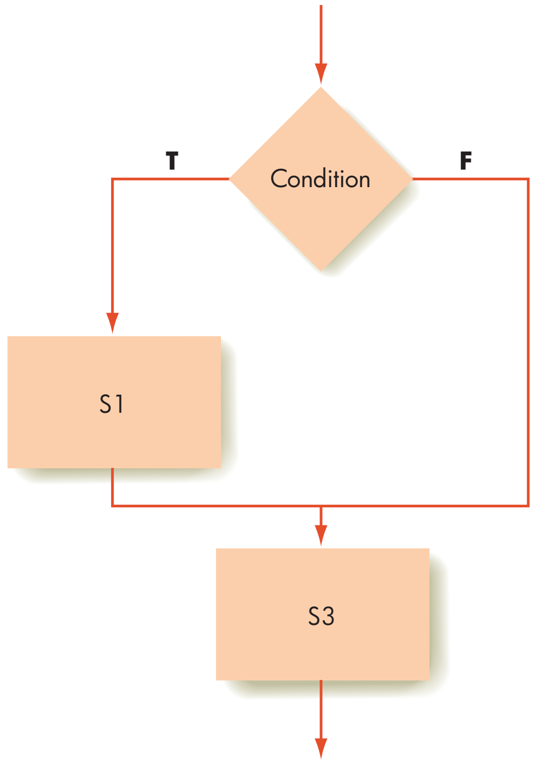
We could write

```
    if (B < (A + C))
         A = 2*A;
```

This has the effect of doubling the value of *A* if the condition is true and of doing nothing if the condition is false.

FIGURE 9

if-else with Empty else



It is possible to combine statements into a group by putting them within the curly braces { and }. The group is then treated as a single statement, called a **compound statement**. A compound statement can be used anywhere a single statement is allowed. For example,

```
{
 System.out.println( "This is the first statement.");
 System.out.println( "This is the second statement.");
 System.out.println( "This is the third statement.");
}
```

is treated as a single statement. The implication is that in Figure 8, S1 or S2 might be compound statements. This makes the if-else statement potentially much more powerful, and similar to the pseudocode conditional statement in Figure 2.9.

Let's expand on our TravelPlanner program and give the user of the program a choice of computing the time either as a decimal number (3.75 hours) or as hours and minutes (3 hours, 45 minutes). This situation is ideal for a conditional statement. Depending on what the user wants to do, the program does one of two tasks. For either task, the program still needs information about the speed and distance. The program must also collect information to indicate which task the user wishes to perform. We need an additional variable in the program to store this information. Let's use a variable called *choice* of type *char* to collect the user's choice of which task to perform. We also need two new integer variables to store the values of hours and minutes.

There is one little glitch in collecting input of type *char*. While the *Scanner* class has nice input methods for reading type *integer* values (*nextInt( )*) and type *double* values (*nextDouble ( )*), there is, unfortunately, no "nextChar()" method. There is, however, a *next( )* method that reads in a string of characters. What we will do is declare a variable of type *String* (not a primitive data type of Java, but available for use in any Java program), read in a string, and then use the String method *charAt*(index) to peel off the first character of the string, which will be at index 0. Strings, like arrays, are counted from location 0, not location 1.

Figure 10 shows the new program. Note that all variables are now initialized as part of the declaration. The condition evaluated at the beginning of the if-else statement tests whether *choice* has the value 'D'. If so, then the condition is true, and the first group of statements is executed—that is, the time is output in decimal format as we have been doing all along. If *choice* does not have the value 'D', then the condition is false. In this event, the second group of statements is executed. Note that because of the way the condition is written, if *choice* does not have the value 'D', it is assumed

FIGURE 10

The TravelPlanner Program with a Conditional Statement

```
// Computes and outputs travel time
// for a given speed and distance
// Written by J. Q. Programmer, 6/28/16

import java.util.*;
public class TravelPlanner
{
   public static void main(String[] args)
   {
      int speed = 1;           //rate of travel
      double distance = 0.0;  //miles to travel
      double time = 0.0;      //time needed for travel (decimal)
      int hours = 0;          //time for travel in hours
      int minutes = 0;        //leftover time in minutes
      String response = " "; //user's response
      char choice = 'M';      //choice of output as decimal
                              //hours
                              //or hours and minutes
      Scanner inp = new Scanner(System.in);

      System.out.print("Enter your speed in mph: ");
      speed = inp.nextInt();
      System.out.print("Enter your distance in miles: ");
      distance = inp.nextDouble();
      System.out.println("Enter your choice of format "
         + "for time, ");
      System.out.print("decimal hours (D) "
         + "or hours and minutes (M): ");
      response = inp.next();
      choice = response.charAt(0);

      System.out.println();
```

FIGURE 10

The TravelPlanner Program with a Conditional Statement (continued)

```
        if (choice == 'D')
        {
          time = distance/speed;
          System.out.println("At " + speed + " mph, it will take "
             + time + " hours ");
          System.out.println("to travel " + distance + " miles.");
        }
        else
        {
          time = distance/speed;
          hours = (int)time;
          minutes = (int)((time - hours)*60);
          System.out.println("At " + speed + " mph, it will take "
             + hours + " hours " + minutes + " minutes");
          System.out.println("to travel " + distance + " miles.");
        }
     }
  }
```

that the user wants to compute the time in hours and minutes, even though *choice* may have any other value besides 'D' (including 'd') that the user may have typed in response to the prompt.

To compute hours and minutes (the "else" clause of the if-else statement), time is computed in the usual way, which results in a decimal value. The whole number part of that decimal is the number of hours needed for the trip. We can get this number by type casting the decimal number to an integer. This is accomplished by

```
hours = (int)time;
```

which drops all digits behind the decimal point and stores the resulting integer value in *hours*. To find the fractional part of the hour that we dropped, we subtract *hours* from *time*. We multiply this by 60 to turn it into some number of minutes, but this is still a decimal number. We do another type cast to truncate this to an integer value for *minutes*:

```
minutes = (int)((time - hours)*60);
```

For example, if the user enters data of 50 mph and 475 miles and requests output in hours and minutes, the following table shows the computed values.

Quantity	*Value*
speed	50
distance	475
time = distance/speed	9.5
hours = (int)time	9
time − hours	0.5
(time − hours) *60	30.0
minutes = (int)((time − hours)*60)	30

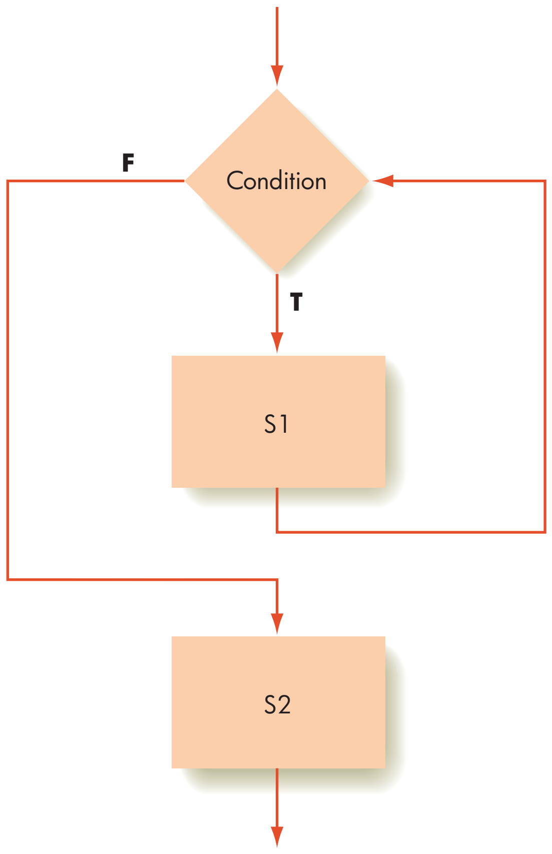
Here is the actual program output for this case:

```
Enter your speed in mph: 50
Enter your distance in miles: 475
Enter your choice of format for time,
decimal hours (D) or hours and minutes (M): M
At 50 mph, it will take 9 hours and 30 minutes
to travel 475 miles.
```

The two statement groups in an if-else statement are identified by the enclosing curly braces, but in Figure 10 we also indented them to make them easier to pick out when looking at the program. Like comments, indentation is ignored by the computer but is valuable in helping people to more readily understand a program.

Now let's look at the third variation on flow of control, namely looping (iteration). We want to execute the same group of statements (called the **loop body**) repeatedly, depending on the result of a Boolean condition. As long as (while) the condition remains true, the loop body is executed. The condition is tested before each execution of the loop body. When the condition becomes false, the loop body is not executed again, which is usually expressed by saying that the algorithm *exits* the loop. To ensure that the algorithm ultimately exits the loop, the condition must be such that its truth value can be affected by what happens when the loop body is executed. Figure 11 illustrates the while loop. The loop body is statement S1 (which can be a compound

FIGURE 11
While Loop



statement), and S1 is executed *while* the condition is true. Once the condition is false, the flow of control moves on to statement S2. If the condition is false when it is first evaluated, then the body of the loop is never executed at all. We saw this same scenario when we discussed pseudocode looping statements in Chapter 2 (Figure 2.6).

Java uses a while statement to implement this type of looping. The form of the statement is

```
while (Boolean condition)
    S1;
```

For example, suppose we want to write a program to add a sequence of nonnegative integers that the user supplies and write out the total. We need a variable to hold the total; we'll call this variable *sum* and make its data type *int*. To handle the numbers to be added, we could declare a bunch of integer variables such as *n1*, *n2*, *n3*, . . . and do a series of input-and-add statements of the form

```
n1 = inp.nextInt();
sum = sum + n1;
n2 = inp.nextInt();
sum = sum + n2;
```

and so on. There are two problems with this approach. The first is that we may not know ahead of time how many numbers the user wants to add. If we declare variables *n1*, *n2*, . . ., *n25*, and the user wants to add 26 numbers, the program won't do the job. The second problem is that this approach requires too much effort. Suppose that we know the user wants to add 2000 numbers. We could declare 2000 variables, (*n1*, . . ., *n2000*), and we could write the above input-and-add statements 2000 times, but it wouldn't be fun. Nor is it necessary—we are doing a very repetitive task here, and we should be able to use a loop mechanism to simplify the job. (We faced a similar situation in the first pass at a sequential search algorithm, Figure 2.11; our solution there was also to use iteration.)

Even if we use a loop mechanism, we are still adding a succession of values to *sum*. Unless we are sure that the value of *sum* is zero to begin with, we cannot be sure that the answer isn't nonsense. When we declare and initialize the variable *sum*, we should set its value to zero.

Now on to the loop mechanism. First, let's note that once a number has been read in and added to *sum*, the program doesn't need to know the value of the number any longer. We can declare just one integer variable called *number*, and use it repeatedly to hold the first numerical value, then the second, and so on.

The general idea is then

```
int number = 0;
int sum = 0;

while (there are more numbers to add)
{
      number = inp.nextInt();
      sum = sum + number;
}
System.out.println("The total is " + sum);
```

Now we have to figure out what the condition “there are more numbers to add” really means. Because we are adding nonnegative integers, we could ask the user to enter one extra integer that is not part of the legitimate data but is instead a signal that there *are* no more data. Such a value is called a **sentinel value**. For this problem, any negative number would be a good sentinel value. Because the numbers to be added are all nonnegative, the appearance of a negative number signals the end of the legitimate data. We don’t want to process the sentinel value (because it is not a legitimate data item); we only want to use it to terminate the looping process. This might suggest the following code:

```
int number = 0;
int sum = 0;
while (number >= 0) //but there is a problem here,
                    //see following discussion
{
     number = inp.nextInt();
     sum = sum + number;
}
System.out.println("The total is " + sum);
```

Here’s the problem. How can we test whether *number* is greater than or equal to 0 if we haven’t read the value of *number* yet? We need to do a preliminary input for the first value of *number* outside of the loop, then test that value in the loop condition. If it is nonnegative, we want to add it to *sum* and then read the next value and test it. Whenever the value of *number* is negative (including the first value), we want to do nothing with it—that is, we want to avoid executing the loop body. The following statements do this; we’ve also added instructions to the user.

```
int number = 0;
int sum = 0;
System.out.println("Enter numbers to add; "
    + "terminate with a negative number.");

number = inp.nextInt();
while (number >= 0)
{
  sum = sum + number;
  number = inp.nextInt();
}
System.out.println("The total is " + sum);
```

The value of *number* gets changed within the loop body by reading in a new value. The new value is tested, and if it is nonnegative, the loop body executes again, adding the data value to *sum* and reading in a new value for *number*. The loop terminates when a negative value is read in. Remember the requirement that something within the loop body must be able to affect the truth value of the condition. In this case, it is reading in a new value for *number* that has the potential to change the value of the condition from true to false. Without this requirement, the condition, once true, would remain true forever, and the loop body would be endlessly executed. This results in what is called an **infinite loop**. A program that contains an infinite loop will execute forever (or until the programmer gets tired of waiting and interrupts the program, or until the program exceeds some preset time limit).

Here is a sample of the program output.

```
Enter numbers to add; terminate with a negative
number.
5
6
10
-1
The total is 21
```

The problem we've solved here, adding nonnegative integers until a negative sentinel value occurs, is the same one solved using assembly language in Chapter 6. The Java code above is almost identical to the pseudocode version of the algorithm shown in Figure 6.7. Thanks to the power of the language, the Java code embodies the algorithm directly, at a high level of thinking, whereas in assembly language this same algorithm had to be translated into the lengthy and awkward code of Figure 6.8.

To process data for a number of different trips in the TravelPlanner program, we could use a while loop. During each pass through the loop, the program computes the time for a given speed and distance. The body of the loop is therefore exactly like our previous code. All we are adding here is the framework that provides looping. To terminate the loop, we could use a sentinel value, as we did for the program above. A negative value for *speed*, for example, is not a valid value and could serve as a sentinel value. Instead of that, let's allow the user to control loop termination by having the program ask the user whether he or she wishes to continue. We'll need a variable to hold the user's response to this question. Of course, the user could answer "N" at the first query, the loop body would never be executed at all, and the program would terminate. Figure 12 shows the complete program.

FIGURE 12

The TravelPlanner Program with Looping

```
// Computes and outputs travel time
// for a given speed and distance
// Written by J. Q. Programmer, 7/05/16

import java.util.*;
public class TravelPlanner
{
   public static void main(String[] args)
   {
      int speed = 1;          //rate of travel
      double distance = 0.0; //miles to travel
      double time = 0.0;     //time needed for travel (decimal)
      int hours = 0;          //time for travel in hours
      int minutes = 0;        //leftover time in minutes
      String response = " "; //user's response
      char choice = 'M';      //choice of output as decimal hours
                              //or hours and minutes
      char more = 'Y';        //user's choice to do another trip
      Scanner inp = new Scanner(System.in);
```

FIGURE 12

The TravelPlanner Program with Looping (continued)

```
         System.out.println("Do you want to plan "
             + "a trip? (Y or N): ");
         response = inp.next();
         more = response.charAt(0);

         while(more == 'Y')    //more trips to plan
         {

            System.out.println("Enter your speed in mph:");
            speed = inp.nextInt();
            System.out.println("Enter your distance in miles:");
            distance = inp.nextDouble();
            System.out.println("Enter your choice of format "
               + "for time, ");
            System.out.println("decimal hours (D) "
               + "or hours and minutes (M): ");
            response = inp.next();
            choice = response.charAt(0);
            System.out.println();

            if (choice == 'D')
            {
               time = distance/speed;
               System.out.println("At " + speed + " mph,"
                  + "it will take " + time + " hours ");
               System.out.println("to travel " + distance
                  + " miles.");
            }
            else
            {
               time = distance/speed;
               hours = (int)time;
               minutes = (int)((time - hours)*60);
               System.out.println("At " + speed + " mph,"
                 + "it will take " + hours + " hours " + minutes
                 + " minutes");
               System.out.println("to travel " + distance
                 + " miles.");
            }

            System.out.println();
            System.out.println("Do you want to plan "
               + "another trip? (Y or N): ");
            response = inp.next();
            more = response.charAt(0);
         }  //end of while loop
      }
   }
```

PRACTICE PROBLEMS

1. What is the output from the following section of code?

```
int number1 = 15;
int number2 = 7;
if (number1 >= number2)
  System.out.println(2*number1);
else
  System.out.println(2*number2);
```

2. What is the output from the following section of code?

```
int scores = 1;
while (scores < 20)
{
  scores = scores + 2;
  System.out.println(scores);
}
```

3. What is the output from the following section of code?

```
int quotaThisMonth = 7;
int quotaLastMonth = quotaThisMonth + 1;
if ((quotaThisMonth > quotaLastMonth) ||
  (quotaLastMonth >= 8))
{
  System.out.println("Yes");
  quotaLastMonth = quotaLastMonth + 1;
}
else
{
  System.out.println("No");
  quotaThisMonth = quotaThisMonth + 1;
}
```

4. How many times is the output statement executed in the following section of code?

```
int left = 10;
int right = 20;
while (left <= right)
{
  System.out.println(left);
  left = left + 2;
}
```

5. Write a Java statement that outputs “Equal” if the integer values of *night* and *day* are the same, but otherwise does nothing.

4 Another Example

Let's briefly review the types of Java programming statements we've learned. We can do input and output—reading values from the user into memory, writing values out of memory for the user to see, being sure to use meaningful variable identifiers to reference memory locations. We can assign values to variables within the program. And we can direct the flow of control by using conditional statements or looping. Although there are many other statement types available in Java, you can do almost everything using only the modest collection of statements we've described. The power of Java lies in how these statements are combined and nested within groups to produce ever more complex courses of action.

For example, suppose we write a program to assist SportsWorld, a company that installs circular swimming pools. In order to estimate their costs for swimming pool covers or for fencing to surround the pool, SportsWorld needs to know the area or circumference of a pool, given its radius. A pseudocode version of the program is shown in Figure 13.

We should be able to translate this pseudocode fairly directly into the body of the main method. Other things we need to add to complete the program are

- A prologue comment to explain what the program does (optional but always recommended for program documentation)
- An import statement so we can use the *Scanner* class for collecting input
- The class header; we'll call the class *SportsWorld*
- The main method header; remember this is always
  `public static void main(String[] args)`
- Variable declarations

Finally, the computations for circumference and area both involve the constant pi (π). We could use some numerical approximation for pi each time it occurs in the program, but the *Math* class of the standard Java library already defines the constant *PI*. We can invoke this constant value by writing

```
Math.PI
```

Figure 14 gives the complete program; the prologue comment notes the use of the *Math* class. Figure 15 shows what actually appears on the screen when this program is executed with some sample data.

It is inappropriate (and messy) to output the value of the area to 14 or 15 decimal places based on a value of the radius given to one or two decimal places of accuracy. Exercise 11 at the end of this chapter tells how to format real number output to a specified number of decimal digits.

FIGURE 13

A Pseudocode Version of the SportsWorld Program

```
Get value for user's choice about continuing
While user wants to continue, do the following steps
    Get value for pool radius
    Get value for choice of task
    If task choice is circumference
        Compute pool circumference
        Print output
    Else (task choice is area)
        Compute pool area
        Print output
    Get value for user's choice about continuing
Stop
```

FIGURE 14

The SportsWorld Program

```
//This program helps SportsWorld estimate costs
//for pool covers and pool fencing by computing
//the area or circumference of a circle
//with a given radius.
//Any number of circles can be processed.
//Uses class Math for PI
//Written by M. Phelps, 10/05/16

import java.util.*;
public class SportsWorld
{
  public static void main(String[] args)
  {
    double radius = 0.0;          //radius of a pool -
                                  //given
    double circumference = 0.0;   //pool circumference -
                                  //computed
    double area = 0.0;            //pool area - computed
    String response = " ";        //user's response
    char taskToDo = 'C';          //holds user choice to
                                  //compute circumference
                                  //or area
    char more = 'Y';              //controls loop for
                                  //processing
                                  //more pools
    Scanner inp = new Scanner(System.in);

    System.out.print("Do you want to process "
        + "a pool? (Y or N): ");

    response = inp.next();
    more = response.charAt(0);

    while(more == 'Y')      //more circles to process
    {
      System.out.println();
      System.out.print("Enter the value of the "
          + "radius of the pool: ");
      radius = inp.nextDouble();
```

FIGURE 14

The SportsWorld Program (continued)

```
      //See what user wants to compute
      System.out.println("Enter your choice of task.");
      System.out.print("C to compute circumference, "
          + "A to compute area: ");
      response = inp.next();
      taskToDo = response.charAt(0);
      System.out.println();

      if (taskToDo == 'C')          //compute circumference
      {
        circumference = 2*Math.PI*radius;
        System.out.println("The circumference for a "
           + "pool of radius " + radius + " is "
           + circumference);
      }
      else                          //compute area
      {
        area = Math.PI * radius * radius;
        System.out.println("The area for a pool"
           + " of radius " + radius + " is " + area);
      }
      System.out.println();
      System.out.print("Do you want to process "
          + "more pools? (Y or N): ");

      response = inp.next();
      more = response.charAt(0);
    } //end of while loop

    //finish up
    System.out.println("Program will now terminate.");

  } //end of main method
}   //end of class SportsWorld
```

FIGURE 15

A Sample Session Using the Program of Figure 14

```
Do you want to process a pool? (Y or N): Y
Enter the value of the radius of the pool: 2.7
Enter your choice of task.
C to compute circumference, A to compute area: C
The circumference for a pool of radius 2.7 is 16.964600329384883
Do you want to process more pools? (Y or N): Y
Enter the value of the radius of the pool: 2.7
Enter your choice of task.
C to compute circumference, A to compute area: A
The area for a pool of radius 2.7 is 22.902210444669592
Do you want to process more pools? (Y or N): Y
Enter the value of the radius of the pool: 14.53
Enter your choice of task.
C to compute circumference, A to compute area: C
```

FIGURE 15

A Sample Session Using the Program of Figure 14 (continued)

```
The circumference for a pool of radius 14.53 is 91.29468251331939
Do you want to process more pools? (Y or N): N
Program will now terminate.
```

PRACTICE PROBLEMS

1. Write a complete Java program to read in the user's first and last initials and write them out.
2. Write a complete Java program that asks for the price of an item and the quantity purchased, and writes out the total cost.
3. Write a complete Java program that asks for a number. If the number is less than 5, it is written out, but if it is greater than or equal to 5, twice that number is written out.
4. Write a complete Java program that asks the user for a positive integer *n*, and then writes out all the numbers from 1 up to and including *n*.

5 Managing Complexity

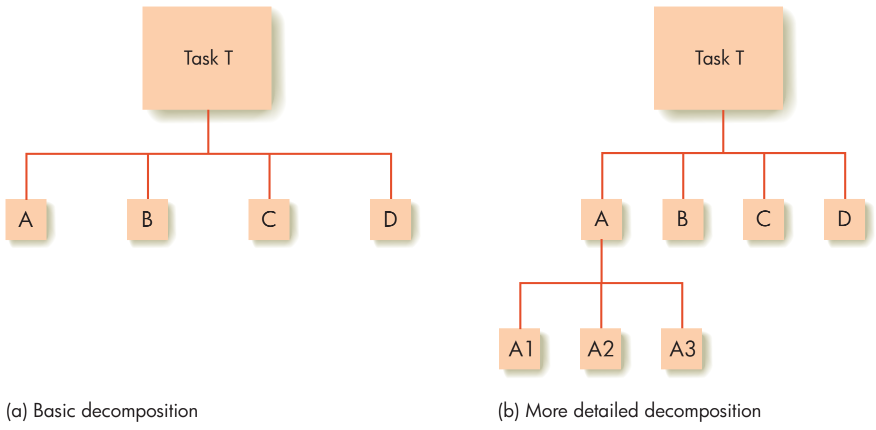
The programs we have written have been relatively simple. More complex problems require more complex programs to solve them. Although it is fairly easy to understand what is happening in the 50 or so lines of the SportsWorld program, imagine trying to understand a program that is 50,000 lines long. Imagine trying to write such a program! It is not possible to understand—all at once—everything that goes on in a 50,000-line program.

5.1 *Divide and Conquer*

Writing large programs is an exercise in managing complexity. The solution is a problem-solving approach called **divide and conquer**. Suppose a program is to be written to do a certain task; let's call it task T. Suppose further that we can divide this task into smaller tasks, say A, B, C, and D, such that, if we can do those four tasks in the right order, we can do task T. Then our high-level understanding of the problem need only be concerned with *what* A, B, C, and D do and how they must work together to accomplish T. We do not, at this stage, need to understand *how* tasks A, B, C, and D can be done. Figure 16(a), an example of a **structure chart** or **structure diagram**, illustrates this situation. Task T is composed in some way of subtasks A, B, C, and D. Later we can turn our attention to, say, subtask A, and see if it too can be decomposed into smaller subtasks, as in Figure 16(b). In this way, we continue to break the task down into smaller and smaller pieces, finally arriving at subtasks that are simple enough that it is easy to write the code to carry them out. Better yet,

FIGURE 16

Structure Charts



we may find a helpful class with methods that will do these subtasks for us. By *dividing* the problem into small pieces, we can *conquer* the complexity that is overwhelming if we look at the problem as a whole.

Divide and conquer is a problem-solving approach and not just a computer programming technique. Outlining a term paper into major and minor topics is a divide-and-conquer approach to writing the paper. Doing a Form 1040 Individual Tax Return for the Internal Revenue Service can involve subtasks of completing Schedules A, B, C, D, and so on, and then reassembling the results. Designing a house can be broken down into subtasks of designing floor plans, wiring, plumbing, and the like. Large companies organize their management responsibilities using a divide-and-conquer approach; what we have called structure charts become, in the business world, organization charts.

How is the divide-and-conquer problem-solving approach reflected in the resulting computer program? If we think about the problem in terms of subtasks, then the program should show that same structure; that is, part of the code should do subtask A, part should do subtask B, and so on. We divide the code into *modules* or *subprograms*, each of which does some part of the overall task. Then we empower these modules to work together to solve the original problem.

Which Java?

Java programs come in two renditions, **Java applications** and **Java applets**. Applications are complete stand-alone programs that reside and run on some computer. These are the kinds of programs we have been working with in this module.

But Java's development went hand in hand with the development of Web browsers. Applets (small applications) are programs designed to run from Web pages. The bytecode for an applet is embedded in a Web page on a server machine; when the user views the Web page with a Java-enabled browser, a copy of the applet's bytecode is temporarily transferred to the user's system (whatever that system may be) and interpreted/executed by the browser itself. Today's common Web browsers, such as Internet Explorer, Firefox, Chrome, and Safari, are Java-enabled. Java applets bring audio, video, and real-time user interaction to Web pages, making them "come alive" and become much more than static hyperlinked text. For example, a Java applet might display an animated analog clock face on the screen that shows your computer system's time, or a streaming ticker tape of stock market quotes, or a form that allows you to book an airline reservation online. Java applets held much of the original appeal of the Java language, but big, serious programs are also written using Java applications.

5.2 *Using Methods*

In Java, modules of code are called **methods**. We have already seen that a main method is required in each Java program. In the SportsWorld program (Figure 14) the main method appears to do the entire task. But the divide-and-conquer approach is already at work here. The main method does not handle the subtasks of reading various kinds of input. Instead, it creates an object from the *Scanner* class that in turn uses methods from the *Scanner* class, such as *nextDouble()*, that provide those services. The main method does not really do all the details of writing output, either; it makes use of the *print* and *println* methods.

Let's review the main method of the SportsWorld program with an eye to further subdividing the task. There is a loop that does some operations as long as the user wants. What gets done? Input is obtained from the user about the radius of the circle and the choice of task to be done (compute circumference or compute area). Then the circumference or the area gets computed and written out. Aside from input and output, we can identify two related subtasks: computing the area of a circle and computing the circumference of a circle. Instead of having the main method do these computations, we will create a *Circle* class (Figure 17) with two methods that provide these two services to the main method. A Java program can have only one main method, and that is where execution of the program begins. Figure 18 shows a pseudocode description of the main method using a modular approach that calls on the methods in the *Circle* class. When the flow of control reaches the "Ask Circle class to compute circumference," it transfers to the appropriate method code in the *Circle* class and executes that code. When execution of that method code is complete, flow of control transfers back to the main method and picks up where it left off. The same thing happens for "Ask *Circle* class to compute area".

Methods are named using ordinary Java identifiers, customarily starting with a lowercase letter. We'll name the two Circle methods *doCircumference* and *doArea*. Because we're using meaningful identifiers, it is obvious which subtask is carried out by which method.

There are two types of methods. A **void method** carries out some task, perhaps using values it receives from the main method, but does not pass any new values back to the main method. (The word *void* signifies "returning nothing.") A **nonvoid method** returns a single new value back to the main method—that is its primary job. This gives the main method information it did not have pre-

FIGURE 17
Methods in the Circle *Class*

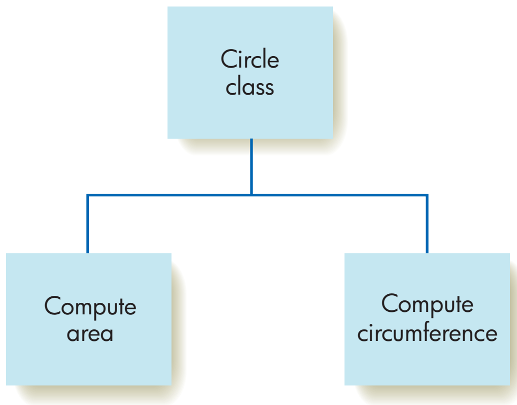


FIGURE 18

Pseudocode for the SportsWorld Main Method Using the Circle *Class*

```
Get value for user's choice about continuing
While user wants to continue, do the following steps
    Get value for pool radius
    Get value for choice of task
    If task choice is circumference
        Ask Circle class to compute circumference
        Print output
    Else (task choice is area)
        Ask Circle class to compute area
        Print output
    Get value for user's choice about continuing
Stop
```

viously. This single value can be some new data value that the method has collected from the user, or it can be some new data value that the method has computed, perhaps using values it received from the main method.

The *doCircumference* and *doArea* methods compute new values (the circumference and area, respectively) and return them to the main method. So *doCircumference* and *doArea* are nonvoid methods.

Either kind of method may need certain information from the main method to do its job; specifically, it may need to know the current value of certain quantities in the main method. When the main method wants a method in another class to be executed, it must "invoke" the method. It does this by giving the class name, followed by a dot, then the method name, and finally a list in parentheses of the identifiers for variables that concern that method. This is called an **argument list**. The overall form of a method invocation is thus

    class-identifier.method-identifier(argument list)

The *doCircumference* and *doArea* methods each need to know the current radius value in order to carry out their computations, so when these methods are invoked, each of their argument lists consists of the single variable *radius*.

The invocation of a void method is a complete Java program statement by itself (followed by the semicolon, of course), but the invocation of a nonvoid method is not. Remember that a nonvoid method returns a single value to the main method. You can think of the method invocation just as if it were a variable containing that single returned value. You cannot have the method invocation as a complete statement, just as you cannot have a single variable identifier as a complete statement. Instead, you use the method invocation as part of a statement, in the same way you use any variable identifier. For example, you can make it part of what an output statement writes out, or what an assignment statement assigns to a variable. In our circle program, when the *doCircumference* method returns the value of the circle's circumference, we would like to assign that value to the main method's *circumference* variable, so we use the assignment statement

```
    circumference = Circle.doCircumference(radius);
```

in the main method.

Figure 19 shows the new main method. It closely follows the pseudocode of Figure 18. At a glance, it does not look a great deal different from our

FIGURE 19

A Modularized Version of the SportsWorld Program

```
//This program helps SportsWorld estimate costs
//for pool covers and pool fencing by computing
//the area or circumference of a circle
//with a given radius.
//Any number of circles can be processed.
//Uses class Circle
//Written by M. Phelps, 10/23/16

import java.util.*;
public class SportsWorld
{
  public static void main(String[] args)
  {
    double radius = 0.0;               //radius of a pool -
                                       //given
    double circumference = 0.0;        //pool circumference -
                                       //computed
    double area = 0.0;                 //pool area - computed
    String response = " ";             //user's response
    char taskToDo = 'C';               //holds user choice to
                                       //compute circumference
                                       //or area
    char more = 'Y';                   //controls loop for
                                       //processing
                                       //more pools

    Scanner inp = new Scanner(System.in);

    System.out.print("Do you want to process "
        + "a pool? (Y or N): ");

    response = inp.next();
    more = response.charAt(0);

    while(more == 'Y')                 //more pools to process
    {
      System.out.println();
      System.out.print("Enter the value of the "
          + "radius of the pool: ");
      radius = inp.nextDouble();

      //See what user wants to compute
      System.out.println("Enter your choice of task.");
      System.out.print("C to compute circumference, "
          + "A to compute area: ");

      response = inp.next();
      taskToDo = response.charAt(0);
      System.out.println();
      System.out.println();
```

FIGURE 19

A Modularized Version of the SportsWorld Program (continued)

```
      if (taskToDo == 'C')            //compute circumference
      {
        circumference = Circle.doCircumference(radius);
        System.out.println("The circumference for a pool"
            + " of radius " + radius + " is " + circumference);
      }
      else                            //compute area
      {
        area = Circle.doArea(radius);
        System.out.println("The area for a pool"
                 + " of radius " + radius + " is " + area);
      }

       System.out.println();
       System.out.print("Do you want to process "
           + "more pools? (Y or N): ");

       response = inp.next();
       more = response.charAt(0);
     } //end of while loop

     //finish up
     System.out.println("Program will now terminate.");

   } //end of main method
  }  //end of class SportsWorld
```

former main method. However, it is conceptually quite different; it uses a helping class (*Circle*), and the subtasks of computing the circumference and computing the area have been relegated to methods of this class. The details (in this case the formulas for computing circumference and area) are now hidden and have been replaced by method invocations. If these subtasks had required many lines of code, our new main method would indeed be shorter—and easier to understand—than before.

The main method now invokes methods of the *Circle* class. It is time to see how to write the code for these other, nonmain methods.

5.3 *Writing Methods*

The outline for a Java method is shown in Figure 20. The method header has the general form

scope-indicator return-indicator identifier(parameter list)

Note that no semicolon appears at the end of a method header.

Let's look at each of these parts in turn.

- *scope-indicator*. The scope indicator uses keywords to determine how and where the method can be invoked. If the scope indicator is *public static*, then any method can invoke this method by giving the name of

FIGURE 20

The Outline for a Java Method

```
method header
//comment
{
     local declarations [optional]
     method body
}
```

the class, then a dot, then the method identifier and argument list. This is the syntax that the main method in Figure 19 uses to invoke the two methods of the *Circle* class.

- *return-indicator*. The return indicator classifies a method as void or nonvoid. If it's a void method, the return indicator is just the word *void*. If it's a nonvoid method, the return indicator is the data type of the single value the method returns.
- *identifier*. This is the name of the method and can be any legal Java identifier, although most programmers use a lowercase letter to begin the name.
- *parameter list*. The parameters in the **parameter list** correspond to the arguments in the statement that invoke this method; that is, the first parameter in the list matches the first argument given in the statement that invokes the method, the second parameter matches the second argument, and so on. It is through this correspondence between parameters and arguments that the method receives data from the invoking method. The data type of each parameter must be given as part of the parameter list, and it must match the data type of the corresponding argument.

For example, consider a method *findAverage* within class *Weather* to compute and return the average daily rainfall over a certain number of days. The total rainfall (a real number) and the number of days (an integer) are data values the method needs to know in order to compute the daily average, and these values are passed to the method as arguments. The value returned by the method, the daily average, is type *double*. This method can be invoked in the main method by a statement such as

```
dailyAverage = Weather.findAverage(totalRain, days);
```

The header for the *findAverage* method could look like

```
public static double findAverage(double total, int n)
```

Here the parameters *total* and *n* are in the correct order and have the correct data type to match with their corresponding arguments. The argument names, *totalRain* and *days*, are variable identifiers declared in the main method, but the parameters can have different identifiers, as they do here. Arguments and parameters correspond by virtue of their respective positions in the argument list and the parameter list, regardless of the identifiers used. Within the body of the method, it is the parameter identifiers that are used; *total* has the value passed to it by *totalRain*, and *n* has the value passed to it by *days*, as follows:

```
    dailyAverage = Weather.findAverage(totalRain, days);

    public static double findAverage(double total, int n)
```

Arguments in Java are **passed by value**. This means that the method can use the argument value but cannot permanently change it. What really happens is that the method receives a copy of the data value to store in a local memory location, but never knows the memory location where the original value is stored. If the method changes the value of its copy, this change has no effect when control returns to the main method.

In the Circle program, the *doCircumference* method is invoked with a single argument *radius* of type *double*. It is a nonvoid method and returns a type *double* value. Its header can be written as

```
    public static double doCircumference(double radius)
```

where this time we used the same name for the parameter as for the argument.

The complete *doCircumference* method is shown in Figure 21. Because it is a separate method, we have added a comment right below the method header to describe specifically what this method does. A variable *circumference* is declared within the method. A variable declared within a method is known and can be used only within that method; it is said to be **local** to that method. This local variable *circumference* has nothing to do with the *circumference* variable in the main method of the *SportsWorld* class. It is natural to use the same name for each, but the program works perfectly well if we name this local variable something entirely different.

Because *doCircumference* is a nonvoid method, it must return a single value to the main method. This is done by the **return statement**, whose syntax is

```
    return expression;
```

The expression must evaluate to the data type that the nonvoid method has promised to return in its header, which in the case of *doCircumference* is type *double*. All nonvoid methods must have a return statement, but void methods generally do not have a return statement.

FIGURE 21
The doCircumference *Method*

```
public static double doCircumference (double radius)
//returns circumference of a Circle
{
  double circumference;
  circumference = 2*Math.PI*radius;
  return circumference;
}
```

The *doArea* method is very similar to *doCircumference*. The complete *Circle* class is given in Figure 22. Notice that this class has no main method. A Java program always begins execution with the main method, so the code in Figure 22 will compile, but by itself, it cannot be executed. It is not a standalone program but a useful tool.

To run the program, each class must be in a separate file, and the filename must be the name of the class with a .java extension. So there is a *SportsWorld.java* file and a *Circle.java* file. Each .java file is compiled into a .class file, and the .class file containing the main method is then executed. It is helpful if all these files are in the same folder or directory on your computer so that the system knows where to find them.

So there we have it—a complete modularized version of our SportsWorld program. Because it seems to have taken a lot of effort to arrive at this second version (which, after all, does the same thing as the program in Figure 14), let's review what the new version does and why this effort is worthwhile. The major task is accomplished by doing a series of subtasks (computing circumference and area), and the work for these subtasks takes place within methods of a separate class. The main method doesn't need to know how these tasks are done; it only needs to invoke the appropriate method at the appropriate point. As an analogy, we may think of the president of a company calling on various assistants to carry out tasks as needed. The president does not need to know *how* a task is done, only the company division (class name) and name of the person (method name) responsible for carrying it out.

This compartmentalization is useful in many ways. It is useful when we *plan the solution* to a problem, because it allows us to use a divide-and-conquer approach. We can think about the problem in terms of subtasks.

FIGURE 22
The Circle Class in a Modularized Version of the SportsWorld Program

```
//Class for circles. Computes circumference and
//area, given radius.
//Uses class Math for PI
//Written by I. M. Euclid, 10/23/16

public class Circle
{
  public static double doCircumference(double radius)
  //returns circumference of a circle
  {
    double circumference;
    circumference = 2*Math.PI*radius;
    return circumference;
  }
  public static double doArea(double radius)
  //returns area of a circle
  {
    double area;
    area = Math.PI * radius * radius;
    return area;
  }
}
```

This makes it easier for us to understand how to achieve a solution to a large and complex problem. We can group similar subtasks together and think of them as methods of a helping class. It is also useful when we *code the solution* to a problem. Instead of having to write every detail of the code in a monolithic main method, we can write a main method that invokes other methods of other classes as needed. We can write methods for these other classes one at a time, so that the program gradually expands. Developing a large software project is a team effort, and different parts of the team can be writing different classes and methods at the same time. It is useful when we *test the program*, because we can test one new method at a time as the program grows, and any errors are localized to the method being added. (The main method can be tested early by writing appropriate headers but empty bodies for the other methods.) Compartmentalization is useful when we *modify the program*, because changes tend to be local to certain subtasks, hence within certain methods in the code. And finally it is useful for anyone (including the programmer) who wants to *read* the resulting program. The overall idea of how the program works, without the details, can be gleaned from reading the main method; if and when the details become important, the appropriate code for the other methods can be consulted. In other words, modularizing a program is useful for its

- Planning
- Coding
- Testing
- Modifying
- Reading

Finally, once a class has been developed and tested, it is then available for any application program to use. An application program that does quite different things than SportsWorld, but that needs the value of the area or circumference of a circle computed from the radius, can use our *Circle* class.

Figure 23 summarizes several terms introduced in this section.

FIGURE 23
Some Java Terminology

TERM	MEANING	TERM	MEANING
void method	Performs a task, but returns no value; method invocation is a complete Java statement	nonvoid method	Computes a value, must include a return statement; method invocation is used within another Java statement
argument	Variable passed to method when it is invoked	parameter	"Dummy variable" in a method that receives its value from the corresponding argument
local variable	Declared and known only within a method		
argument passed by value	Method receives a copy of the value and can make no permanent changes in the value		

PRACTICE PROBLEMS

1. What is the output of the following Java program?

```
public class Problem1
{
  public static void main(String[] args)
  {
    int number = 10;
    int newNumber = 0;
    newNumber = Helper.doIt(number);
    System.out.println(newNumber);
  }
}
public class Helper
{
  public static int doIt(int n)
  {
    int twice = 0;
    twice = 2*n;
    return twice;
  }
}
```

2. What is the output of the following Java program?

```
public class Problem2
{
  public static void main(String[] args)
  {
    int number = 10;
    System.out.println(Helper.doIt(number));
  }
}
public class Helper
{
  public static int doIt(int n)
  {
    return 2*n;
  }
}
```

3. What is the output of the following Java program?

```
public class Problem3
{
  public static void main(String[] args)
  {
    int number = 10;
    System.out.println(Helper.doIt(number));
    System.out.println(number);
  }
}
```

(continues)

PRACTICE PROBLEMS (continued)

```
public class Helper
{
  public static int doIt(int number)
  {
    number = 7;
    System.out.println(number);
    return 2*number;
  }
}
```

4. Suppose a nonvoid method called *tax* in a class called *Sales* gets a value *subtotal* from the main method, multiplies *subtotal* by the tax rate of 0.55, and returns the resulting tax amount. All quantities are type *double*.
   a. Write the method header.
   b. Write the method body.
   c. Write a single statement in the main method that invokes the *tax* method and writes out the resulting tax amount.

6 Object-Oriented Programming

6.1 *What Is It?*

The divide-and-conquer approach to programming is a “traditional” approach. The focus is on the overall task to be done: How to break it down into subtasks, and how to write algorithms for these subtasks that are carried out by communicating modules—in the case of Java, by methods in various classes. The program can be thought of as a giant statement executor designed to carry out the major task, even though the main module may simply call on, in turn, the various other modules that do the subtask work.

Object-oriented programming (OOP) takes a somewhat different approach. A program is considered a simulation of some part of the world that is the domain of interest. “Objects” populate this domain. Objects in a banking system, for example, might be savings accounts, checking accounts, and loans. Objects in a company personnel system might be employees. Objects in a medical office might be patients and doctors. Each object is an example drawn from a class of similar objects. The savings account “class” in a bank has certain properties associated with it, such as name, Social Security number, account type, and account balance. Each individual savings account at the bank is an example of (an object of) the savings account class, and each has specific values for these common properties; that is, each savings account has a specific value for the name of the account holder, a specific value for the account balance, and so forth. Each object of a class therefore has its own data values.

A class also has one or more subtasks associated with it, and all objects from that class can perform those subtasks. In carrying out a subtask, each object can be thought of as providing some service. A savings account, for example, can compute compound interest due on the balance. When an object-oriented program is executed, the program generates requests for services that go to the various objects. The objects respond by performing the requested service—that is, carrying out the subtask. Thus, a program that is using the savings account class might request a particular savings account object to perform the service of computing interest due on the account balance. An object always knows its own data values and may use them in performing the requested service.

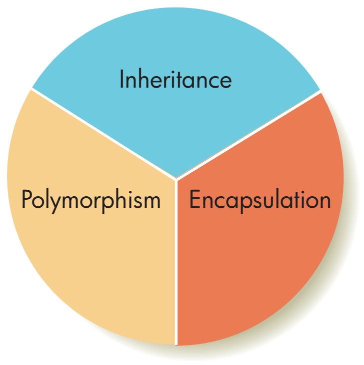
Some of this sounds familiar. We know about subtasks (methods) associated with a class. The new idea is that, instead of directly asking a class to carry out a subtask, we ask an object of that class to carry out a subtask. The even bigger new idea is that such objects have data values for the class properties. Instead of storing data in variables that are available to the whole program and then passing them as arguments to subtasks, the program can simply ask an object to use its own data when it carries out a subtask.

There are three terms often associated with object-oriented programming, as illustrated in Figure 24. The first term is **encapsulation**. Each class has its own program module to perform each of its subtasks. Any user of the class (which might be some other program) can ask an object of that class to invoke the appropriate module and thereby perform the subtask service. The class user needs to know what services objects of the class can provide and how to request an object to perform any such service. The details of the module code belong to the class itself, and this code may be modified in any manner, as long as the way the user interacts with the class remains unchanged. (In the savings account example, the details of the algorithm used to compute interest due belong only to the class, and need not be known by any user of the class. If the bank wants to change how it computes interest, only the code for the interest module in the savings account class needs to be modified; any programs that use the services of the savings account class can remain unchanged.) Furthermore, the class properties represent data values that will exist as part of each object of the class. A class therefore consists of two components, its subtask modules and its properties, and both components are "encapsulated"—bundled—with the class.

A second term associated with object-oriented programming is **inheritance**. Once a class A of objects is defined, a class B of objects can be defined as a

FIGURE 24

Three Key Elements of OOP



"subclass" of A. Every object of class B is also an object of class A; this is sometimes called an "is a" relationship. Objects in the B class will "inherit" all of the properties and be able to perform all the services of objects in A, but they may also be given some special property or ability. The benefit is that class B does not have to be built from the ground up, but rather can take advantage of the fact that class A already exists. In the banking example, a senior citizen's savings account would be a subclass of the savings account class. Any senior citizens' savings account object is also a savings account object, but it may have special properties or be able to provide special services.

The third term is **polymorphism**. *Poly* means "many." Objects may provide services that should logically have the same name because they do roughly the same thing, but the details differ. In the banking example, both savings account objects and checking account objects should provide a "compute interest" service, but the details of how interest is computed differ in these two cases. Thus, one name, the name of the service to be performed, has several meanings, depending on the class of the object providing the service. It may even be the case that more than one service with the same name exists for the same class, although there must be some way to tell which service is meant when it is invoked by an object of that class.

Let's change analogies from the banking world to something more fanciful, and consider a football team. Every member of the team's backfield is an "object" of the "backfield" class. The quarterback is the only "object" of the "quarterback" class. Each backfield object can perform the service of carrying the ball if he (or she) receives the ball from the quarterback; ball-carrying is a subtask of the backfield class. The quarterback who hands the ball off to a backfield object is requesting that the backfield object perform that subtask because it is "public knowledge" that the backfield class carries the ball and that this service is invoked by handing off the ball to a backfield object. The "program" to carry out this subtask is *encapsulated* within the backfield class, in the sense that it may have evolved over the week's practice and may depend on specific knowledge of the opposing team, but at any rate, its details need not be known to other players. *Inheritance* can be illustrated by the halfback subclass within the backfield class. A halfback object can do everything a backfield object can but may also be a pass receiver. And *polymorphism* can be illustrated by the fact that the backfield may invoke a different "program" depending on where on the field the ball is handed off. Of course our analogy is imperfect, because not all human "objects" from the same class behave in precisely the same way—fullbacks sometimes receive passes and so on.

6.2 *Java and OOP*

Java is very much an object-oriented programming language. We learned right at the beginning of this chapter that all Java code (except for comments and import statements) must be either a class header or inside a class definition. In the modularized version of our SportsWorld program, we use a *Circle* class with methods that the main method in the *SportsWorld* class can invoke. The main method does not create any objects of the *Circle* class, but can nevertheless invoke these methods, because each method has the word "static" in the method header. A **static method** is one that doesn't need to be invoked by an object of that class. Instead, it can be invoked by giving the class name, followed by a dot, and then the method name with an appropriate list of arguments, just as we have done with all the methods we have used so far.

Suppose we write a new *Circle* class with the assumption that applications programs using this class will create objects of the class. The objects are individual circles. A Circle object has a radius. A Circle object, which knows the value of its own radius, should be able to perform the services of computing its own circumference and its own area. At this point, we are well on the way to answering the two major questions about our new *Circle* class:

- What are the properties common to any object of this class? (In this case, there is a single property—the radius.)
- What are the services that any object of the class should be able to perform? (In this case, it must be able to compute its circumference and compute its area, although as we will see shortly, we will need other services as well.)

Now we can create a truly object-oriented version of the SportsWorld program. What are the objects of interest within the scope of this problem? SportsWorld deals with circular swimming pools, but they are basically just circles. So the SportsWorld program creates a Circle object. In Java terminology, objects are called **instances of a class**, the properties are called **instance variables**, and the services are called **instance methods**.

Figure 25 shows the complete code for the new *Circle* class. Four instance methods are given, followed by a declaration of the single instance variable, *radius*. The first method is void, and the remaining three return values. None of the methods is static, meaning that they must be invoked by Circle objects.

As before, the *SportsWorld* class handles all of the user interaction and makes use of the *Circle* class. It creates a Circle object and requests that object to set the value of its radius and to find its area or find its circumference, depending on the program user's preference. The object invokes the *Circle* methods to carry out these tasks. From Figure 25, we see that the *setRadius* method uses an assignment statement to change the value of *radius* to whatever quantity is passed to the parameter *value*. The *doCircumference* and *doArea* methods use the usual formulas for their computations, but instead of using local variables for circumference and area, we've compressed the code into a single return statement. (This has nothing to do with object orientation; we could have done this in version 2 of the program.) The purpose of the *getRadius* method will be explained shortly.

The methods of the *Circle* class are all declared using the keyword **public**. Public methods can be used anywhere, including any Java program (like SportsWorld) that wants to make use of this class. Think of the *Circle* class as handing out a business card that advertises these services: Hey, you want a Circle object that can find its own area? Find its own circumference? Set the value of its own radius? I'm your class! (Class methods can also be **private**, but a private method is a sort of helping task that can be used only within the class in which it occurs.)

The single instance variable of the class (*radius*) is declared using the keyword private. Only methods in the *Circle* class itself can use this variable. Note that *doCircumference* and *doArea* have no parameter for the value of the radius; as methods of this class, they know at all times the current value of

FIGURE 25

The New Circle Class

```
//Class for Circle objects. A circle has a radius
//and can compute its circumference and area
//Uses class Math for PI
//Written by I.M. Euclid, 10/23/16

public class Circle
{
  public void setRadius(double value)
  //sets radius equal to value
  {
    radius = value;
  }
  public double getRadius()
  //returns current radius
  {
    return radius;
  }
  public double doCircumference()
  //computes and returns circumference of a circle
  {
    return 2*Math.PI*radius;
  }
  public double doArea()
  //computes and returns area of a circle
  {
    return Math.PI * radius * radius;
  }
  //instance variable
  private double radius;
}
```

radius for the object that invoked them, and it does not have to be passed to them as an argument. Because *radius* has been declared private, however, the *SportsWorld* class cannot use the value of *radius*. It cannot write out that value or directly change that value by some assignment statement. It can, however, request a Circle object to invoke the *getRadius* method to return the current value of the radius in order to write it out. It can also request a Circle object to invoke the *setRadius* method to change the value of its radius; *setRadius* does have a parameter to receive a new value for *radius*. Instance variables are generally declared private instead of public, to protect the data in an object from reckless changes some application program might try to make. Changes in the values of instance variables should be performed only under the control of class objects through methods such as *setRadius*.

The new *SportsWorld* class (Figure 26) differs from the earlier version (Figure 19) in several ways. The main method must create a Circle object, an instance of the *Circle* class. The following statement does this:

```
Circle swimmingPool = new Circle();
```

FIGURE 26

The New SportsWorld Class

```
//This program helps SportsWorld estimate costs
//for pool covers and pool fencing by computing
//the area or circumference of a circle
//with a given radius.
//Any number of circles can be processed.
//Uses class Circle
//Written by M. Phelps, 11/12/16

import java.util.*;
public class SportsWorld
{
  public static void main(String[] args)
  {
    double newRadius = 0.0; //radius of a pool - given
    String response = " ";  //user's response
    char taskToDo = 'C';    //holds user choice to
                            //compute circumference
                            //or area
    char more = 'Y';        //controls loop for
                            //processing
                            //more pools
    Scanner inp = new Scanner(System.in);
    Circle swimmingPool = new Circle(); //create a
                                        //Circle object

    System.out.print("Do you want to process "
       + "a pool? (Y or N): ");

    response = inp.next();
    more = response.charAt(0);

    while(more == 'Y') //more pools to process
     {
       System.out.println();
       System.out.print("Enter the value of the "
          + "radius of the pool: ");
       newRadius = inp.nextDouble();

       swimmingPool.setRadius(newRadius); //give pool
                                          //this radius

       //See what user wants to compute
       System.out.println("Enter your choice of task.");
       System.out.print("C to compute circumference, "
          + "A to compute area: ");

       response = inp.next();
       taskToDo = response.charAt(0);
       System.out.println();
       System.out.println();
```

FIGURE 26

The New SportsWorld Class (continued)

```
        if (taskToDo == ‘C’) //compute circumference
        {
          System.out.println("The circumference for a pool"
             + " of radius " + swimmingPool.getRadius()
             + " is " + swimmingPool.doCircumference() );
        }
        else                //compute area
        {
          System.out.println("The area for a pool"
             + " of radius " + swimmingPool.getRadius()
             + " is " + swimmingPool.doArea() );
        }

        System.out.println();
        System.out.print("Do you want to process "
             + "more pools? (Y or N): ");

        response = inp.next();
        more = response.charAt(0);
     } //end of while loop

     //finish up
     System.out.println("Program will now terminate.");

   } //end of main method
 }   //end of class SportsWorld
```

The left side of this statement:

```
Circle swimmingPool
```

looks like an ordinary variable declaration such as

```
int number
```

It seems to be saying, “Give me a memory location in which I will store something of type *Circle* and call it *swimmingPool*.” What we are asking for, however, is memory space to store the instance variables of the object, of which there might be many for some classes of objects. Unlike ordinary variables, Java does not give us memory locations in which to store the instance variables of an object until we specifically request “new” memory for this purpose via the right side of the statement

```
new Circle()
```

After

```
Circle swimmingPool = new Circle();
```

the object *swimmingPool* exists, and the main method can ask *swimmingPool* to perform the various services of which instances of the *Circle* class are capable.

The syntax to request an object to invoke a method is to give the name of the object, followed by a dot, followed by the name of the class method, followed by any arguments the method may need.

object-identifier.method-identifier(argument list)

The object that invokes a method is the **calling object**. Therefore the expression

```
swimmingPool.doCircumference()
```

in the main method uses *swimmingPool* as the calling object to invoke the *doCircumference* method of the *Circle* class. No arguments are needed because this method has no parameters, but the empty parentheses must be present.

There are no variables in the main method for the circumference and the area of the circle. The *doCircumference* and *doArea* methods are now invoked within an output statement, so these values get printed out without being stored anywhere. (This has nothing to do with object orientation; we could have done this in version 2 of the program.) But there is also no declaration in the main method for a variable called *radius*. There is a declaration for *newRadius*, and *newRadius* receives the value entered by the user for the radius of the circle. Therefore, isn't *newRadius* serving the same purpose as *radius* did in the old program? No—this is rather subtle, so pay attention: While *newRadius* holds the number the user wants for the circle radius, it is not itself the radius of *swimmingPool*. The radius of *swimmingPool* is the instance variable *radius*, and only methods of the class can change the instance variables of an object of that class. The *Circle* class provides the *setRadius* method for this purpose. The main method of *SportsWorld* must ask the object *swimmingPool* to invoke *setRadius* to set the value of its radius equal to the value contained in *newRadius*. The *newRadius* argument corresponds to the *value* parameter in the *setRadius* method, which then gets assigned to the instance variable *radius*.

```
swimmingPool.setRadius(newRadius);

public void setRadius(double value)
//sets radius equal to value
{
  radius = value;
}
```

The *setRadius* method is a void method because it returns no information to the invoking method; it contains no return statement. The invocation of this method is a complete Java statement.

Finally, the output statements that print the values of the circumference and area also have *swimmingPool* invoke the *getRadius* method to return its current *radius* value so it can be printed as part of the output. We could have used the variable *newRadius* here instead. However, *newRadius* is what we THINK has been used in the computation, whereas *radius* is what has REALLY been used.

Now that we understand the syntax, we can see that an output statement such as

```
System.out.println("Here's your output: " + answer);
```

asks the *System.out* object to invoke a *println* method using the string parameter included in the parentheses. The *System.out* object is unusual in that we do not have to explicitly create it using a "new" statement.

This completes version 3 of the SportsWorld program, a truly object-oriented version. The main method creates a Circle object and repeatedly requests that object to perform (or, technically, cause to have performed) the appropriate methods of its class to set its own radius and compute its circumference and area. Many people would say this is the only good way to write a Java program!

6.3 *One More Example*

The object-oriented version of our SportsWorld program illustrates encapsulation. All data and calculations concerning circles are encapsulated in the *Circle* class. Let's look at one final example that illustrates the other two watchwords of OOP—polymorphism and inheritance.

Figure 27(a)–(d) shows four simple geometric shape classes. Figure 27(e) is the application program that uses these classes. The main method creates objects from these various classes and has those objects set their dimensions and compute their areas. Each of these five classes is in a separate .java file of the same name as the class.

FIGURE 27

A Java Program with Polymorphism and Inheritance

```
//Class for circles. Area can be
//computed from radius.

public class Circle
{
  public void setRadius(double value)
  //sets radius equal to value
  {
    radius = value;
  }

  public double getRadius()
  //returns current radius
  {
    return radius;
  }

  public double doArea()
  //computes and returns area of a circle
  {
    return Math.PI * radius * radius;
  }

  //instance variable
  private double radius;
}
```

(a) The *Circle* Class

FIGURE 27

A Java Program with Polymorphism and Inheritance (continued)

```
//Class for rectangles. Area can be
//computed from length and width.

public class Rectangle
{
  public void setWidth(double value)
  //sets width equal to value
  {
    width = value;
  }

  public void setHeight(double value)
  //sets height equal to value
  {
    height = value;
  }

  public double getWidth()
  //returns width
  {
    return width;
  }

  public double getHeight()
  //returns height
  {
    return height;
  }

  public double doArea()
  //computes and returns area of a rectangle
  {
    return width*height;
  }

  //instance variables
  protected double width, height;
}
```

(b) The *Rectangle* Class

```
//Class for squares. Area can be
//computed from side.
public class Square
{
  public void setSide(double value)
  //sets side equal to value
  {
    side = value;
  }

  public double getSide()
  //returns side
```

FIGURE 27

A Java Program with Polymorphism and Inheritance (continued)

```
  {
    return side;
  }

  public double doArea()
  //computes and returns area of a square
  {
    return side*side;
  }

  //instance variable
  private double side;
}
```

(c) The *Square* Class

```
//Square2 is derived class of Rectangle,
//uses the inherited height and width
//properties and the inherited doArea method

public class Square2 extends Rectangle
{
  public void setSide(double value)
  //sets width and height equal to value
  {
    width = value;
    height = value;
  }
}
```

(d) The *Square2* Class

```
//Computes areas of geometric shapes.
//Uses classes Circle, Rectangle, Square, Square2
public class Geometry
{
  public static void main (String[] args)
  {
    Circle joe = new Circle();
    joe.setRadius(23.5);
    System.out.println("The area of a circle "
       + "with radius " + joe.getRadius()
       + " is " + joe.doArea());

    Rectangle luis = new Rectangle();
    luis.setWidth(12.4);
    luis.setHeight(18.1);
    System.out.println("The area of a rectangle "
       + "with dimensions " + luis.getWidth()
       + " and " + luis.getHeight()
       + " is " + luis.doArea());

    Square anastasia = new Square();
    anastasia.setSide(3);
```

FIGURE 27

A Java Program with Polymorphism and Inheritance (continued)

```
      System.out.println("The area of a square "
         + "with side " + anastasia.getSide()
         + " is " + anastasia.doArea());

      Square2 tyler = new Square2();
      tyler.setSide(4.2);
      System.out.println("The area of a square "
         + "with side " + tyler.getWidth()
         + " is " + tyler.doArea());
   }
}
```

(e) The *Geometry* Class—Main Method

The instance variables for each class represent the properties that any object of the class possesses. A Circle object has a radius property, whereas a Rectangle object has a width property and a height property. A Square object has a side property, as one might expect, but a Square2 object doesn't seem to have any properties, or for that matter any way to compute its area. We'll explain the difference between the *Square* class and the *Square2* class shortly.

The output (rounded to two decimal places for simplicity and wrapped to fit on the page) after running the program in Figure 27 is

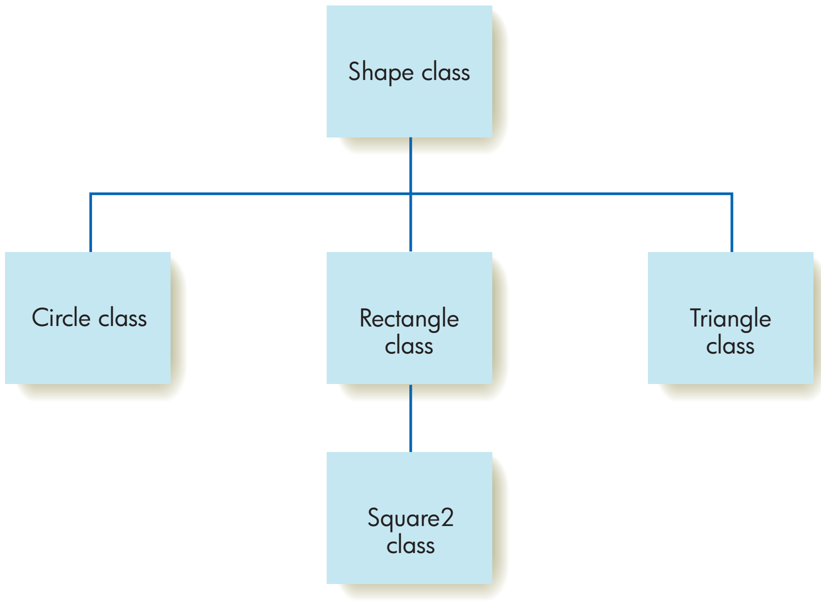
```
The area of a circle with radius 23.5 is 1734.95
The area of a rectangle with dimensions 12.4 and
   18.1 is 224.44
The area of a square with side 3 is 9.0
The area of a square with side 4.2 is 17.64
```

In Figure 27 we see polymorphism at work, because each class has its own *doArea* method. When the program executes, the correct method is used, on the basis of the class to which the object invoking the method belongs. After all, computing the area of a circle is quite different from computing the area of a rectangle. The methods themselves are straightforward; they employ assignment statements to set the dimensions and the usual formulas to compute the area of a circle, rectangle, and square.

Square is a separate class with a *side* property and a *doArea* method. The *Square2* class, however, recognizes the fact that squares are special kinds of rectangles. The *Square2* class is a subclass of the *Rectangle* class, as is indicated by the reference in the header of the *Square2* class that it **extends** the *Rectangle* class. The *Square2* class inherits the *width* and *height* properties from the "parent" *Rectangle* class; the "protected," rather than private, status of these properties in the *Rectangle* class indicates that they can be extended to any subclass. *Square2* also inherits the *setWidth*, *setHeight*, *getWidth*, *getHeight*, and *doArea* methods. In addition, *Square2* has its own method, *setSide*, because setting the value of the "side" makes sense for a square but not for an arbitrary rectangle. What the user of the *Square2* class doesn't know is that there really isn't a "side" property; the *setSide* method merely sets the inherited *width* and *height* properties to the same value. To compute the area, then, the *doArea* method inherited from the *Rectangle* class can be used, and there is no need to redefine it or even to copy the existing code. Here we see inheritance at work.

FIGURE 28

A Hierarchy of Geometric Classes



Inheritance can be carried through multiple "generations." We might redesign the program so that there is one "superclass" that is a general *Shape* class, of which *Circle* and *Rectangle* are subclasses, with *Square2* a subclass of *Rectangle* (see Figure 28 for a possible class hierarchy).

6.4 *What Have We Gained?*

Now that we have some idea of the flavor of object-oriented programming, we should ask what we gain by this approach. There are two major reasons why OOP is a popular way to program:

- Software reuse
- A more natural "worldview"

SOFTWARE REUSE. Manufacturing productivity took a great leap forward when Henry Ford invented the assembly line. Automobiles could be assembled using identical parts so that each car did not have to be treated as a unique creation. Computer scientists are striving to make software development more of an assembly-line operation and less of a handcrafted, start-over-each-time process. Object-oriented programming is a step toward this goal: A useful class that has been implemented and tested becomes a component available for use in future software development. Anyone who wants to write an application program involving circles, for example, can use the already written, tried, and tested *Circle* class and simply create Circle objects as needed. As the "parts list" (the class library) grows, it becomes easier and easier to find a "part" that fits, and less and less time has to be devoted to writing original code. If the objects from a class don't quite fit, perhaps the class can be modified by creating a subclass; this is still less work than starting from scratch. Software reuse implies more than just faster code generation. It also means improvements in *reliability*; these classes have already been tested, and if properly used, they will work correctly. And it means improvements in *maintainability*. Thanks to the encapsulation property of object-oriented programming, changes can be made in the details of class methods without affecting other code, although such changes require retesting the classes.

A MORE NATURAL "WORLDVIEW." The traditional view of programming is procedure-oriented, with a focus on tasks, subtasks, and algorithms. But wait—didn't we talk about subtasks in OOP? Haven't we said that computer science is all about algorithms? Does OOP abandon these ideas? Not at all. It is more a question of *when* these ideas come into play. Object-oriented programming recognizes that in the "real world," tasks are done by entities (objects). Object-oriented program design begins by identifying those objects that are important in the domain of the program because their actions contribute to the mix of activities going on in the banking enterprise, the medical office, or wherever. Then it is determined what data should be associated with each object and what subtasks the object contributes to this mix. Finally, an algorithm to carry out each subtask must be designed.

Object-oriented programming is an approach that allows the programmer to more closely model or simulate the world as we see it, rather than mimicking the sequential actions of the Von Neumann machine. It provides another buffer between the real world and the machine, another level of abstraction in which the programmer can create a virtual problem solution that is ultimately translated into electronic signals on hardware circuitry.

Finally, we should mention that a graphical user interface, with its windows, icons, buttons, and menus, is an example of object-oriented programming at work. A general button class, for example, can have properties of height, width, location on the screen, text that may appear on the button, and so forth. Each individual button object has specific values for those properties. The button class can perform certain services by responding to messages, which are generated by events (for example, the user clicking the mouse on a button triggers a "mouse-click" event). Each particular button object individualizes the code to respond to these messages in unique ways. We will not go into details of how to develop graphical user interfaces in Java, but in the next section you will see a bit of the programming mechanics that can be used to draw the graphics items that make up a visual interface.

PRACTICE PROBLEMS

1. What is the output from the following section of code if it is added to the main method of the Java program in Figure 27?

```
Square one = new Square();
one.setSide(10);
System.out.println("The area of a square "
    + "with side " + one.getSide()
    + " is " + one.doArea());
```

2. In the *Shape* hierarchy described in this section, suppose that the *Triangle* class includes a *doArea* method. What two properties should any triangle object have?

7 Graphical Programming

The programs that we have looked at so far all produce *text output*—output composed of the characters {A . . . Z, a . . . z, 0 . . . 9} along with a few punctuation marks. For the first 30 to 35 years of software development, text was virtually the only method of displaying results in human-readable form, and in those early days it was quite common for programs to produce huge stacks of alphanumeric output. These days an alternative form of output—*graphics*—has become much more widely used. With graphics, we are no longer limited to 100 or so printable characters; instead, programmers are free to construct whatever shapes and images they desire.

The intelligent and well-planned use of graphical output can produce some phenomenal improvements in software. We discussed this issue in Chapter 6, where we described the move away from the text-oriented operating systems of the 1970s and 1980s, such as MS-DOS and VMS, to operating systems with more powerful and user-friendly graphical user interfaces (GUIs), such as Windows 7, Windows 8 and Mac OS X. Instead of requiring users to learn dozens of complex text-oriented commands for such things as copying, editing, deleting, moving, and printing files, GUIs can present users with simple and easy to understand visual metaphors for these operations. In Figure 29a, the operating system presents the user with icons for printing, deleting, or copying a file. In Figure 29b, dragging a file to the printer icon prints the file.

Not only does graphics make it easier to manage the tasks of the operating system, it can help us visualize and make sense of massive amounts of output produced by programs that model complex physical, social, and mathematical systems. (We will discuss modeling and visualization in Chapter 13.) Finally, there are many applications of computers that would simply be impossible without the ability to display output visually. Applications such as virtual reality, computer-aided design/computer-aided manufacturing (CAD/CAM), games and entertainment, medical imaging, and computer mapping would not be anywhere near as important as they are without the enormous improvements that have occurred in the areas of graphics and visualization.

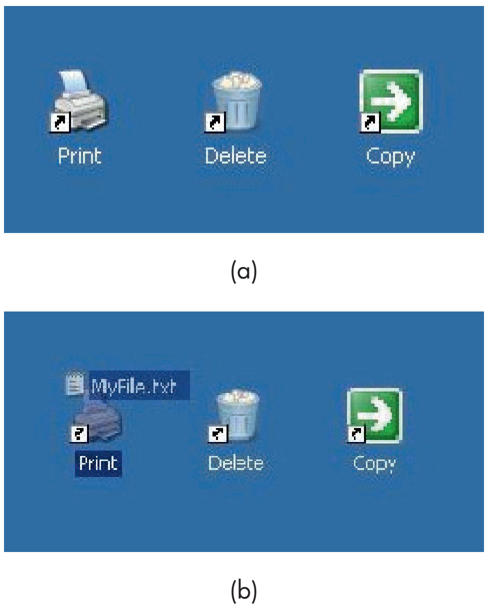


FIGURE 29

An Example of the Use of Graphics to Simplify Machine Operation

So, we know that graphical programming is important. The question is, What features must be added to a programming language like Java to produce graphical output?

7.1 *Graphics Hardware*

Modern computer terminals use what is called a **bitmapped display**, in which the screen is made up of thousands of individual picture elements, or **pixels**, laid out in a two-dimensional grid. These are the same pixels used in visual images, as discussed in Chapter 4. In fact, the display is simply one large visual image. The number of pixels on the screen varies from system to system; typical values range from 800 × 600 up to 1560 ×1280. Terminals with a high density of pixels are called **high-resolution** terminals. The higher the resolution—that is, the more pixels available in a given amount of space—the sharper the visual image because each individual pixel is smaller. However, if the screen size itself is small, then a high resolution image can be too tiny to read. A 30″ wide-screen monitor might support a resolution of 2560 × 1600, but that would not be suitable for a laptop screen. In Chapter 4 you learned that a color display requires 24 bits per pixel, with 8 bits used to represent the value of each of the three colors red, green, and blue. The memory that stores the actual screen image is called a **frame buffer**. A high-resolution color display might need a frame buffer with (1560 × 1280) pixels × 24 bits/pixel = 47,923,000 bits, or about 6 MB, of memory for a single image. (One of the problems with graphics is that it requires many times the amount of memory needed for storing text.)

The individual pixels in the display are addressed using a two-dimensional coordinate grid system, the pixel in the upper-left corner being (0, 0). The overall pixel-numbering system is summarized in Figure 30. The specific values for *maxX* and *maxY* in Figure 30 are, as mentioned earlier, system-dependent. (Note that this coordinate system is not the usual mathematical one. Here the origin is in the upper-left corner, and *y* values are measured downward.)

FIGURE 30

Pixel-Numbering System in a Bitmapped Display

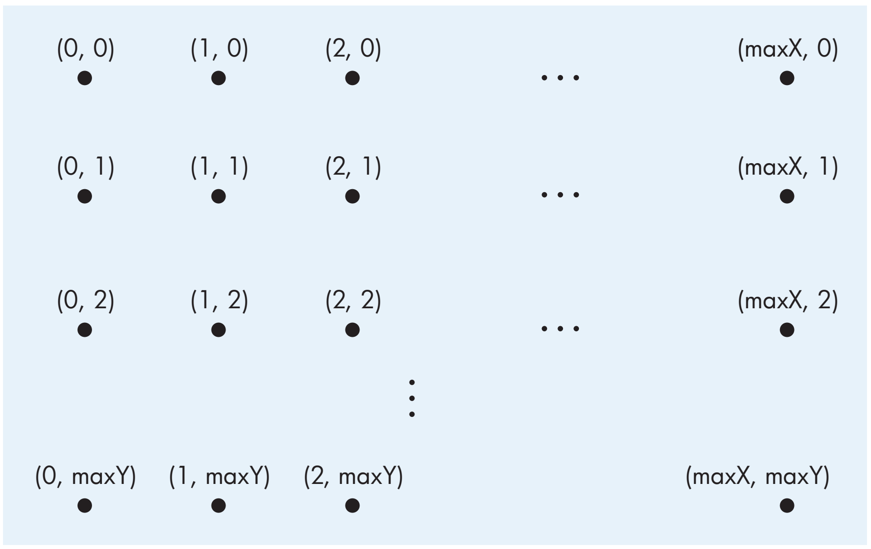
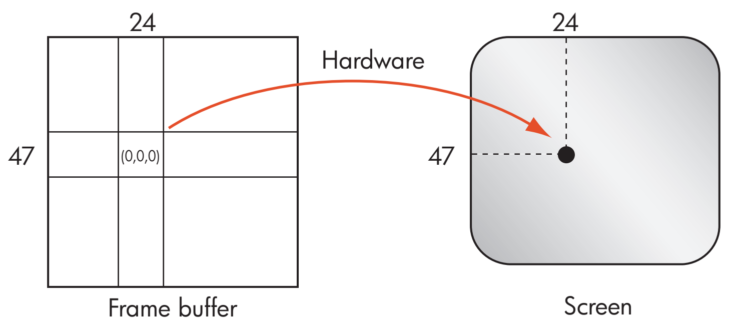


FIGURE 31

Display of Information on the Terminal



The terminal hardware displays on the screen the frame buffer value of every individual pixel. For example, if the frame buffer value on a color monitor for position (24, 47) is RGB (0, 0, 0), the hardware sets the color of the pixel located at column 24, row 47 to black, as shown in Figure 31. The operation diagrammed in Figure 31 must be repeated for all of the 500,000 to 2 million pixels on the screen. However, the setting of a pixel is not permanent; on the contrary, its color and intensity fade quickly. Therefore, each pixel must be "repainted" often enough so that our eyes do not detect any "flicker," or change in intensity. This requires the screen to be completely updated, or refreshed, 30–50 times per second. By setting various sequences of pixels to different colors, the user can have the screen display any desired shape or image. This is the fundamental way in which graphical output is achieved.

7.2 *Graphics Software*

To control the setting and clearing of pixels, programmers use a collection of software modules that are part of a special package called a **graphics library**. Virtually all modern programming languages, including Java, come with an extensive and powerful graphics library for creating a wide range of shapes and images. Typically, an "industrial strength" graphics library includes dozens or hundreds of modules for everything from drawing simple geometric shapes like lines and circles, to creating and selecting colors, to more complex operations such as displaying scrolling windows, pull-down menus, and buttons.

The Java library includes a package called the **Abstract Windowing Toolkit**, usually abbreviated **AWT**. This toolkit contains dozens of methods that allow users to create powerful interfaces. The AWT package includes methods for

- Creating the basic set of GUI components, including windows, buttons, text boxes, icons, and menus
- Allowing the user to control the size and placement of these components
- Allowing the user to define special objects called **listeners** that automatically activate methods when screen events, such as moving or clicking the mouse, or selecting a menu item, occur

In 1998, Sun introduced a package of even more powerful GUI components, commonly called **Swing** components. Both the AWT package and the Swing package are huge, and their descriptions are well beyond the scope of this text. In this discussion we restrict our focus to a modest subset of methods. Although the set is small, the graphics methods we present give you a good idea of what visual programming is like, and will allow you to display some interesting, nontrivial images on the screen. (To access these Java graphics methods, you must include the statement

```
import java.awt.*
```

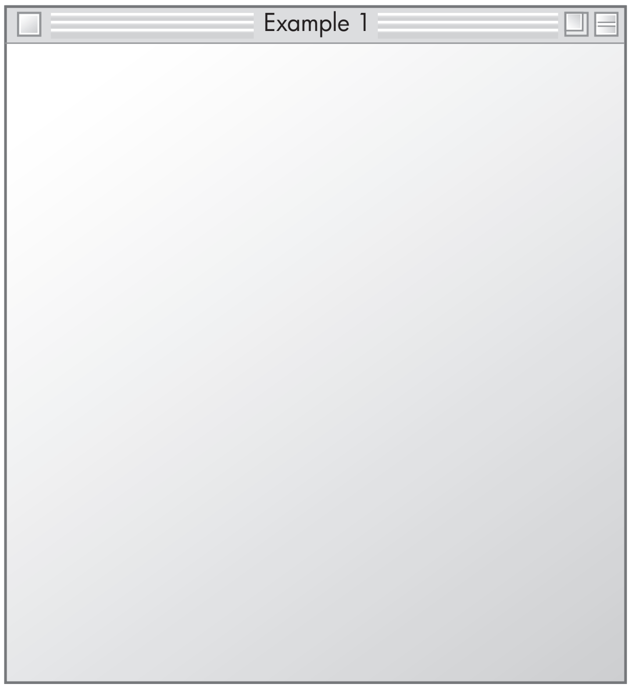
at the beginning of your program. This makes the classes in the Java AWT available for your program to use.)

In Java, the primary means of drawing geometric shapes is with the class called *Graphics*. Every open window contains an instance of this class called its **graphics context**, which by tradition is represented by the letter *g*. When you open a window you cannot see this "graphics context object," but it is there, and it responds to messages (i.e., method invocations) that ask it to display various shapes and patterns within the window. It carries out this drawing operation using the identical bitmapped techniques described previously.

To create a new window for our drawings (called a **frame** in Java), we can use the following sequence of three commands:

```
Frame f = new Frame("Example 1");
f.setSize(500, 500);
f.setVisible(true);
```

The first line creates a new window called *f* containing the label "Example 1" in the title bar at the top of the window. The second line sets the size of this window at 500 pixels × 500 pixels. When setting the window size, be sure that you do not exceed the maximum value allowed on your system. The last line makes the window visible on the screen. After executing these three lines, your screen should display the following:



Now, to obtain the graphics context of the window *f* created above, we use the method called *getGraphics*. This method returns the graphics context of the window to which the message is sent and assigns it to the *Graphics* object *g*:

```
Graphics g;
g = f.getGraphics();
```

We can now send whatever drawing commands we want to *g*, and it will display the shapes that we ask for inside window *f*.

What types of messages does *g* respond to? There are literally dozens of drawing commands in the *Graphics* library that allow you to (1) draw geometric shapes (e.g., lines, rectangles, ovals, polygons); (2) set, change, and define colors; (3) fill in or shade objects; (4) create text in a range of fonts and sizes; and (5) produce many different types of graphs and charts. There are far too many methods to discuss here; instead, we introduce a few of the most important and most basic methods, to give you an idea of the type of graphic operations available in Java. You will have a chance to use these operations in the exercises at the end of this module.

1. *drawLine(int x1, int y1, int x2, int y2)*. This draws a straight line from point (*x1*, *y1*) on the screen (measured in pixels) to point (*x2*, *y2*). Thus the operation

   ```
   g.drawLine(100, 100, 200, 200);
   ```

   would produce something like the following image:

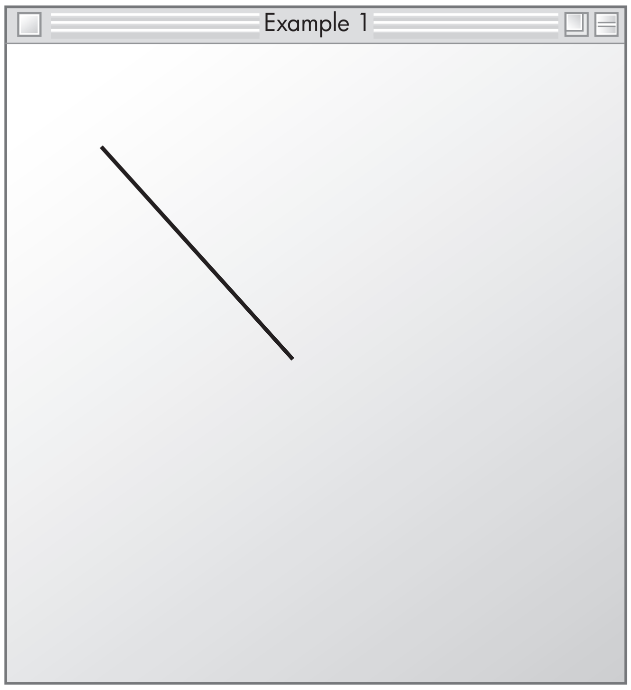


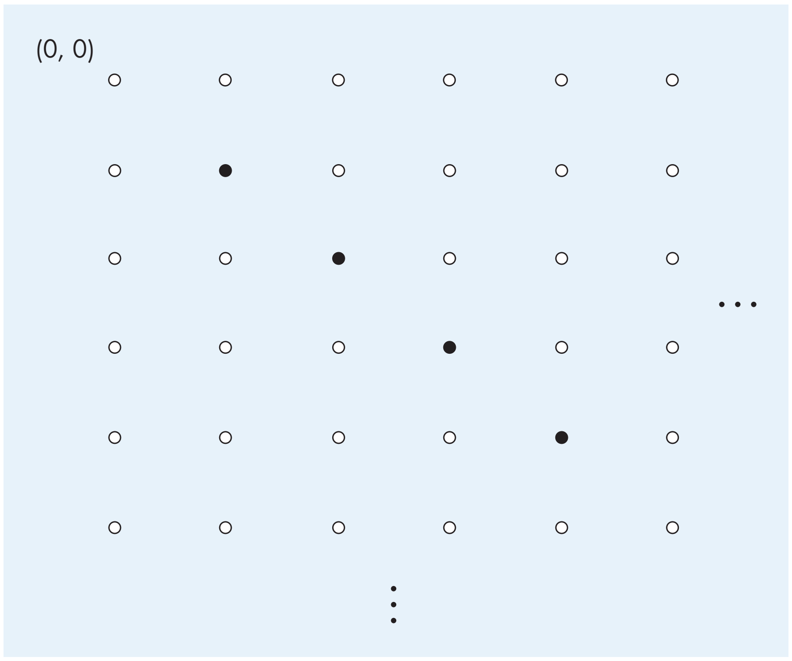
FIGURE 32

Complete Java Program for Drawing a Line

```
import java.awt.*;
import java.util.*;
public class Line
{
    public static void main(String[] args)
     {
        double response = 0;
        Scanner inp = new Scanner(System.in);

        Frame f = new Frame("Example 1");
        f.setSize(500, 500);
        f.setVisible(true);

        Graphics g;
        g = f.getGraphics();

        while (response < 10000)
        {
            g.drawLine(100, 100, 200, 200);
            response = response + 1;
        }

        System.out.print("Type 1 to exit: ");
        response = inp.nextDouble();
        System.exit(0);
     }
}
```

On your system, the exact location and length of the line may be slightly different because of minor differences in screen resolution. You may also need to write some code to hold the image on the screen. Figure 32 shows a complete program to draw the line and repaint the image on the screen a fixed number of times; user input closes the program.

What actually happens internally when you execute a *drawLine* command? The terminal hardware determines (using some simple geometry and trigonometry) exactly which pixels on the screen must be "turned on" (i.e., set to the current value of the drawing color) to draw a straight line between the specified coordinates. For example, if the drawing color is black, then the command *drawLine(1, 1, 4, 4)* causes the following four pixels in the frame buffer to be set to the RGB value (0, 0, 0).

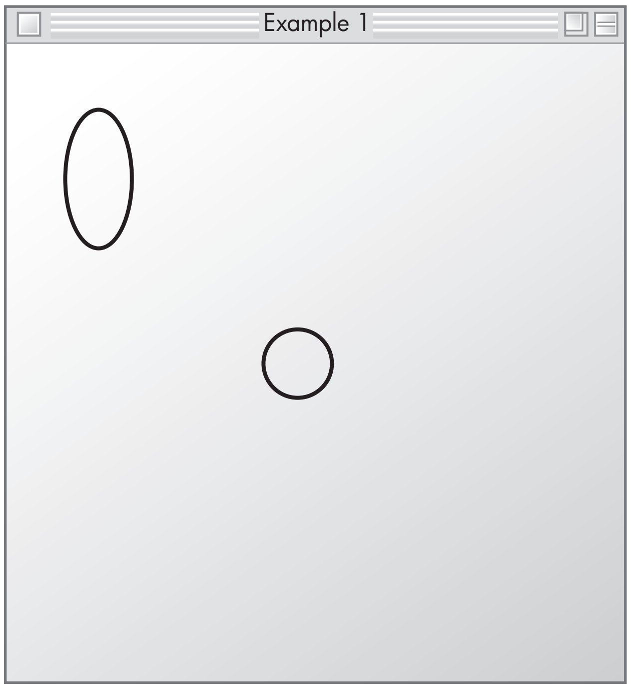


Now, when the hardware draws the frame buffer on the screen, these four pixels are colored black. Because pixels are only about 1/100th of an inch apart, our eyes do not perceive four individual black dots but an unbroken line segment.

2. *drawOval(int x, int y, int width, int height)*. This operation draws an oval that fits within a rectangle whose upper-left corner is located at (*x*, *y*) and whose dimensions are the specified width and height. If the width and height values are the same, you produce a circle. Thus, the following two commands

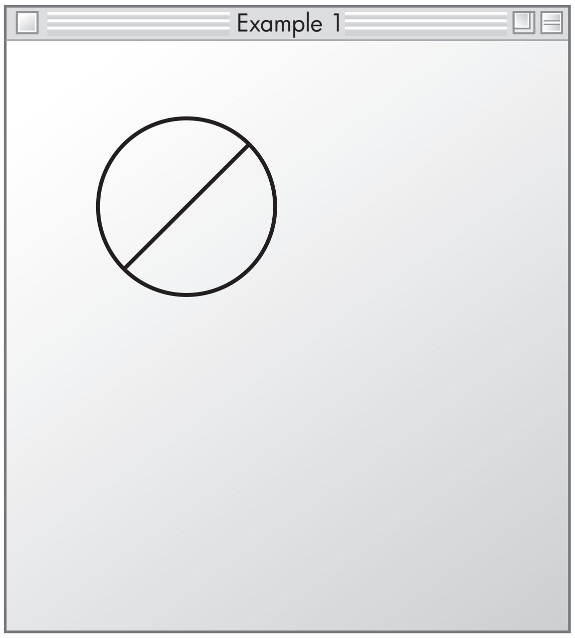
```
g.drawOval(50, 50, 20, 100);
g.drawOval(200, 200, 30, 30);
```

produce the following image:



Now, using the two commands we have just introduced—*drawLine* and *drawOval*—we can produce an image of the well-known international traffic sign for No Entry:

```
g.drawOval(100, 100, 80, 80);
g.drawLine(168, 112, 112, 168);
```

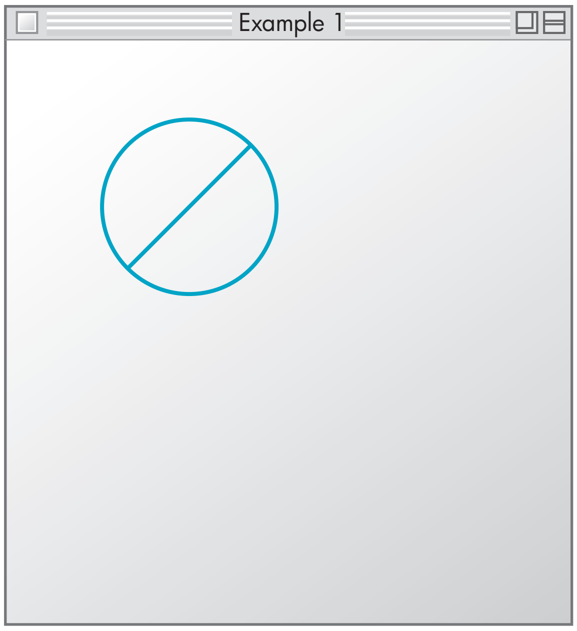


However, we also know that this No Entry sign sometimes appears in either blue or red rather than black. Java allows us to control the drawing color using the *setColor* method from the *Color* class.

3. *setColor(Color c)*. This method allows us to set our drawing color to any one of the following 12 preset colors: red, yellow, blue, orange, pink, cyan, magenta, black, white, gray, lightGray, and darkGray. (Java also lets you define totally new colors based on the intensities of their red, green, and blue components. We won't discuss that feature here.) Using this new method, we can redraw our traffic sign in blue as follows:

```
g.setColor(Color.blue);
g.drawOval(100, 100, 80, 80);
g.drawLine(168, 112, 112, 168);
```

which produces the next image. Note that executing the operation *setColor* does not change the color of images already drawn, only the color of any new images subsequently placed on the screen.

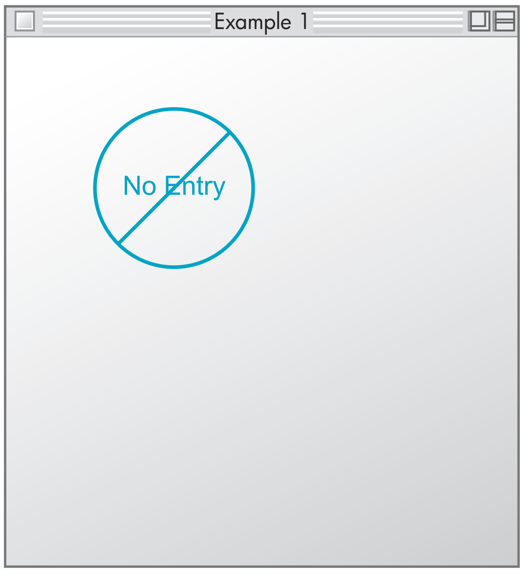


The last thing we might want to add to our traffic sign is the phrase "No Entry". Java uses the method *drawString* to put text into a drawing:

4. *drawString(String str, int x, int y)*. This method writes the string *str* into the image. The lower-left position of the first character of the string is placed at position (*x*, *y*). Thus, to put the desired text into the drawing, we could write the following four lines:

```
g.setColor(Color.blue);
g.drawOval(100, 100, 80, 80);
g.drawString("No Entry", 112, 145);
g.drawLine(168, 112, 112, 168);
```

which produce the desired image:



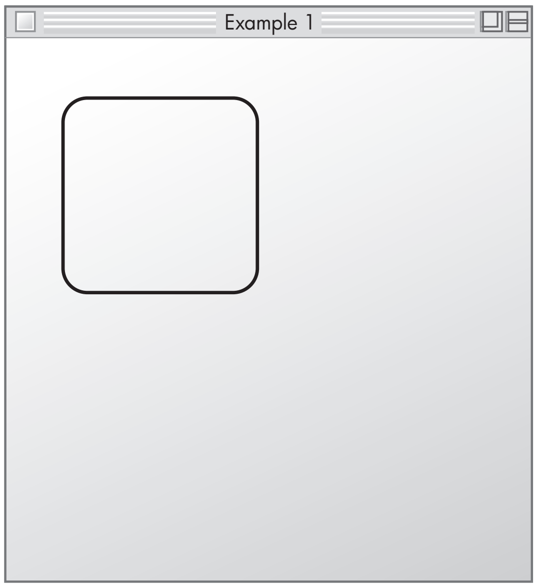
(There are a number of Java methods for controlling font size and font type, but we will not mention them here.)

There are other drawing commands to produce a variety of interesting shapes:

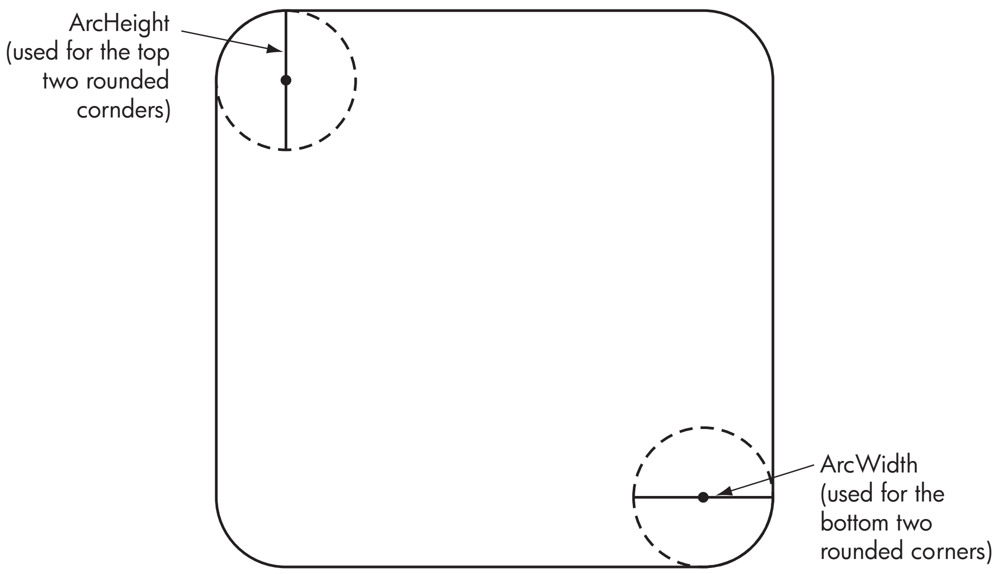
5. *drawRect(int x, int y, int width, int height)*. This method draws a rectangle whose upper-left corner is at position (*x*, *y*) and whose dimensions are the specified height and width.
6. *drawRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)*. This method draws a rectangle with smoothly rounded corners. For example, the command

```
g.drawRoundRect(10, 10, 100, 100, 50, 50);
```

produces:



The parameters *arcWidth* and *arcHeight* set the diameter of the circles whose arcs are used to form the rounded edges of the rectangle, as shown in the next diagram.



Sometimes we want to produce a filled shape, rather than just an outline. Java has a number of methods to draw shapes whose insides are filled using the currently declared drawing color:

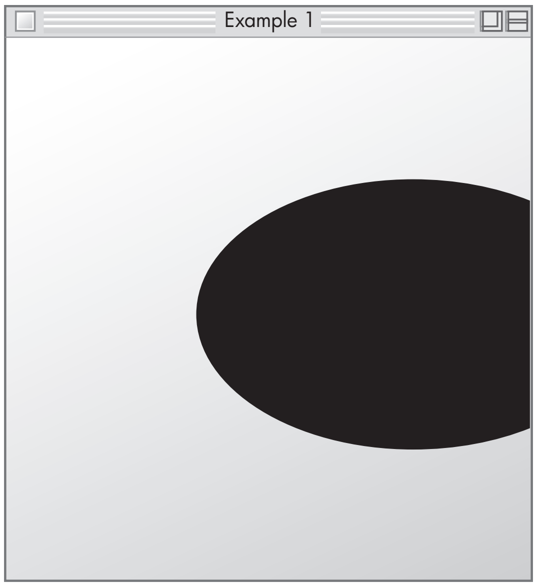
7. *fillRect(int x, int y, int width, int height)*
8. *fillRoundRect(int x, int y, int width, int height, int arcWidth, int arcHeight)*
9. *fillOval(int x, int y, int height, int width)*

All three of the above commands draw the specified shape with its insides filled with whatever color you have specified. (*Note*: If you have not specified a drawing color, then the shape is filled with the default color, which is usually black.)

If we define a screen of size 300 × 300, then the following command:

```
g.fillOval(80, 80, 300, 200);
```

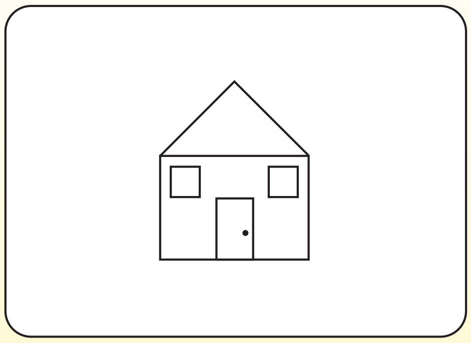
produces the next image. Notice that the portions of the circle beyond the edge of the window are discarded, an operation called **clipping**. All methods in the graphics library clip those parts of an image that lie outside its window boundaries.



This has been a brief introduction to the topic of graphics software. As mentioned earlier, the number of methods in the Java *Graphics* class—or in any large-scale production graphics package—is much, much larger. However, the nine operations we have introduced are sufficient to allow you to produce some interesting images and, even more important, give you an appreciation for how visually oriented software is developed.

PRACTICE PROBLEM

Write the sequence of commands to draw the following "house" on the graphics window:



Create the house using four rectangles (for the base of the house, the door, and the two windows), two line segments (for the roof), and one filled circle (for the doorknob). Locate the house anywhere you want on the window.

8 Conclusion

In this module we looked at one representative high-level programming language, Java. Of course, there is much about this language that has been left unsaid, but we have seen how the use of a high-level language overcomes many of the disadvantages of assembly language programming, creating a more comfortable and useful environment for the programmer. In a high-level language, the programmer need not manage the storage or movement of data values in memory. The programmer can think about the problem at a higher level, can use program instructions that are both more powerful and more natural language–like, and can write a program that is much more portable among various hardware platforms. We also saw how modularization, through the use of methods and parameters, allows the program to be more cleanly structured and how object-oriented programming allows a more intuitive view of the problem solution and provides the possibility for reuse of helpful classes. We even had a glimpse of graphical programming.

Java is not the only high-level language. You might be interested in looking at the other online language modules for languages similar to Java (C++, C#, Python, and Ada). Still other languages take quite a different approach to problem solving. In Chapter 10 of *Invitation to Computer Science*, we look at some other languages and language approaches and also address the question of why there are so many different programming languages.

EXERCISES

1. Write a Java declaration for one real number quantity called *rate*, initialized to 5.0.
2. Write a single Java statement to declare two integer quantities called *orderOne* and *orderTwo*, each initialized to 0.
3. A Java main method needs one character variable *choice*, one integer variable *inventory*, and one real number variable *sales*. Write the necessary declarations; initialize *choice* to the blank character and the other values to zero.
4. a. Write a Java output statement to print the value of *PI* supplied by the Math library.

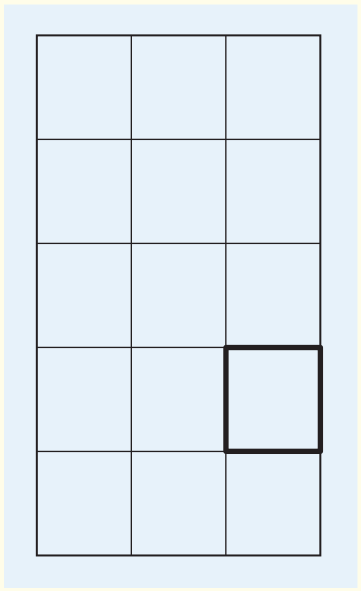
   b. Write a Java declaration for a constant quantity to be called *EVAPORATION_RATE*, which is to have the value 6.15.
5. You want to write a Java program to compute the average of three quiz grades for a single student. Decide what variables your program needs, and write the necessary declarations.
6. Given the declaration

   ```
   int[ ] list = new int[10];
   ```

   how do you refer to the eighth number in the array?
7. An array declaration such as

   ```
   int[ ][ ] table = new int[5][3];
   ```

   represents a two-dimensional table of values with 5 rows and 3 columns. Rows and columns are numbered in Java starting at 0, not at 1. Given this declaration, how do you refer to the marked cell that follows?



8. Write Java statements to prompt for and collect values for the time in hours and minutes (two integer quantities). Assume the declarations

   ```
   int hours = 0, minutes = 0;
   Scanner inp = new Scanner(System.in);
   ```

   have already been made.
9. A program computes two integer quantities *inventoryNumber* and *numberOrdered*. Write a single output statement that prints these two quantities along with appropriate text information.
10. The integer quantities *age* and *weight* currently have the values 32 and 187, respectively. Write the exact output generated by the following statement:

    ```
    System.out.println("Your age is"
        + age + "and your weight is" +
        weight + ".");
    ```
11. Output that is a real number can be formatted so that the number is rounded to a specified number of decimal places. To do this, add the following statement at the very beginning of the program

    ```
    import java.text.*;
    ```

    and add the following at the beginning of the main method body:

    ```
    DecimalFormat p = new
        DecimalFormat("0.00");
    ```

    where the desired format for the output is given in quotes—for example, “0.00” is requesting that the output be rounded to two decimal digits. Output statements are then modified as follows (from the SportsWorld program):

    ```
    System.out.println("The " +
        "circumference for a pool" +
        " of radius " + radius + " is "
        + p.format(circumference));
    ```

    Write Java formatting and output statements to generate the following output, assuming that *density* is a type *double* variable with the value 63.78:

    ```
    The current density is 63.8, to
    within one decimal place.
    ```
12. What is the output after the following sequence of statements is executed? (Assume the integer variables *a* and *b* have been declared.)

    ```
    a = 12;
    b = 20;
    b = b + 1;
    a = a + b;
    System.out.println(2*a);
    ```

13. Write a Java main method that gets the length and width of a rectangle from the user and computes and writes out the area.

14. a. In the SportsWorld program of Figure 14, the user must respond with "C" to choose the circumference task. In such a situation, it is preferable to accept either uppercase or lowercase letters. Rewrite the condition in the program to allow this.

b. In the SportsWorld program, rewrite the condition for continuation of the program to allow either an uppercase or a lowercase response.

15. Write a Java main method that gets a single character from the user and writes out a congratulatory message if the character is a vowel (a, e, i, o, or u), but otherwise writes out a "You lose, better luck next time" message.

16. Insert the missing line of code so that the following adds the integers from 1 to 10, inclusive.

```
value = 0;
top = 10;
score = 1;
while (score <= top)
{
    value = value + score;
    - - - - //the missing line
}
```

17. What is the output after the following main method is executed?

```
public static void main(String[]
args)
{
   int low = 1;
   int high = 20;
   while (low < high)
   {
     System.out.println(low
        + " " + high);
     low = low + 1;
     high = high - 1;
   }
}
```

18. Write a Java main method that outputs the even integers from 2 through 30, one per line. Use a while loop.

19. In a while loop, the Boolean condition that tests for loop continuation is done at the top of the loop, before each iteration of the loop body. As a consequence, the loop body might not be executed at all. Our pseudocode language of Chapter 2 contains a do-while loop construction, in which a test for loop termination occurs at the bottom of the loop rather than at the top, so that the loop body always executes at least once. Java has a **do-while** statement that tests for loop continuation at the bottom of the loop. The form of the statement is

```
do
   S1;
while (Boolean condition);
```

where, as usual, S1 can be a compound statement. Write a Java main method to add up a number of nonnegative integers that the user supplies and to write out the total. Use a negative value as a sentinel, and assume that the first value is nonnegative. Use a do-while statement.

20. Write a Java program that asks for a duration of time in hours and minutes and writes out the duration only in minutes.

21. Write a Java program that asks for the user's age in years; if the user is under 35, then quote an insurance rate of $2.23 per $100 for life insurance; otherwise, quote a rate of $4.32.

22. Write a Java program that reads integer values until a 0 value is encountered, then writes out the sum of the positive values read and the sum of the negative values read.

23. Write a Java program that reads in a series of positive integers and writes out the product of all the integers less than 25 and the sum of all the integers greater than or equal to 25. Use 0 as a sentinel value.

24. a. Write a Java program that reads in 10 integer quiz grades and computes the average grade. (*Hint*: Remember the peculiarity of integer division.)

b. Write a Java program that asks the user for the number of quiz grades, reads them in, and computes the average grade.

25. Write a void Java method that receives two integer arguments and writes out their sum and their product.

26. Write a nonvoid Java method that receives a real number argument representing the sales amount for videos rented so far this month. The method asks the user for the number of videos rented today and returns the updated sales figure to the main method. All videos rent for $4.25.

27. Write a nonvoid Java method that receives three integer arguments and returns the maximum of the three values.

28. a. Write a Java *doPerimeter* method for the *Rectangle* class of Figure 27.

b. Write Java code that creates a new *Rectangle* object called *yuri*, then writes out information about this object and its perimeter using the *doPerimeter* method from part (a).

29. Draw a class hierarchy diagram similar to Figure 28 for the following classes: *Student*, *Undergraduate_Student*, *Graduate_Student*, *Sophomore*, *Senior*, *PhD_Student*.

30. Imagine that you are writing a program using an object-oriented programming language. Your program will be used to maintain records for a real estate office. Decide on one class in your program and a service that objects of that class might provide.

31. Write a Java program to balance a checkbook. The main method of the *CheckbookApp* class should get the initial balance from the user, allow the user to process as many transactions as desired, and write the final balance. The *Checkbook* class should contain two public static methods to handle deposits and checks, respectively. Each method should collect and write out the amount of the transaction, and compute, write out, and return the new balance. (See Exercise 11 on how to format output to two decimal places, as is usually done with monetary values.)

32. Write a Java program to compute the cost of carpeting three rooms. *Room* objects have dimensions of width and length, and they can compute and return their area and (given the price per square unit) the cost to carpet themselves. The main method of the *RoomApp* class should create a *Room* object and use a loop to process each of three rooms: get the dimensions and carpet price, write out the individual areas and costs, add the three costs, then write out the total cost. (See Exercise 11 on how to format output to two decimal places, as is usually done with monetary values.)

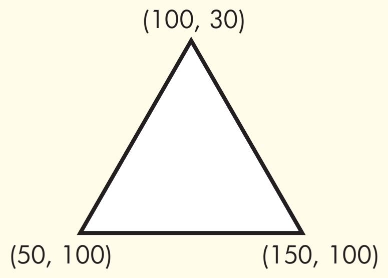
33. Determine the resolution on the screen on your computer (ask your instructor or the local computer center how to do this). Using this information, determine how many bytes of memory are required for the frame buffer to store the following:

a. A black-and-white image (1 bit per pixel)

b. A grayscale image (8 bits per pixel)

c. A color image (24 bits per pixel)

34. Using the *drawLine* command described in Section 7.2, draw an isosceles triangle with the following configuration:

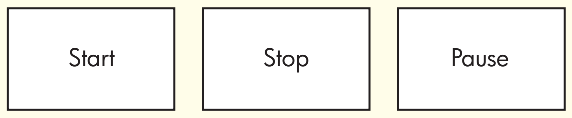


35. Discuss what problem the display hardware might encounter while attempting to execute the following operations, and describe how this problem could be solved.

```
drawLine(1, 1, 4, 5);
```

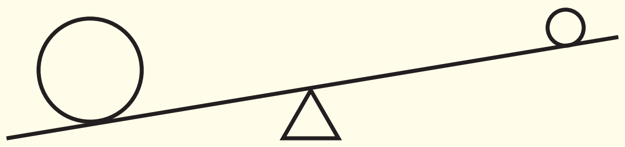
36. Draw a square with sides 100 pixels in length. Then inscribe a circle of radius 50 inside the square. Position the square so its upper-left corner is at position (60, 100).

37. Create the following three labeled rectangular buttons:



Have the space between the Start and Stop buttons be the same as the space between the Stop and Pause buttons.

38. Create the following image of a "teeter-totter":



ANSWERS TO PRACTICE PROBLEMS

Section 2

1. the first three

   `martinBradley` (camel case)

   `C3P_OH` (although best not to use the underscore character)

   `Amy3` (Pascal case)

   `3Right` (not acceptable, begins with digit)

   `double` (not acceptable, Java keyword)

2. `int Number;`
3. `final double TAX_RATE = 5.5;`
4. `hits[7]`

Section 3.1

1.
```
System.out.print("Enter quantity as an integer: ");
quantity = inp.nextInt();
```
2.
```
System.out.println("The average high temperature "
    + "in San Diego for the month of May is "
    + average);
```
3.
```
This isgoodbye, Steve
```

Section 3.2

1. `next = newNumber;`
2. 55.0

Section 3.3

1. 30
2.
```
3
5
7
9
11
13
15
17
19
21
```
3. Yes
4. 6
5.
```
if (night==day)
  System.out.println("Equal");
```

Section 4

1.

```
//program to read in and write out
//user's first and last initials

import java.util.*;
public class Practice
{
   public static void main(String[] args)
   {
      String response = " ";
      char initial1, initial2;
      Scanner inp = new Scanner(System.in);

      System.out.print("Enter your first initial: ");
      response = inp.next();
      initial1 = response.charAt(0);
      System.out.print("Enter your last initial: ");
      response = inp.next();
      initial2 = response.charAt(0);

      System.out.println("Your initials are "
         + initial1 + initial2);
   }
}
```

2.

```
//program to compute cost based on price per item
//and quantity purchased

import java.util.*;
public class Cost
{
   public static void main(String[] args)
   {
      double price = 0.0, cost = 0.0;
      int quantity = 0;
      Scanner inp = new Scanner(System.in);

      System.out.println("What is the price of the item? ");
      price = inp.nextDouble();
      System.out.println("How many of this item are "
         + "being purchased? ");
      quantity = inp.nextInt();
      cost = price*quantity;
      System.out.println("The total cost for this "
         + "item is $" + cost);
   }
}
```

3.

```
//program to test a number relative to 5
//and write out the number or its double

import java.util.*;
public class FiveTester
{
   public static void main(String[] args)
   {
       int number = 0;
       Scanner inp = new Scanner(System.in);

       System.out.print("Enter a number: ");
       number = inp.nextInt();
       if (number < 5)
           System.out.println("The number is " + number);
       else
           System.out.println("Twice the number is "
              + 2*number);
   }
}
```

4.

```
//program to collect a number, then write all
//the values from 1 to that number

import java.util.*;
public class Counter
{
   public static void main(String[] args)
   {
     int number = 0;           //number user enters
     int counter = 1;          //counter to control loop
     Scanner inp = new Scanner(System.in);

     System.out.print("Enter a positive number: ");
     number = inp.nextInt();
     while (counter <= number)
     {
        System.out.println(counter);
        counter = counter + 1;
     }
   }
}
```

Section 5.3

1. 20
2. 20
3. 7

   14

   10 (*number* was passed by value, it cannot be permanently changed by *doIt*)
4. a. `public static double tax(double subtotal)`

   b. `return subtotal * 0.55;`

   c.
   ```
   System.out.println("The tax is "
               + Sales.tax(subtotal));
   ```

Section 6.4

1. `The area of a square with side 10 is 100.0`
2. Height and Base

Section 7.2

```
g.drawRect(150, 150, 250, 200);
g.drawRect(170, 170, 40, 40);
g.drawRect(340, 170, 40, 40);
g.drawRect(250, 270, 50, 80);
g.drawLine(150, 150, 275, 75);
g.drawLine(275, 75, 400, 150);
g.fillOval(290, 310, 5, 5);
```