CHAPTER 1

Two Simple Java Programs

OBJECTIVES

- Understand the nature of programming
- Understand basic computer organization
- Edit, compile, and run a simple Java program
- Edit, compile, and run a Java program that creates objects
- Understand encapsulation and information hiding

1.1 Fundamental Problem with Computers

Once upon a time, an engineer was assigned the task of designing a bridge. Rather than design the bridge himself, he programmed a computer to do it. Unfortunately, he programmed the computer incorrectly. As a result, the computer produced a design that was grossly (and obviously) incorrect: It called for beams that were only 1 inch in diameter. When he showed the design to his boss, the boss immediately recognized the error and said, "This couldn't be right." The engineer responded by saying, "But this design was produced by the computer." "Oh," the boss replied, "in that case, it must be right."

Our engineer and his boss thought the design *had* to be correct because a computer produced it. However, computers are not infallible super-brains. To the contrary, computers are quite dumb. They are electronic machines that simply follow the instructions they are provided. When you program a computer, you provide it with these instructions. If you give a computer incorrect instructions, it will blindly follow them.

Obviously, if we program a computer, we should be sure to give it the correct instructions. But this goal is not easy to accomplish. The fundamental problem with computers is that they require programming, and programming is so difficult to do correctly. However, there is a plus side to programming: It gives computers their incredible flexibility. A toaster can only toast, but with the proper programming, a computer can do almost anything. For example, it can design bridges, play chess, and even diagnose rare diseases (who needs Dr. House?).

Computers often make mistakes because the programmers fail to program them to handle all the possibilities that can occur in real life. For example, many years ago the IRS sent a letter to a taxpayer asking for an explanation for an unauthorized name change. The letter—computer generated, no doubt—wanted to know why Mr. Dos Reis had changed his name to "DosReis" without going through the proper legal procedures. Mr. Dos Reis had not changed his name. Apparently, on the return in question, his name was entered into the computer *without* a space, but the IRS had his name from previous returns *with* a space. The resulting mismatch of names caused a computer to generate the letter of inquiry. The computer's program failed to check if the mismatch was simply due to a missing space rather than a genuine name change.

1.2 Computer Organization

All the information within a computer is stored in the form of binary numbers. **Binary numbers** consist of 0's and 1's. We call symbols that make up binary numbers—the 0's and 1's—**bits**.

Binary numbers work very much like decimal numbers. Consider the decimal number 235. Each digit has a weight, depending on its position (see Fig. 1.1a). Weights from right

to left are the successive powers of 10, starting with $10^0 = 1$. Thus, the digits from right to left in 235 have the weights $10^0 = 1$, $10^1 = 10$, and $10^2 = 100$. Each digit contributes its own value times its weight to the value of the number. For example, the weight of the 3 digit in 235 is 10. Thus, this 3 digit contributes $3 \times 10 = 30$ to the value of the number.

Now consider the binary number 111. The weights from right to left are the successive powers of 2 starting from $2^0 = 1$. The weights of the three bits in 111 from right to left are $2^0 = 1$, $2^1 = 2$, and $2^2 = 4$. Thus, the bits from right to left contribute 1×1, 1×2, and 1×4 to the value of the number (see Fig. 1.1b). Thus, 111 in binary numbers equals $1 + 2 + 4 = 7$ in decimal numbers.

The value of a binary number is simply the sum of the weights corresponding to the 1 bits. For example, the value in decimal of the binary number 101 is 5, which is the sum of 1 (the weight of the right 1) and 4 (the weight of the left 1).

Computers use binary representation because devices that store binary numbers are inexpensive and reliable. Each bit in a binary number requires only a simple off/on switch. The off position can represent 0; the on position can represent 1. In contrast, each position in a decimal number has 10 possibilities: 0, 1, . . . , 9. Thus, each position in a decimal number requires a 10-position switch, which, of course, would be more complicated and expensive than the simple 2-position switch required by binary. Moreover, binary numbers are not restrictive in any way. They, of course, can represent numbers just as well as decimal. But, with the proper encoding, they can also represent letters, punctuation, pictures, sound, movies, and so on. For example, if we assign a distinct binary code to each letter of the alphabet, we can then use these codes within a computer to represent their corresponding letters.

A computer system consists of **software** (the programs and data) and **hardware** (the circuits, wires, cabinets—anything you can touch). The software consists of two types: **system software** (software that manages the computer system itself) and **application**

a) Decimal

2	3	5	digits
10^2	10^1	10^0	weights

Value = $2 \times 10^2 + 3 \times 10^1 + 5 \times 10^0 = 235$ decimal

b) Binary

1	1	1	bits
2^2	2^1	2^0	weights

Value = $1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7$ decimal

Figure 1.1

software (software that satisfies a user's end need). An example of system software is the **operating system**—the large, complex program that manages all the resources of the computer, and provides an interface that allows a user to interact with the computer. Examples of application software are word processors and spreadsheet programs.

The principal hardware components of a computer system are the central processing unit, main memory (sometimes called primary memory), secondary memory, input devices, and output devices. The **central processing unit** (CPU) is the computational unit of a computer system. It can add, subtract, multiply, divide, and compare numbers. The CPU also controls the operation of the other units in the computer. The CPU, however, cannot do anything unless it is provided with instructions. The only kind of instructions the CPU can use are called **machine language instructions**. Machine language instructions are binary numbers that specify the operations the CPU is to perform. When we say that the CPU is **executing** an instruction, we mean that the CPU is simply performing the operations specified by that instruction.

The CPU gets the machine language instructions that tell it what to do from **main memory**. A program can be executed *only if it is in main memory*. Thus, to execute a program that resides on a hard disk, the program must first be copied from the hard disk into main memory. This requirement is the reason why a program does not start immediately when you click on its screen icon. Before it can start (i.e., before the CPU can execute its instructions), it has to be copied into main memory.

Main memory typically consists of billions of individual memory cells. The CPU can access these cells in any order. For example, it can directly access the third cell without first accessing the first and second cells. For this reason, main memory is sometimes referred to as **random access memory** (RAM). Main memory is **volatile**. That is, the data and instructions it holds are lost whenever power is turned off.

Secondary memory is nonvolatile memory. That is, its contents are not lost when power is turned off. A hard disk is a typical secondary memory device. Secondary memory is the depository of all the programs and data available on a computer system. Because main memory is volatile and of limited size, you obviously cannot use it for this purpose.

When you create a document with a word processing program, the document (as well as the word processing program and the operating system) sits in main memory. If your computer loses power, even for an instant, you will lose your document because main memory is volatile. For this reason, it is always a good idea to frequently save your document. Saving your document copies it from main memory to a secondary memory device. If your computer then loses power, the copy in main memory is lost but not the copy on the secondary memory device.

An **input device** provides information to the computer from the outside world. Examples of input devices are the keyboard and mouse. An **output device** provides information in the computer to the outside world. Examples of output devices are the LCD display and the printer.

The CPU and main memory are all electronic. They have no moving parts to slow them down. Thus, once main memory has the instructions and data required for a computation, the CPU can perform that computation at an incredible speed. Secondary memory devices, such as a hard disk, typically have moving parts. Thus, their operation is considerably slower than the CPU and main memory. *A computer is fast because its core components—the CPU and main memory—are all electronic.*

It is generally advantageous to have a large amount of main memory. Recall that a program must be in main memory to be executed. With a large amount of main memory, the operating system can simultaneously keep itself and all the active programs in main memory. Then, because all the active programs are already in main memory, the CPU can then switch from executing one program to another virtually instantaneously.

1.3 Let's Start Programming—Some Initial Considerations

When writing Java™ programs, you must pay attention to small details. For example, wherever a Java program requires a left brace (`{`), you must use a left brace—not a left parenthesis (`(`) or a left square bracket (`[`). You must also pay attention to the case of the letters you use. For example, the words `class`, `public`, `static`, and `void` must in lowercase. The words `String` and `System` must start with a capital letter followed by lowercase letters. Java is **case sensitive**. That is, it treats words spelled the same but differing in case as distinct words.

The Java language allows a program to be **formatted** (i.e., laid out on the page) in a variety of ways. For example, you can insert any number of spaces, tabs, or blank lines between any two **tokens** (i.e., meaningful units) of a program but not within a token. For example, you may write the following Java statement

```
System.out.println(20+3);
```

in this format (with spaces surrounding the plus sign):

```
System.out.println(20 + 3);
```

or in this format (with each token on a separate line):

```
System
.
Out
.
println
(
20
+
3
)
;
```

All three of these formats are equivalent from the computer's point of view. You may not, however, write

```
S ystem.ou t.pri ntln(2 0 + 3);
```

because `System`, `out`, `println`, and `20` are all single tokens, and, therefore, cannot have any embedded spaces.

Although the Java language does not require a specific format, you should, nevertheless, format your programs in the same way the sample programs in this textbook are formatted. Using this format, you will be less likely to make errors when you write programs. It will also make your programs easier to read and understand.

It is easy to make mistakes when writing a computer program. Even the best programmers make mistakes *all the time*. We call these mistakes **bugs**. **Debugging** is the process of eliminating the bugs in a program. Debugging is fun to do (no kidding). An incorrect program that you have written is a puzzle of your own making that you have to solve. When you figure out what's wrong, you'll get a wonderful sense of satisfaction.

1.4 Structure of a Java Program

A Java program consists of one or more classes. A class is the basic unit of a Java program. It has the following structure:

```
class class name
{
   .
   .
   .
}
```

A class can contain methods as well as other items that we will discuss later. A **method** is a named sequence of Java statements. A method has the following structure:

```
method header
{
      statements          } method body
}
```

The **header** of a method contains the name of the method as well as several other items. It is followed by the **body** of the method. The body consists of a sequence of statements enclosed by braces. When a computer **executes** a method, it performs the operations specified by the statements in its body.

A simple Java program—one consisting of a single method inside a single class—has the following structure:

```
class class name
{
    method header
   {
      statements
   }
}
```

Although not required by the Java language, it is a good idea to start the word `class` and its associated braces in the leftmost column. Then indent the left margin of the method from the left margin of the class. This indentation serves an important purpose: It shows us that the method is a subpart of the class. Also indent the statements in the method body from the enclosing braces.

Figure 1.2 shows a simple Java program. This program contains a single method named `main` inside a class named `Program1`. Although just about as simple as can be, this program

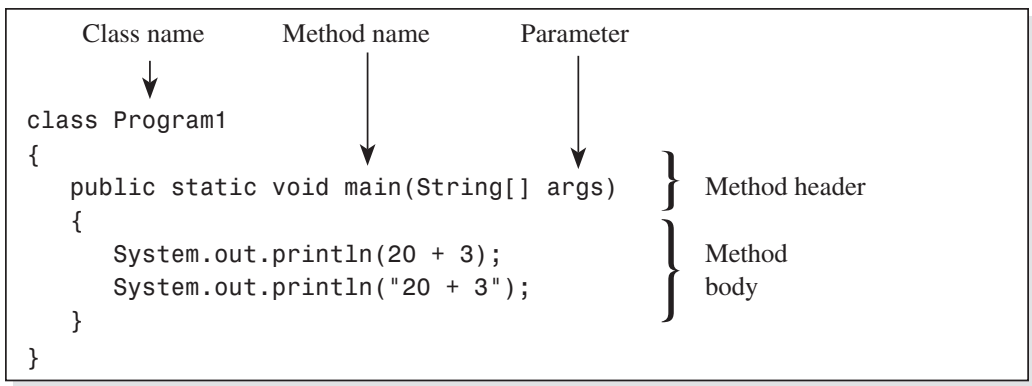

Figure 1.2

necessarily uses some features of Java that you will not be able to fully understand until you learn more about Java. So for now, just try to understand the program as much as you can.

Let's examine the components of the program in Fig. 1.2 in order of appearance.

`class`	Indicates the beginning of a class. Words like `class` that have a predefined meaning are called **reserved words** or **keywords**.
`Program1`	This is the class name. You choose the class name, but it cannot be a reserved word. For example, you can use `Butterfly` or `Rocket` instead of `Program1`. By convention, class names should start with an uppercase letter, although the Java language does not require this.
`{`	The left brace marks the beginning of the body of the class. Braces look very much like parentheses on a computer screen. *Be sure you use a brace here—not a parenthesis.*
`public`	This is the start of the header for the `main` method. `public` indicates that access to this method is unrestricted. That is, it is "available to the public." `public` is a reserved word.
`static`	Indicates that we can execute the method without first creating an object. We will discuss objects later in this chapter. `static` is a reserved word.
`void`	This is the **return type** of the method. `void` indicates that the method does not return (i.e., give back) a value. `void` is a reserved word. We will discuss return types in Chapter 5.
`main`	This is the method name. Every Java program must have a method named `main`. `main` is always executed first. By convention, method names should start with a lowercase letter.
`(`	The left parenthesis marks the beginning of the **parameter list**. It consists of a list of zero or more parameters, each preceded by its type and separated from the next by a comma. In this particular parameter list, only one parameter, `args`, is specified, preceded by its type, `String[]`.
`String[]`	This is a data type. The square brackets in `String[]` stand for "array." Thus, `String[]` stands for "`String` array." Its use here indicates that the next item in the program (the parameter `args`) has the type `String[]`. We will discuss `String` arrays in Chapter 9.
`args`	This is the name of the **parameter**. You choose the name. It does not have to be `args`. We will discuss parameters in Chapter 5.
`)`	The right parenthesis marks the end of both the parameter list and the method header.
`{`	This left brace marks the beginning of the body of the method.

`System.out.println(20 + 3);`

When executed, this statement causes the computer to evaluate `20 + 3` and display the result on the display screen. Thus, this statement displays `23` at the current location of the display cursor. It then positions the display cursor at the beginning of the next line. Notice the semicolon at the end of the statement. Semicolons mark the ends of statements.

`System.out.println("20 + 3");`

This statement displays the sequence of characters inside the quotes and then positions the display cursor at the beginning of the next line. A string of characters enclosed in double quotes is called a **literal `String` constant**. Arithmetic expressions inside a literal `String` constant are not evaluated. Thus, this statement displays `20 + 3` rather than `23`.

`}` This right brace marks the end of the body of the method.

`}` This right brace marks the end of the body of the class.

`main` requires the specific header shown in Fig. 1.2, except for the name of the parameter (for example, you can substitute `parms` for `args`). Whenever you write a `main` method, use this header.

1.5 Compiling and Running a Java Program

To work with Java programs, you need to install the **Java Development Kit** (JDK) on your computer unless you have a Macintosh (the Macintosh comes with the JDK pre-installed). If you have a Windows or Linux system, see the Appendix for installation instructions.

To run (i.e., execute) the program in Fig. 1.2, you first have to create a file that contains the program. The name of this file should be the class name used in the program followed by a period and the extension `java`. The class name of the program in Fig. 1.2 is `Program1`. Thus, you should create a file whose name is `Program1.java`.

To create the file `Program1.java`, use a program called a **text editor**. A text editor allows you to enter text into a computer, edit (i.e., modify) the text if necessary, and then save the text to a computer file. Most computer systems come with a simple text editor. Windows, for example, has `notepad` and `edit`. To use `notepad`, simply click on its icon (which is under `Accessories` in the `All Programs` menu). To use `edit`, first switch to **command-line mode** by clicking on the `Command Prompt` icon (which is under `Accessories` in the `All Programs` menu). Then enter `edit` followed by the name of the file you wish to edit. Linux and Macintosh systems usually have one or more of the following

simple text editors: `pico`, `nano`, or `joe`. To use any of these programs, activate the `Terminal` program (which switches you to command-line mode). Then enter `pico`, `nano`, or `joe` followed by the name of the file you wish to edit.

After you create the file `Program1.java` that contains the program, you have to **compile** (i.e., translate) the program in this file. The CPU of a computer cannot directly execute Java instructions. Recall that the only type of instructions the CPU can directly execute are machine language instructions. Thus, before you can run a Java program, you must translate it to machine language. The Java compiler, `javac`, in the JDK performs this translation for you. To compile the program in `Program1.java`, put your computer in command-line mode if you have not already done so (see instructions above). Then enter

```
javac Program1.java
```

`javac` (the Java compiler) will then translate the Java program in the file `Program1.java` to machine language and output it to a file named `Program1.class`. The name of the output file created by the compiler consists of the name of the class in the program followed by a period and the extension `class`.

Different types of computers have different machine languages. The `javac` compiler translates Java programs to a particular machine language called bytecode. **Bytecode** is the name of the machine language for the **Java Virtual Machine** (JVM). To run the bytecode in `Program1.class`, you must run the `java` interpreter. The `java` **interpreter** makes your computer act like the JVM. The name of the Java interpreter is `java`. To run the `java` interpreter on `Program1.class` (which causes the execution of the program in `Program1.class`), enter

```
java Program1
```

Note that you specify the class name (`Program1`), not the file name (`Program1.class`) of the translated program when you invoke the interpreter. After you enter this command, the `java` interpreter will execute the program in `Program1.class`. You will then see the following output (produced by the program in `Program1.class`):

```
23
20 + 3
```

Be sure to use the correct case when entering these commands. If, for example, you enter

```
java program1
```

with a lowercase letter `p` at the beginning of the class name, the program will not run.

The two files—`Program1.java` and `Program1.class`—contain the same program but in different forms. `Program1.java` contains the Java **source program** (i.e., the original Java program). `Program1.class` contains the corresponding bytecode.

When we interact with the computer when it is in command-line mode, we say that we are using the **command-line interface**. To edit, compile, and run a Java program when we are using the command-line interface, we enter commands on the keyboard to invoke the programs that perform those functions. Another approach to editing, compiling, and running Java programs is to use an **integrated development environment** (IDE). An IDE combines everything we need for program development into a single, convenient package. It also typically includes a **debugger**—a tool that facilitates the debugging of programs. If you use an IDE, you will be able to write and debug programs more easily than if you use the command-line interface. The disadvantage of IDEs is that you have to learn how to use them. For this reason, it is probably best to start out with the command-line interface so initially you can focus all your efforts on learning Java. Once you have learned the basics of Java, you can switch over to an IDE. Two excellent (and free) IDEs that are well-suited for students learning Java are DrJava (available at `http://www.drjava.org`) and BlueJ (available at `http://www.bluej.org`). Both are easy to learn and easy to use. Tutorials on how to use DrJava and BlueJ are available at their respective websites.

Java programs are **portable**—that is, Java programs can be executed without modification on any computer that has a `java` interpreter. The Java interpreter for a computer makes that computer act like the JVM. Thus, any machine language program for the JVM can run without modification on every computer with a Java interpreter. For example, the program in `Program1.class` can run on any computer system with a Java interpreter, regardless of the type of system on which it was created. For example, we can create `Program1.class` on a Windows system (by compiling `Program1.java`) and then run `Program1.class` on a Macintosh system (by running the `java` interpreter on the Macintosh). The portability of Java is one of its most attractive features, and distinguishes it from most other computer languages.

Instead of programming in Java and then using the Java compiler to translate our programs to machine language, why don't we program directly in machine language? We could do this, but it would be far more difficult and time-consuming than programming in Java. It would be like using a toothbrush to clean your basement floor.

One brief note on terminology: The term **code** refers to some portion of a program. For example, when we say "the *code* on Lines 4 through 7," we mean that portion of the program on Lines 4 through 7.

1.6 Syntax, Logic, Compile-Time, and Run-Time Errors

A `syntax error` is a violation of the rules of a programming language. For example, if you use a left parenthesis in a Java program where a left brace is required, you have made a syntax error—you have violated the rules of the Java language. If, on the other hand, you tell the computer to do the wrong thing using correct syntax, you have made a **logic error**. For example, suppose you want to compute and display the sum of 20 and 3. To do this, you write a program that contains the following statement:

```
System.out.println(200 + 3);
```

Unfortunately, this statement contains a mistake: It computes the sum of 200 and 3 rather than 20 and 3. It, however, is a perfectly legal instruction. When you run the program, it will display 203, an incorrect answer to the problem you wanted the computer to solve.

Errors that the compiler can detect are called **compile-time errors**. The `javac` compiler knows all the rules of the Java language. Thus, it can detect all syntax errors. It, however, cannot detect most logic errors. How, for example, could the compiler know in the preceding example that you want to compute the sum of 20 and 3 rather than 200 and 3? It cannot. Thus, this logic error would go undetected by the compiler.

A **run-time error** is an error that is detected at run time (i.e., during program execution). For example, dividing by zero (which is an illegal operation) is a run-time error. What happens when a run-time error occurs depends on the program. One alternative is to terminate execution immediately. Another alternative is to **recover from the error** (i.e., handle the error in such a way that the program can continue executing). When a run-time error causes a program to terminate, we often say (over-dramatically) that the program has "crashed" or has "blown up."

Logic errors are far more insidious than syntax errors. You can easily determine if your program has syntax errors: simply compile your program. If it has any syntax errors, the compiler will generate error messages. However, logic errors generally do not produce error messages at compile time. And they may not produce any error messages at run time. So you may have logic errors in your program but not know it. Your program may be providing an answer that is completely wrong. *Just because a computer provides an answer for some problem does not mean the answer is correct.*

1.7 A Program with Objects

Java is an **object-oriented** (OO) programming language. That is, Java programs, when executed, can create and use objects. An **object** is a structure that contains both data and the methods that operate on that data.

```
 1 class Program2
 2 {
 3    public static void main(String[] args)
 4    {
 5       String p, q;                    // create references p and q
 6       p = new String("hello");        // create object
 7       q = p.toUpperCase();            // create second object
 8       System.out.println(p);          // display p-string
 9       System.out.println(q);          // display q-string
10       String r = new String("bye"); // create ref r and object
11       String s = "all done";          // create ref s and object
12       System.out.println(r);          // display r-string
13       System.out.println(s);          // display s-string
14    }
15 }
```

Comments

Figure 1.3

Object-oriented programming languages have significant advantages over other types of programming languages. We will begin investigating the OO features of Java in Chapter 6. So that we don't have to wait until then to get a taste of object-oriented programming, let's now look at a simple program that creates objects.

Objects are constructed from classes. A class is a blueprint for an object. The Java programming language comes with many predefined classes. Thus, we do not have to create classes before we can create objects; we can simply use the predefined classes as long as they satisfy our requirements. The program in Fig. 1.3 constructs objects from the predefined class named `String`. An object constructed from the `String` class contains data (a string) and a collection of methods that operate on that string.

Let's now examine Fig. 1.3 in detail. The line numbers on the left are not part of the program—we have added them so that we can easily refer to specific lines. If you create a file for this program, it should *not* contain these line numbers. However, the **comments**, the items on the right that start with two slashes, are part of the program. We are using comments here to inform the human reader what each statement does. We have added color to our comments to make them stand out from the rest of the program. However, when you enter comments in a program, you should enter them in the same way you enter everything else in the program. Comments have no effect on how the program is compiled—they are only for the benefit of the human reader. Comments can make a program easier to read and understand (even for the programmer who wrote the program!). You should always comment your programs. But do not over-comment. Specifically, do not

comment on what should be obvious to the intended reader. Such comments waste the reader's time.

Line 5 in Fig. 1.3 creates two `String` reference variables, one named `p` and one named `q` (see Fig. 1.4a). A **variable** is a named location in memory in which a value can be stored. Line 6 is an **assignment statement**, which works this way: The right side of the assignment statement produces a value. This value is then stored in the variable on the left side. The `new` operator on the right side triggers the construction of an object from the class whose name appears right after `new`. Thus, here the `new` operator triggers the construction of an object from the `String` class. This object contains a string (the string within the quotes on Line 6) and methods that operate on that string. One of these methods is named `toUpperCase` (see Fig. 1.4b). The `new` operator not only triggers the construction of the object, it also returns the **reference value** that "points to" the object. This returned value becomes the value of the right side of the assignment statement. It is assigned to `p`, the variable on the left side of the assignment statement. A reference value is a binary number that designates the location in memory at which an object resides. Thus, it, in effect, points to that object, so we represent it in our diagrams with an arrow. The net effect of Line 6 is to create a `String` object and assign to `p` the reference value that points to the new object. Thus, `p` points to the new object.

An object has no name. However, we can access an object via the reference that points to it. For example, to **invoke** or **call** (i.e., to cause to be executed) the `toUpperCase` method in the object created by Line 6, we use

```
7       q = p.toUpperCase();          // create new object
```

`toUpperCase()` on the right side of this assignment statement indicates that the `toUpperCase` method is to be invoked. But which `toUpperCase` method? The initial `p` on the right side indicates that it is the `toUpperCase` method *in the object to which* `p` *points.* When invoked, `toUpperCase` creates a second object identical to the object in which the invoked `toUpperCase` method resides, except that in the new object all the letters in the string are in uppercase. Thus, Line 7 creates a new `String` object that contains the string `HELLO` (see Fig. 1.4c). It also returns a reference value to the new object. This reference is then assigned to `q` by the action of the assignment statement in Line 7. Note that Line 7 contains parentheses to the right of the method name. Parentheses are required when we invoke a method.

Lines 8 and 9 do *not* display the reference values in `p` and `q`. Instead, they display the string data in the objects to which they point. Thus, these lines display `hello` and `HELLO`, respectively.

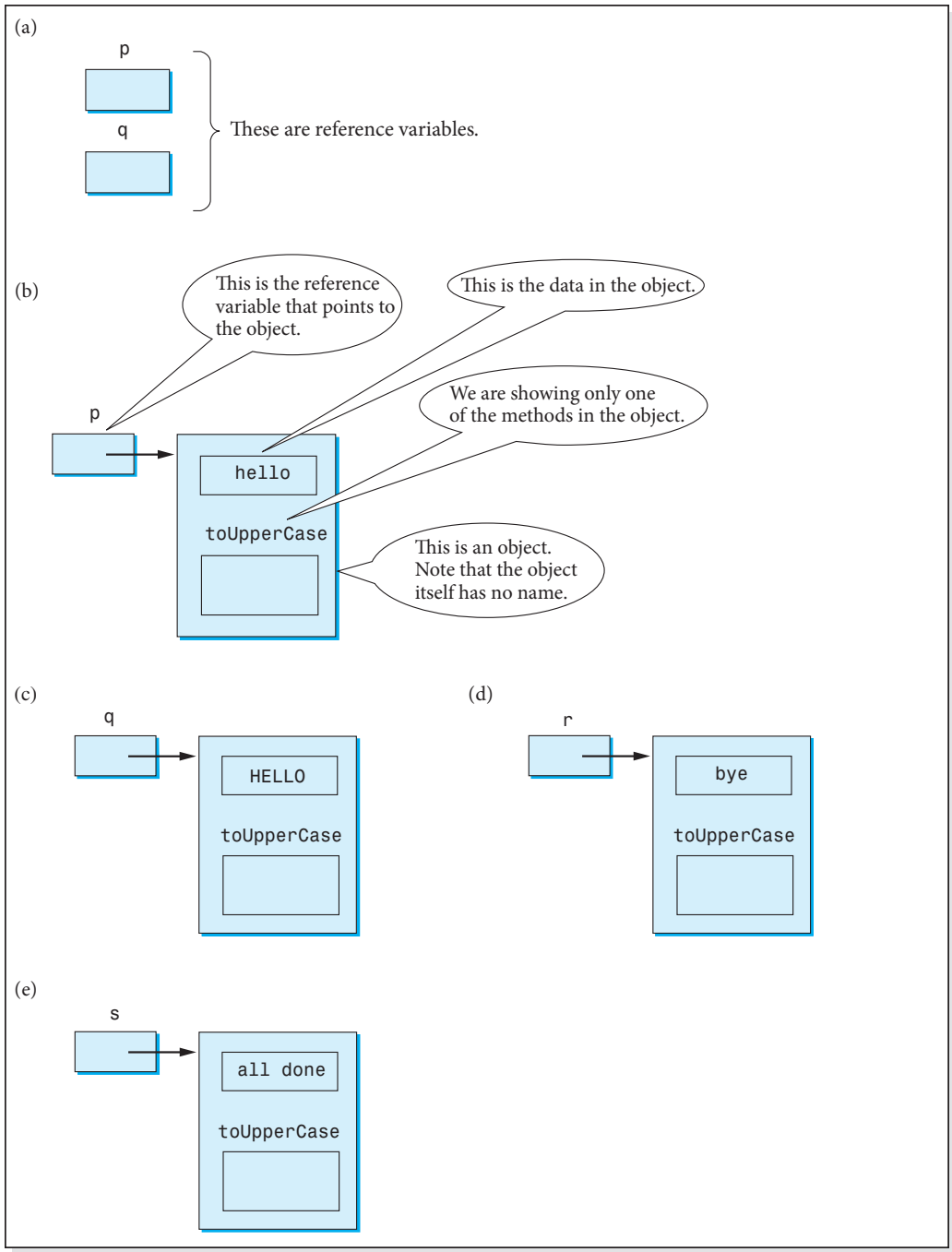
(a)
p
q
These are reference variables.
(b)
This is the reference variable that points to the object.
This is the data in the object.
We are showing only one of the methods in the object.
p
hello
toUpperCase
This is an object. Note that the object itself has no name.
(c)
q
HELLO
toUpperCase
(d)
r
bye
toUpperCase
(e)
s
all done
toUpperCase

Figure 1.4

On line 10 we create a reference variable r, construct an object containing bye, and assign r the reference to the object, all in one statement (see Fig. 1.4d):

```
10        String r = new String("bye");// create reference r and object
```

Strings are used so often in programs that a shorthand technique for constructing String objects is allowed: If we simply specify a string within quotes, a String object is automatically constructed that contains that string. We do not have to use the new operator and specify the String class. For example, line 11 creates the reference variable s and a String object containing all done. It then assigns to s the reference to the object (see Fig. 1.4e):

```
11        String s = "all done";        // create reference s and object
```

Lines 12 and 13 then display the r and s strings:

```
12        System.out.println(r);        // display r-string
13        System.out.println(s);        // display s-string
```

That is, they display the strings in the objects to which r and s point—namely, bye and all done.

Because objects do not have names, when talking about objects, we have to refer to them by using the reference variables that point to them. For example, to refer to the object to which p points, we say "the p object." This phrase is shorthand for the phrase "the object to which p points."

Now that we understand something about objects, we can better understand the program in Fig. 1.2. Let's reexamine the two println statements in this program:

```
System.out.println(20 + 3);
System.out.println("20 + 3");
```

These statements are actually method invocations. System.out is a reference variable that points to an object that contains the println method. Thus, each of these statements invoke the println method in this object. The value of the **argument** (i.e., what is inside the parentheses in a method invocation) is passed to the println method when it is invoked. In the first println statement, 20 and 3 are first added. The result, 23, is then passed to the println method. Thus, this println statement displays 23. In the second println statement, the string "20 + 3" is passed. The println method simply displays the string it is passed—it does not look inside the string for arithmetic expressions to evaluate. Thus, this println statement displays 20 + 3.

Generally, to invoke a method in an object, we first have to construct the object. However, some objects are automatically constructed for us—like the object that contains the println method to which System.out points. Thus, we can use these objects without first having to construct them.

In the invocation of `toUpperCase` on Line 7 in Fig. 1.3, we are not passing any arguments (the arguments, if there were any, would be specified inside the parentheses). `toUpperCase` does not need any arguments passed to it because it operates on the data—the string `hello`—in the object.

The program in Fig. 1.3 illustrates two important features of objects. Each object combines data and methods into a *single* functional unit. We access the data in an object via the methods in the object. We call this feature **encapsulation**. We have no direct access to the data in an object (we access the data only via the methods in an object). Moreover, we can use the object without knowing how the data it holds is represented within the object. We call this feature **information hiding**. For example, consider the string `"do"`. It is represented in computer memory by the binary code for `'d'` followed by the binary code for `'o'`. But suppose in the next memory slot there just happens to be the binary code for some other character, say `'t'`. In that case, do we have the string `"do"` or the string `"dot"`? To avoid this sort of ambiguity, a string is represented either by recording its length (i.e., the number of characters) in addition to its binary codes, or by terminating its sequence of binary codes with a special code that means "end of string." With the first approach, we store the string `"do"` by storing the binary codes for the letters `'d'` and `'o'`, and the length 2 (see Fig. 1.5a). With the second approach, we store the binary codes for `'d'` and `'o'`, followed by the end-of-string code (see Fig. 1.5b). With either approach, the string is well defined—that is, we know for sure where it ends.

But we don't need to know any of these details to use a `String` object that holds the string `"do"`. If, for example, we want an uppercase version, we simply invoke its `toUpperCase` method.

Because of encapsulation and information hiding, you can easily use classes someone else has written—*you do not have to know the internal details of a class to use it.* By using classes

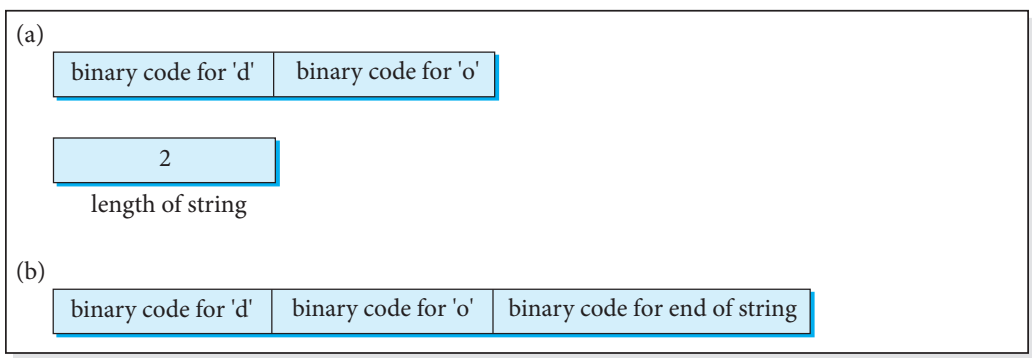

Figure 1.5

someone else has written, you can dramatically cut down the time to implement complex programs. Suppose, for example, a professional programmer wants to write a simple word processing program that uses a **graphical user interface** (i.e., an interface that uses graphical components such as drop-down menus and buttons). If the programmer writes the entire program from scratch, it will take, perhaps, 1 year. If, however, the programmer takes advantage of Java's predefined classes, it will take, perhaps, 1 day. *Making code easy to share is one of the principal advantages of object-oriented programming.*

1.7.1 Note on Prep, Laboratory, and Homework Exercises

Any Java code that appears in the prep, laboratory, and homework exercises is available in the collection of source code that accompanies this textbook. *Don't waste your time typing in this code—use the source code provided.* The names of the files in the source code collection include the chapter number, the category letter ("p" for prep, "e" for lab exercise, and "h" for homework) and the exercise number. For example, `C1p4.java` is the code for Chapter 1, Lab Prep 4; `C1e5.java` is the code for Chapter 1, Lab Exercise 5; `C6h1.java` is the code for Chapter 6, Homework 1. Use the same file naming system when you create files for which a source code file does not already exist. For example, use `C1e11.java` for the program you create for Chapter 1, Lab Exercise 11.

Include a comment containing your name at the beginning of your programs. Include additional comments to explain any nonobvious features of your code. Properly format your programs (use the sample programs in this textbook as models).

LABORATORY

Laboratory 1 Prep

1. Invent a mnemonic (i.e., something that aids memory) so that you can easily remember the header required for the `main` method (see Fig. 1.2).
2. Which of the following identifiers are the names of Java classes: `main`, `Random`, `public`, `String`, `int`, and `Integer`.
3. What are the binary equivalents for the decimal numbers 0 through 15?
4. What is wrong with the following command,

   ```
   javac Program1
   ```

   and the following Java statement:

   ```
   System.out.println[2 + 2]
   ```

5. Write a single statement that creates a `String` reference variable `s`, creates a `String` object that contains the string `"dog"`, and assigns `s` the reference to the object created.

Laboratory 1

1. Compile and run the program in Fig. 1.2 by entering

   ```
   javac Program1.java
   java Program1
   ```

   Now try running the program again by entering (with a lowercase `p`)

   ```
   java program1
   ```

   What happens?
2. Compile and run the program in `C1e2.java`. Verify that the program compiles and runs correctly. Change `args` to `goofy`. Verify that the program still compiles and runs correctly. Java does not require the name `args` for the name of the parameter in the `main` method.
3. Compile and run the program in `C1e3.java` in the source code package. Verify that the program compiles and runs correctly. Remove the semicolon at the end of the first `println` statement. Compile the program and inspect the compile-time error that results. Does the error message correctly indicate the location of the error in the source program?

LABORATORY

4. What's wrong with the following program:

```
Class C1e4
{
    public static void Main(String args)
    {
        system.out.println("C1e4");
    }
}
```

   Compile and run it to check your answer. Fix the program so that it runs correctly.

5. Compile the following program to verify it has four syntax errors. For each error, what is the error message produced by the compiler?

```
clas s C1e5
{
    public static void main(String[] args)
    {
       System.out.println("Good
       bye");
       System.out.println("Go od by e");
       System.out.println("all done);"
    )
}
```

   Fix the program so it runs correctly.

6. Verify that the following program runs correctly:

```
class
  C1e6
  { public static
    void main(String[] args) {System.out.println("hello");
  }}
```

   Reformat the program so that it is easier to read. You should never write a program with sloppy formatting because it makes the program difficult to read. Always format your programs the way the programs in this textbook (except for this one) are formatted. When you indent, always indent the same number of columns (three columns is a good choice) from the start of the previous line. Don't indent fewer than three columns (it will make it difficult for your eye to pick up the indentation). *Indentation is essential! Always indent properly.*

7. Does the following program work? Note that `public` follows `static` in the method header.

LABORATORY

```
class C1e7
{
   static public void main(String[] args)
   {
      System.out.println("hello");
   }
}
```

Now switch the positions of `void` and `public`. Does the program still work? The return type of method must immediately precede the method name. Thus, `void` (which is the return type of `main`) must immediately precede `main` and follow `public` and `static`. `public` can either precede or follow `static`, although it is customary to put `public` first.

8. Insert the following statement in an otherwise correct program. What happens when you run the program?

```
System.out.println(5/0);
```

9. Write a program that displays your name. Use a file named `C1e9.java`. Use the class name `C1e9`. Compile and run.

10. Compile and run the following program:

```
1 // escape sequences example
2 class C1e10
3 {
4   public static void main(String[] args)
5   {
6      System.out.println("hel\"lo"); // escape sequence
7   }
8 }
```

The initial quote on Line 6 is a *special* quote. It is special in the sense that it starts the string constant. The third quote on Line 6 is also a *special* quote. It is special in the sense that it ends the string constant. The middle quote is an *ordinary* quote in the sense it does not act in a special way (it does not end the string constant). This quote is made ordinary by the preceding backslash character. (*Rule:* A backslash that precedes a character that is normally special makes that character ordinary.) We call sequences that start with a backslash **escape sequences**. When the preceding program runs, it displays:

```
hel"lo
```

The backslash-quote sequence is displayed as a single quote. Now delete the backslash character in the `println` statement on Line 6 to get

```
System.out.println("hel"lo"); // escape sequence
```

LABORATORY

Because the middle quote is now not backslashed, it is a special quote. That is, it ends the string constant, making the characters to its right look like garbage. Verify that the program no longer compiles correctly.

Like the quote, a backslash can be either a special character (if it is not preceded by a special backslash) or an ordinary character (if it is preceded by a special backslash). Change Line 6 to

```
System.out.println("hel\\\"lo"); // escape sequence
```

Compile and run. We now have two consecutive escape sequences: `\\` followed by `\"`. The first backslash makes the second backslash an ordinary character. The third backslash makes the middle quote an ordinary character. Note that in sequence

```
\\\"
```

the third backslash is not an ordinary backslash even though it is preceded by a backslash because the middle backslash is an ordinary backslash by virtue of the first backslash. Thus, this sequence represents an ordinary backslash followed by an ordinary quote. This `println` statement displays the following seven characters:

```
hel\"lo
```

11. Write a program that displays the following sequence of eight characters:

    ```
    /\/\"/\"
    ```

12. Can a blank line be inserted anywhere in a Java program? Run a test program to determine the effect of blank lines.
13. Determine the effect of

    ```
    System.out.print("hello");
    System.out.print("hello");
    ```

    How does it differ from

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

    and from

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

    (*Hint:* The "`ln`" in `println` means "go to the beginning of the next line.")
14. Run a test program to determine the effect, if any, of

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

LABORATORY

15. Is the following a legal statement:

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

    Compile a program that contains it to check your answer.

16. Write a program that computes and displays the sum of 1, 2, 3, 4, and 5.

17. What does the following program display?

```
class C1e17
{
   public static void main(String[] args)
   {
      String s = "hellobirdbye";
      System.out.println(s);
      System.out.println(s.length());
      String t = s.substring(0,5);
      System.out.println(t);
      System.out.println(s.equals(t));
      System.out.println(t.equals("hello"));
   }
}
```

    `length`, `substring`, and `equals` are methods in `String` objects. `length` returns the number of characters in the string. `substring` creates a new object that contains the specified substring of the given string. Its first argument specifies the position at which the substring starts (0 corresponds to the first character). Its second argument specifies the position in the given string that is just beyond the end of the substring. `equals` compares two strings for equality.

Homework 1

1. Write a program that displays your name in a box, like so:

```
+-------------------------+
|                         |
|        your name        |
|                         |
+-------------------------+
```

2. Write a program that displays a plus sign formed with asterisks. The vertical and horizontal strokes of the plus sign should each be formed with 11 asterisks.

3. Write a program that displays the following sequence of characters:

```
\""\\"""\\\"""\\\\
```

LABORATORY

4. Write a program that *computes* and *displays* the sum of 5 and −20. Your program should produce a display that looks like this:

   ```
   5 + -20 = -15
   ```

   (*Hint:* Use a `print` statement to display `"5 + -20 = "` [see Lab Exercise 13]. Then use a `println` statement to compute the sum of 5 and −20 and display the result.)

5. Is there a `toLowerCase` method in string objects? Run a test program to find out.

6. Write a program that creates three `String` reference variables. Each variable should point to the *same* object. This object should contain the string `"MeMeMe"`. Display the string pointed to by each reference variable. (*Hint:* Use the assignment statement to assign one reference variable to another.)

7. Write a program that creates three `String` reference variables. Each variable should point to a different object. One object should contain the string `"UPPER"`, one object should contain the string `"lower"`, and one object should contain the string `"MiXeD"`. Display the string in each object. For each object, invoke the `toUpperCase` method and display the string in the new object created.