CHAPTER

1 What Is Programming?

Chapter Preview

Before writing programs in the Java language, we demonstrate the thought processes and problem-solving techniques necessary for programming. We present a problem—leading a mechanical mouse through a maze—and work through the design of an *algorithm* to solve the problem, and a *program* to implement the algorithm. We discuss the importance of *object-oriented programming*. To provide some grounding in the operation of computers, we discuss the architecture of computers, the representation of various kinds of data, and the mechanics of entering and running Java programs.

> "Again, it [the Analytical Engine] might act upon other things besides numbers, were objects found whose mutual fundamental relations could be expressed by those of the abstract science of operations, and which should be also susceptible of adaptations to the action of the operating notation and mechanism of the engine …Supposing, for instance, that the fundamental relations of pitched sounds in the science of harmony and of musical composition were susceptible of such expression and adaptations, the engine might compose elaborate and scientific pieces of music of any degree of complexity or extent."
>
> —Lady Ada Augusta, Countess of Lovelace
> Commenting on Babbage's Analytical Engine
> In: *Scientific Memoirs, Selections from the Transactions of Foreign Academies and Learned Societies and from Foreign Journals,* edited by Richard Taylor, F.S.A., Vol III
> London: 1843, Article XXIX.

The popular perception of the computer is as an all-powerful entity that, with seemingly little human effort or input, is responsible for running business, government, science, and many more mundane aspects of our lives. But a computer is no more than an inanimate collection of wires and silicon, organized so that it can quickly perform simple operations such as adding numbers. So how does it come to have such power? What sleight of hand transforms this box of electrical components into a powerful tool?

program

Programming performs the magic. A *program* is a set of directions that tells the computer exactly what to do; a program thus is the medium used to communicate with the computer. Programs can be written in many *programming languages*—languages for specifying sequences of directions for the computer. In this book we use the language Java.

programming language

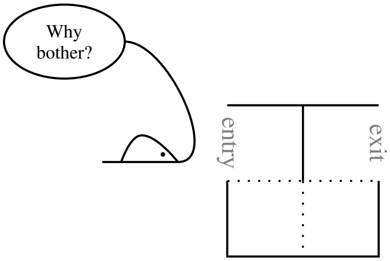
algorithm

Our first job in programming is to clarify the problem. *Exactly* what is required? How can the job be broken down into manageable pieces? What *algorithm*—sequence of steps—is appropriate to solve the problem? How can this algorithm be turned into a program? Does the program work? Is it written as clearly as it can be? Is it fast? Can it be changed easily if needed? Does it demand too great a fraction of the computer's resources? These are the questions confronting the *computer programmer*, the person who designs and writes the program.

computer programmer

1.1 Mechanical Mouse in a Maze

We start learning to program by analyzing a problem that contains, in miniature, the basic components of any programming problem. We want to give instructions to a mechanical mouse so it can get through a maze. For our purposes, a *maze* is a rectangular arrangement of square rooms; adjacent rooms may be separated by a wall, or the boundary between them may be open. We place the mouse facing the entrance to a maze in which solid lines represent walls between rooms and dotted lines represent open boundaries:

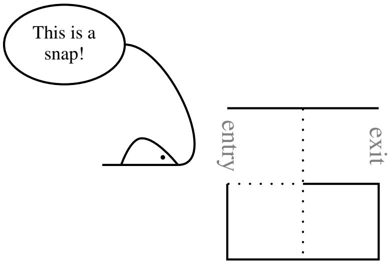


The mouse is like a computer in that it knows how to follow only certain very simple instructions: `step forward` into the next room, `turn right` in place, or `turn left` in place. We must *program* the mouse by writing the precise instructions it must follow to get through the maze. For this simple maze, the instructions are

```
step forward;
turn right;
step forward;
turn left;
step forward;
turn left;
step forward;
turn right;
step forward;
```

This sequence of instructions is comparable to a computer program.

What happens if the maze has a different configuration? For example, what if we had the following?

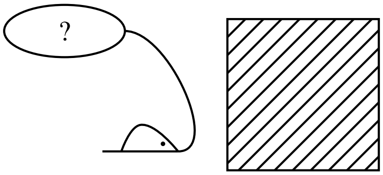


Then our preceding program would not work correctly, but the simpler program

```
step forward;
step forward;
step forward;
```

would work.

It is easy to write out instructions to the mouse for any specific maze as long as we know the maze's internal configuration. But what if the inside of the maze is hidden from view, beneath an opaque cover? All we see is a mouse-eye view of things:



Now our problem is to give the mouse proper instructions to get it through *any* maze. In fact, we have to approach the problem from a completely different perspective. In the language of computer programming, it is our task to figure out an *algorithm* to solve this task—a general method precise enough to be turned into a mouse program — and then to write the corresponding *program*, the exact sequence of mouse instructions.

algorithm
program

The first step in computer programming always is to make the problem as precise as we can. Where is the mouse initially placed with respect to the maze? What should happen if the maze has no exit? We'll assume the mouse is placed in front of the entrance to the maze so that a single `step forward` instruction takes it into the maze. Our instructions to the mouse must get it out the exit of the maze or back out the entrance if there is no path to the exit or no exit at all. This problem, of course, is impossible to solve unless the mouse is capable of examining the inside of the maze and making decisions based on what it finds as it goes along; the three instructions `step forward`, `turn left`, and `turn right` are not sufficient. The mouse, however, also can look forward myopically to see whether there is a wall immediately in front of it, and it can detect when it is inside the maze. These, then, are five instructions that the mouse can follow:

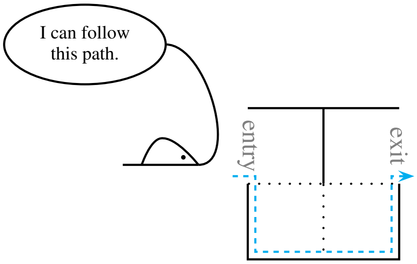
```
step forward
turn right
turn left
facing a wall?
inside the maze?
```

We are limited to these instructions to get the mouse through the maze.

First, we must design an algorithm for getting through an unspecified maze, and then we need to express the algorithm in terms that the mouse can follow. For the mouse in the maze, the algorithm we will use is a familiar trick for going through a maze:

Have the mouse walk hugging the wall to its right.

In our first example above, the mouse would travel like this:



Our first job is to convince ourselves that this algorithm actually works. As is often the case, this is not obvious. A careful argument would hinge on the observation that the mouse never hugs the same wall twice. We will not prove this, but you should run the algorithm on enough examples to convince yourself.

QUICK EXERCISE 1.1

Follow this algorithm on the second example above. (Note that the path given by this algorithm is longer than the path we chose for this maze, which was just to walk straight ahead from entry to exit. We never said this algorithm would always find a *good* path.)

QUICK EXERCISE 1.2

What does this algorithm do if there is no separate exit from the maze?

QUICK EXERCISE 1.3

An alternative algorithm would be to have the mouse walk hugging the wall to its left. Give a maze for which this algorithm gives a better result—that is, a shorter path through the maze—than our "hugging the wall to the right" algorithm. Then give a maze for which this algorithm gives a worse result.

We have an algorithm—a precise method that will always get us through the maze—but that is not enough. We need to express this algorithm with the set of five instructions available to our mouse. This is always the situation in computer programming: First we need to discover an algorithm, then we need to express the algorithm using the limited repertoire of actions that the computer has available to it.

For example, the algorithm directs us to keep the wall to the mouse's right, but the only instruction the mouse understands about walls is `facing a wall?`. How can it check if there is a wall to its right? It must `turn right` first and then ask whether it is facing a wall. If it is, then it originally had a wall to its right, so it should turn back. Now the question is, Can it move forward? If it is facing a wall, then it has a wall to its right *and* a wall in front, and it should keep turning left to find an opening. Thus, the instructions to tell the mouse to find an opening with a wall to its right are

```
turn right;
if facing a wall? then
  turn left and if facing a wall? then
    turn left and if facing a wall? then
      ...
```

continuing as long as necessary—that is, turn right and then keep turning left until no wall is in front. Since the mouse got into this room to begin with, there must be at least one opening; eventually the preceding instructions will have it facing an opening with a wall to its right.

We're not done, of course. We've figured out how to perform one part of the algorithm—finding an exit from any room in the maze, being sure that a wall is to our right—but not the entire algorithm. Before finishing it, we introduce another aspect of computer programs: their notation. If our mouse spoke Java, it would not understand the above set of instructions, because they are not in Java notation. Instead, we would—and will, from now on—write it as

```
turn right;
if (facing a wall?) {
  turn left;
  if (facing a wall?) {
    turn left;
    if (facing a wall?) {
      ...
    }
  }
}
```

omitting the words *then* and *and*, and using `{` and `}` to indicate the grouping of instructions and `(` and `)` to denote a question or condition.

We still haven't said what the `...` means. It means to continue with the same instructions while the mouse is facing a wall, stopping only when the mouse is *not* facing a wall. So we write it instead as

```
turn right;
while (facing a wall?) {
  turn left;
}
```

which means, "after turning right, repeat the process of checking whether you're facing a wall, and if you are, turn left." The distinction between *while* and *if* is that *while* means "continue checking as long as" and *if* means "check the current state of things just one time."

After following these instructions, the mouse will be facing an opening. In other words, it can now step forward (without crashing into a wall).

```
turn right;
while (facing a wall?) {
  turn left;
}
step forward;
```

This is all there is to making a single move. However, we must consider one small detail. What happens if the mouse reaches the end of the maze, that is, manages to exit the maze? What is the meaning of the code when the mouse is *outside* the maze? The mouse must be able to detect when it is outside the maze and simply do nothing if asked to make a move in that circumstance:

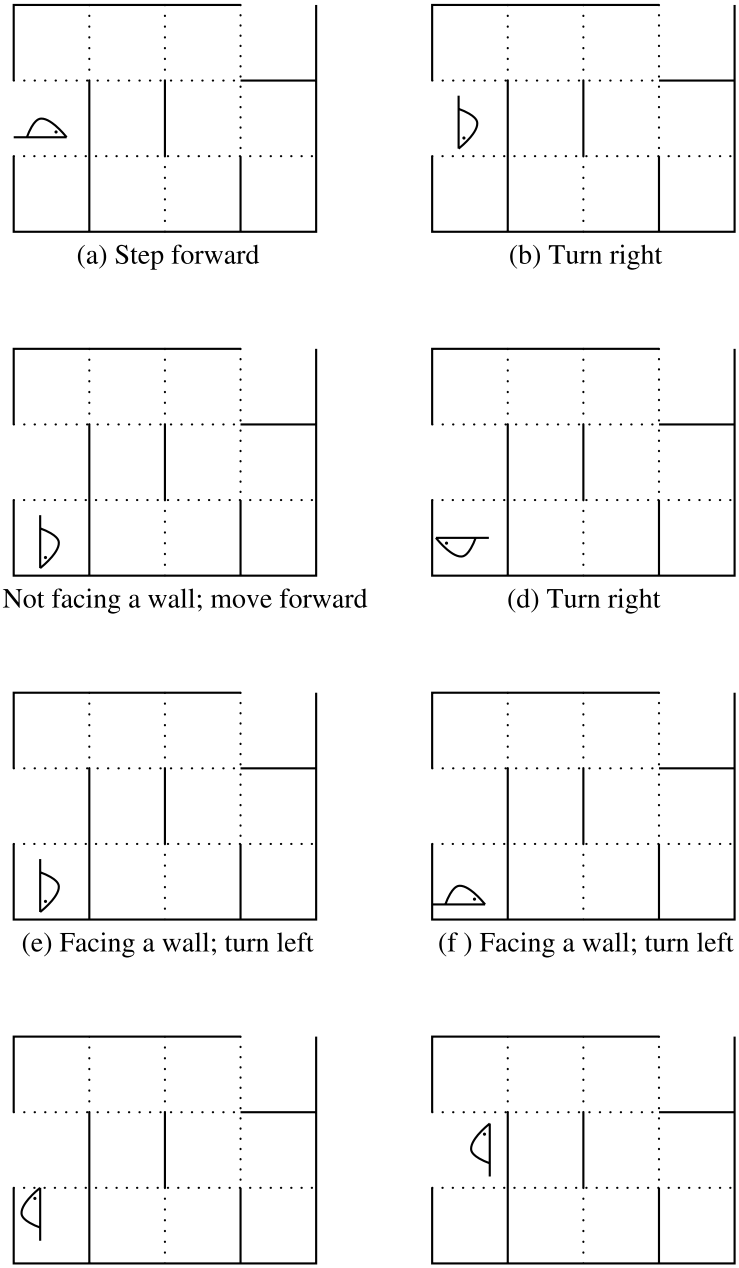
```
while (inside the maze?) {
  turn right;
  while (facing a wall?) {
    turn left;
  }
  step forward;
}
```

This raises another small problem. What happens with the very first move? The mouse cannot execute the code just shown, since initially it is outside the maze. The solution is that the mouse's very first move is simply `step forward`, and subsequent moves are given by the preceding code. Here is the final version of the program:

```
1   step forward;
2   while (inside the maze?) {
3     turn right;
4     while (facing a wall?) {
5       turn left;
6     }
7     step forward;
8   }
```

How simple and elegant this program is! It reads almost like English, but it expresses exactly how the mouse (with its limited abilities) can be made to trace its way, step by step, through a maze by keeping its right paw on the wall.

Let's follow the mouse's trip for a bit to make sure we understand the algorithm. The first few steps are shown in Figure 1.1. The mouse begins by following the first instruction and steps forward into the entry (*a*). Since it is inside the maze, the mouse follows the next group of instructions, starting by turning right (*b*). It is not facing a wall (that is, its nose is not touching a wall), so it doesn't turn left, but instead steps forward (*c*). It is still inside of the maze, so it again follows the group of instructions between lines 3 and 7. First, it turns right (*d*). Since it is facing a wall, it turns left (*e*) and checks again; still facing a wall, it turns left again (*f*) and checks again; *still* facing a wall, it again turns left (*g*). Finally, it is no longer facing a wall, so it skips to line 7 and steps forward (*h*).



(a) Step forward

(b) Turn right

(c) Not facing a wall; move forward

(d) Turn right

(e) Facing a wall; turn left

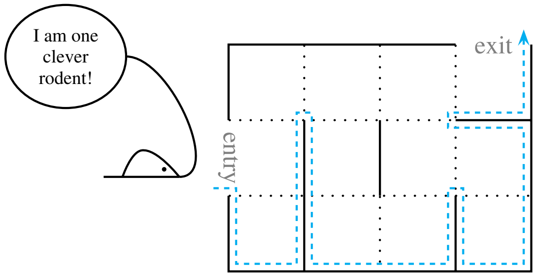
(f) Facing a wall; turn left

(g) Facing a wall; turn left

(h) Move forward

FIGURE 1.1 The start of the mouse's trip through the maze.

In the end, the mouse follows the path shown by the dashed line in the following picture:



Quick Exercise 1.4

Complete the next eight steps in the narrative of the mouse's trip that was begun in Figure 1.1.

The mouse takes a rather circuitous route to the exit, going through every room once and through three of the rooms twice! We could have taken a different approach to this problem, leading to a different, possibly shorter, path through the maze. With programming, the only limit is our ingenuity.

This program is the result of breaking down the problem at hand into a workable *algorithm*. The algorithm was then expressed using the five simple instructions available to our mechanical mouse. By expressing the algorithm in the very precise syntax of Java, we obtained a *program*.

Quick Exercise 1.5

Rewrite the maze-walking program using the "walk hugging the wall to your left" algorithm.

This example illustrates what beginners find to be perhaps the most difficult and even disconcerting aspect of programming: It requires a degree of precision quite unknown in ordinary human activities. The *algorithm* must be correct and carefully designed; in real life, algorithms are rarely written with the same level of precision—anyone who has ever tried to follow a recipe that says "Season to taste" knows that. The *program*—the written expression of the algorithm—must be written in the language that the mouse understands, with parentheses, brackets, semicolons, and so on, in just the right places.

In this book, we teach you how to design algorithms and how to express them in the Java language. To beginners, mastering the syntax of Java seems like the hard part, but as you get more practice, the syntax will become second nature. The development of algorithms, on the other hand, is a deep intellectual challenge and always will be. Indeed, the study of algorithms is one of the major research areas in computer science. This book contains many algorithms. Most are simple, like the mouse's algorithm, but a few are famously ingenious (e.g., the "quicksort" algorithm, page 550), and some are even historic (e.g., Gaussian Elimination, page 558).

While you are learning the Java syntax and learning how to design algorithms for a variety of problems, we hope you will develop a sense of *programming style*. Style is the intangible quality that makes programs easy to read, elegant, and even admirable. Donald Knuth of Stanford University, one of the leading practitioners of the art of algorithm design, has written, "The chief goal of my work as educator and author is to help people learn to write *beautiful programs*."[1] We cannot claim that you will be writing beautiful programs when you have finished this book, but we hope that you will at least understand what Knuth means.

1.2 Object-Oriented Programming

Java is one of the modern programming languages that encourages the use of *object-oriented programming* (*OOP*), which is a way of organizing programs. In object-oriented programming, a program is structured as a collection of *classes*, where each class describes a type of *object*. The objects represent the entities naturally occurring in the program, like the mouse and the maze.

object-oriented programming

classes

Every Java program is nothing but a collection of classes. When a Java program runs, it uses these classes to create a collection of *objects* that interact with one another by sending *messages*. For example, if we wrote the mouse-in-the-maze program in object-oriented form, the mouse would be an object (Figure 1.2), and so would the maze. The mouse would exchange messages with the maze to

objects

messages

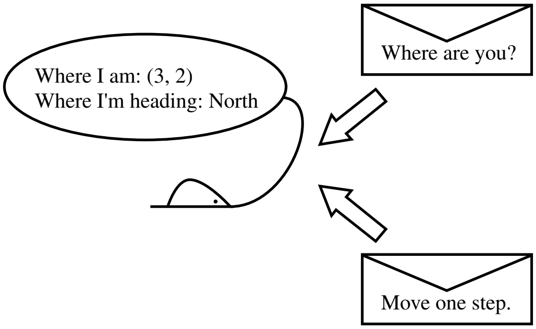


FIGURE 1.2 A mouse object receiving messages.

[1] Computer Programming as an Art, 1974 ACM Turing Award Lecture.

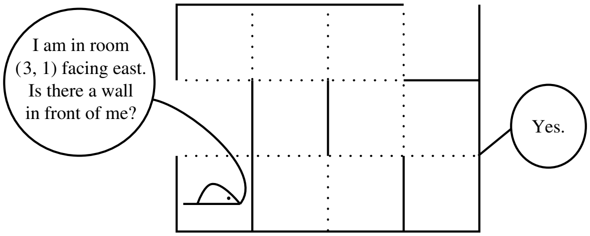

I am in room (3, 1) facing east. Is there a wall in front of me?
Yes.

FIGURE 1.3 The mouse in the maze in object-oriented form. The algorithm is the same as before, but now the mouse and the maze are both objects that communicate by sending messages.

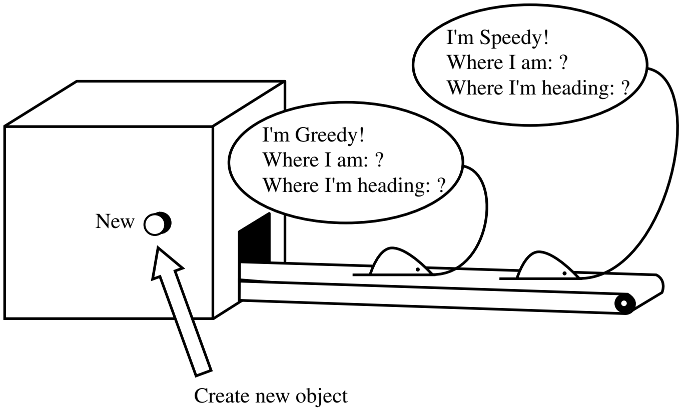

I'm Speedy! Where I am: ? Where I'm heading: ?
I'm Greedy! Where I am: ? Where I'm heading: ?
New
Create new object

FIGURE 1.4 Relationship between a class and its objects.

determine where it can move, as illustrated in Figure 1.3. Note that the mouse still follows an algorithm—in this case, the "wall to the right" algorithm—but the way we think about and organize the program is quite different.

The main job of the Java programmer, then, is to describe the objects that need to be created when the program runs. This is done by writing a class for each type of object, which we can think of as an *assembly line* for that type of object. The relationship between classes and objects is illustrated in Figure 1.4. For the mouse in the maze, the programmer would write a `Mouse` class and a `Maze` class. (It is conventional in Java to capitalize the names of classes.)

To be more specific, the Java programmer would create two classes by entering the following:

```
public class Mouse {
    we'll see what goes inside the class later in the book
}

public class Maze {
    ditto
}
```

Then, in a different part of the program, the programmer would write something like this:

```
Mouse righty = new Mouse();      ←  create Mouse object
Maze bigmaze = new Maze();       ←  create Maze object
righty.enter(bigmaze);           ←  tell mouse to enter maze
righty.getout();                 ←  tell mouse to find a way out
```

These four lines, or *statements*, are executed in sequence. The first two create the object described by the classes we defined above and give the objects the names `righty` and `bigmaze`, respectively. The third line "sends the `enter` message" to the mouse object and provides the maze as an *argument*; this tells the mouse object to enter the maze. The last line tells the mouse to find a way out of the maze.

The use of object-oriented structure is pervasive in Java. This structure has many advantages: It makes it easy to create multiple objects of the same type; for example, we could have two mice finding their way through the maze at the same time. It also makes it easy to create a variety of *similar* objects; for example, we could create one kind of mouse that follows the "wall to the right" algorithm and another that follows the "wall to the left" algorithm, without having to rewrite the first mouse class completely. Using object-oriented programming helps to make programs simpler to understand and more reliable, because it prevents one kind of object from knowing more than necessary about the structure of other kinds of objects. We will begin using objects in the very first real Java program we write in Chapter 2, and very shortly after that we will begin writing our own classes.

1.3 Computers and Data Representations

Although computers are amazingly complicated machines, the basic principles upon which they operate are fairly simple. Knowing something about these principles will help you understand how programming languages such as Java fit into the picture. It is especially useful to know something about how the computer represents different types of data: text, pictures, sound, and so on.

1.3.1 Bits, Bytes, and Binary Numbers

To a computer, everything is a number. Numbers are used to represent all kinds of data: pictures, sounds, characters, and, of course, numbers themselves. Numbers are also used to represent your program inside the computer as the computer executes it.

Throughout this book, we use the decimal representation of numbers, just as we all do in everyday life. Recall that our normal notation for numbers is called

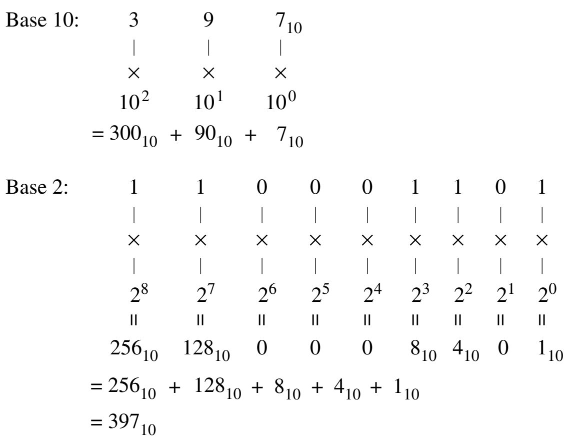


FIGURE 1.5 Positional notation.

positional notation. The value of each digit depends upon its position within the number. Thus, the four numerals 5427 represent the value $5 \times 1000 + 4 \times 100 + 2 \times 10 + 7$. In general, the digit that is i positions from the right is multiplied by 10^i.

binary

bits

Numbers in positional notation are so convenient that their invention is considered one of the greatest technical innovations in human history. *Base 10*, on the other hand, is a mere accident. Any number can be used instead of 10. In building computers, *binary*, or *base 2*, representation is most efficient. In base 2, there are two digits—0 and 1—which are called *bits* (for binary digit). In general, the digit that is i positions from the right is multiplied by 2^i. For example, the number 1010100110011 in base 2 has the value: $1 \times 4096 + 0 \times 2048 + 1 \times 1024 + 0 \times 512 + 1 \times 256 + 0 \times 128 + 0 \times 64 + 1 \times 32 + 1 \times 16 + 0 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$, which happens to add up to the same number as 5427 in base 10. The translation from binary to decimal is further illustrated in Figure 1.5.

Quick Exercise 1.6

Convert the following binary numbers to decimal:

1. `1010101`
2. `1010110`
3. `1111111`

word

The use of binary rather than decimal in computers is confusing at first, but is not of fundamental significance. More important is that numbers in computers are of *fixed size*. In most computers, each integer is restricted to 32 bits (commonly called a *word*). If you do the conversion $(1 \times 2^{31} + \cdots + 1 \times 2^0)$, you will see

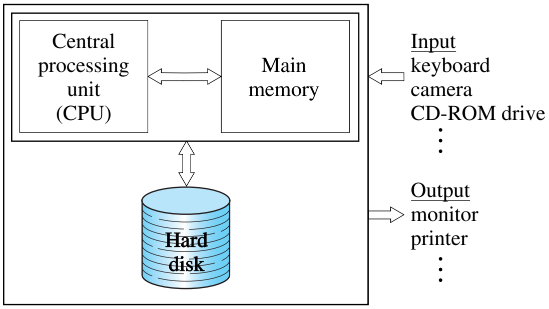


FIGURE 1.6 Block diagram of a computer. Computation is done by the CPU working with program and data in main memory; data move between these components at extremely high speed. New programs and data can be brought in from the hard disk; it is much larger and much slower than the main memory.

that this allows for numbers ranging from 0 up to a maximum of 4,294,967,295. This range is quite large enough for most purposes, so the restriction is rarely a problem in practice. However, it can result in some puzzling results on occasion. An example is given in Chapter 2.

Computer memories are commonly divided into 8-bit numbers called *bytes*. Thus, a word consists of 4 bytes. In describing the size of computer memories, one commonly uses the terms *megabyte* (2^{20}, or a little more than a million, bytes) and *gigabyte* (2^{30}, or a little more than a billion, bytes).

bytes

megabyte

gigabyte

1.3.2 Computer Organization

Internally, a computer consists of three major components: a central processing unit, or *CPU*, that does all the actual work of executing the program; a *main memory* that holds all the data and the program while it is executing; and a long-term storage device, usually a *hard disk*, that holds the programs that are not currently executing and the data that are not currently being used. See Figure 1.6. (Think of the CPU as an office worker, the main memory as the worker's desk, and the hard disk as the items on her or his shelves. All real work is done while sitting at the desk, but items are frequently pulled down off the shelves and placed back onto them.) There are numerous additional parts, called *input/output devices*, that let the computer communicate with the outside world (without which the computer is pretty useless). The three we've mentioned are the key internal components.

CPU

main memory

hard disk

input/output devices

The main memory, unlike an average desktop, is not a disorganized collection of documents. Rather, it is a long list of binary numbers; in most modern computers, it is a list of bytes. Each byte can have a different value, and the CPU can tell the memory to change the value of any byte in memory. To do this,

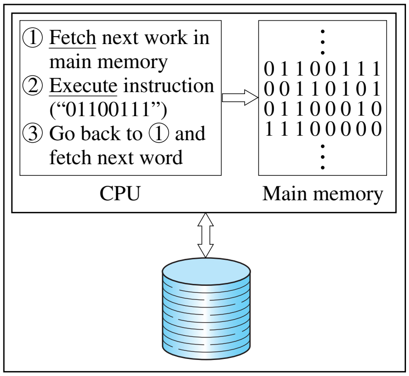

① Fetch next work in main memory
② Execute instruction ("01100111")
③ Go back to ① and fetch next word
CPU
01100111
00110101
01100010
11100000
Main memory

FIGURE 1.7 The fetch/execute cycle. The CPU repeats this cycle, fetching its instruction from the next memory location, until the program is finished.

address

each byte has an *address*, namely, its position in the memory. Thus, the CPU can direct the main memory to change byte 345 (the 346th byte in memory, since the first address is 0) to 45 (that is, 00101101). Location 345 will then continue to hold number 45 until the CPU tells it to change to something else. The CPU can also direct the main memory to return to it the number stored at a given location (just as you recover the value stored in the memory of a calculator). On most computers, the CPU actually reads and writes entire words (groups of 4 bytes) at once; this is more efficient than reading and writing individual bytes.

Nowadays, main memories are quite large. Their sizes are usually measured in megabytes. Thus, a 64-megabyte computer has 64×2^{20}, or about 64 million, bytes, or about 16 million words, or about 512 million bits. Disk memories are much larger. New personal computers (PCs) have disks that hold at least 10 gigabytes, about 10 billion bytes.

The action of the computer when it runs a program can be summarized roughly as follows: First, the program and all the data it uses must be loaded from the hard disk into the main memory. The program is placed in one part of the main memory and the data are placed in another; keep in mind that both the program and the data are nothing but a sequence of numbers. The CPU goes to the location of the first word of the program and reads that word. Each different number—that is, each different pattern of bits—tells the CPU to do something different. It may tell the CPU to load a number from a certain location in the data area and increment it, or to place a character at a certain location on the computer's screen, or to read a number from the disk and place it into a certain location in memory. The CPU performs that action and then goes to the next location in the program area of memory, reads that number, and executes it. This is known as the *fetch/execute cycle* (Figure 1.7).

fetch/execute cycle

Computers can do amazing things, but only by performing many, many tiny actions, one at a time. Even relatively inexpensive home computers nowadays can run through the fetch/execute cycle about 100 million times per second.

The correspondence between the number in a word and the action that the CPU takes when it sees that number is determined by the type of CPU. It is called the *machine language* of the processor. All PC-compatible machines have the same machine language, based on the Intel x86 line of CPUs,[2] so programs can be run on any such machine regardless of the manufacturer. On the other hand, Apple Macintoshes have been built with two different processors. Programs from earlier Macs, based on the Motorola 680x0 CPUs, will not run on today's Macs, which use the Motorola PowerPC. None of the programs, from either generation, will run on a PC-compatible computer. It is possible to write programs directly in machine language, but this is much more difficult than writing them in Java, and, of course, a program written in machine language can run on only one type of computer.

machine language

The main thing to remember is that only programs in machine language can be executed by the computer, which raises the question: How does a Java program become a machine language program? We discuss that in Section 1.4.

1.3.3 Data Representations

You might ask, If all the CPU can do is to take numbers and perform arithmetic operations on them, how can it print documents, or play music, or perform computer animations? What do documents or music or animated pictures have to do with numbers?

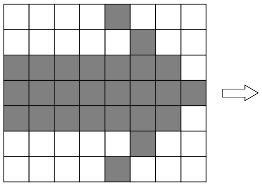
Indeed, in the early years of computation, all that computers were ever used for was numerical calculations. But it gradually became clear that numbers could be used to represent just about any kind of information:

Text. This is easy: Just choose a correspondence between *numbers* and *characters*. We can represent an entire book in memory by placing the characters of the book in sequence. Furthermore, since normal writing in English uses fewer than 256 characters (including letters, digits, and all the usual punctuation marks), we can represent each character in a single byte. (Shakespeare's complete works have a total of about 5 million characters; at 1 byte per character, we can easily fit them into the main memory of a modern computer; in fact, we could fit 10 copies or more.)

Quick Exercise 1.7

The most common code for characters used in English is ASCII, an abbreviation of American Standard Code for Information Interchange. Look up "ASCII" on the Internet. What are the codes for the characters "`a`", "`z`", "`A`", "`Z`", "`0`", "`9`", and "`&`"?

[2] Actually, the machine languages for the newer processors, such as the Pentium 4, are somewhat different from the machine languages of earlier processors, such as the 8086. Programs being written today may not run on CPUs of an earlier generation.



```
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1  =  8, 4, 254, 255, 254, 8, 4
1 1 1 1 1 1 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
```

FIGURE 1.8 Representation of black-and-white pictures using numbers. Here, an arrowhead is drawn on a screen with 7 × 8 = 56 pixels. (Real screens have sizes closer to 1000 × 1000 = 1,000,000 pixels, so they need many more numbers and produce much better images.)

pixel

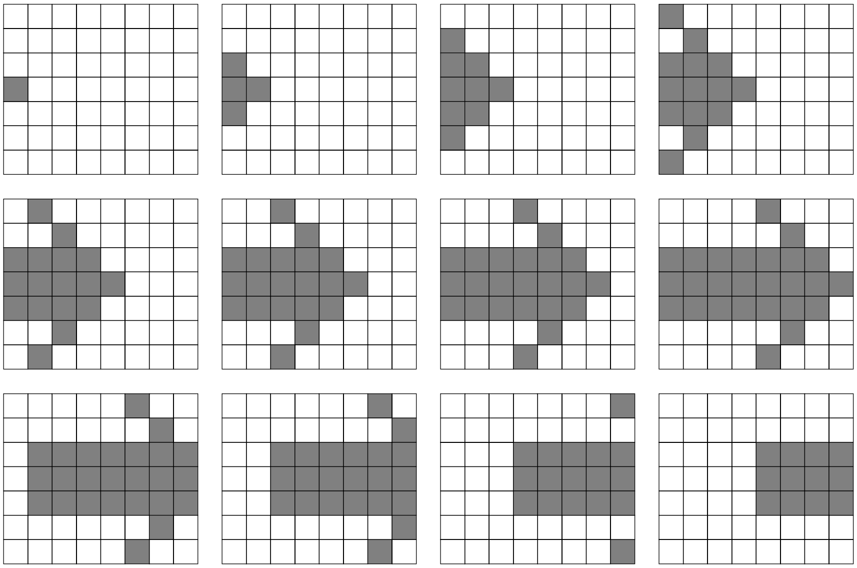
Pictures. Most readers probably already know the answer to this one. A picture can be reduced to a collection of dots, or *pixels*. This is how computer screens and computer printers work. In the simple case of a pure black-and-white picture, each pixel is either black or white, so it can be represented by a single bit. See Figure 1.8. A 1000 × 1000 image, which is quite a detailed image, can thus be represented in 1,000,000 bits, or 125,000 bytes. If the image is "gray scale" or color, it will need several bits per pixel; so the memory requirement will go up substantially, but the principle is the same. Modern digital cameras represent pictures using as many as 3000 × 2000, or more than 6 million, color pixels.

Animations. Since an animation is just a sequence of pictures, we can represent animations easily. Memory space will be stretched, and we might have to apply some cleverness to "compress" the images. But the point is that animations can be represented using numbers (Figure 1.9).

Sounds. You have undoubtedly seen pictures of sound waves (Figure 1.10*a*). Such curves can be *digitized*—turned into sequences of numbers—very easily: Just record the numerical values at regular intervals. If the interval is small enough, you can get as faithful a copy of the curve as you could possibly want. Hardware devices known as analog-to-digital (A/D) converters can perform this sampling (Figure 1.10*b*), and digital-to-analog (D/A) converters can convert this sequence of numbers back to the wave and thereby reproduce the sound. This is exactly how digital music formats like compact disks (CDs) work.

files

Data stored on your computer are kept on the hard disk in *files*. Each file is a collection of bytes, with a name and a type. See Figure 1.11. Thus, a file containing a picture might be called `my_new_car.jpg`. On many systems, the type of a file is indicated by the last part of its name (`jpg`, in this case). Every file takes up some space on the hard disk, and, of course, a large file takes up more space than a small one. As the discussion we've just had might suggest,



Animation is represented as a sequence of pictures. Each picture is a sequence of numbers. First picture: 0, 0, 0, 128, 0, 0, 0; Second picture: 0, 0, 128, 192, 128, 0, 0; Third picture: 0, 128, 192, 224, 192, 128, 0. Place two numbers at beginning, giving height and width of each picture: 7, 8, 0, 0, 0, 128, 0, 0, 0, 0, 0, 128, 192, 128, 0, 0, . . .

FIGURE 1.9 Representations of animations using numbers.

files containing sound and animations tend to be very large, while files containing text are usually of more modest size.

1.4 Compilers

Let's talk about the mechanics of programming. After you have designed the algorithm and written the program on paper, you begin the process of entering it into a file on the computer's hard disk. Entering and modifying your program in a file are done using a *text editor*, or just *editor*, a program provided to you that permits the storage and retrieval of what you have written on the disk. Learning how to use an editor well can make the job of entering a program much easier.

text editor

The program you have entered is written in Java. Java is not the same as the machine language of your computer. Rather, it is a *high-level language* designed

high-level language

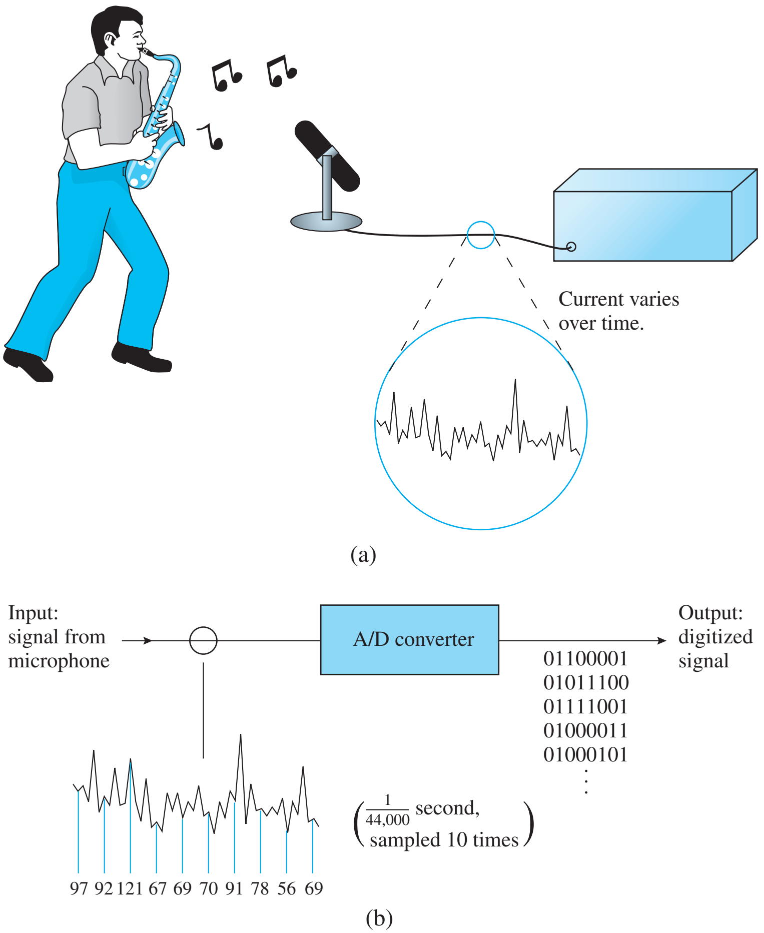


FIGURE 1.10 (a) Sound is carried through the air. Microphone turns sound into electrical impulses of varying strength. (b) Analog-to-digital (A/D) converter samples signal at regular intervals ($\frac{1}{44,000}$ second intervals in this illustration). The sampled values (in binary, of course) are sent to the device that will process the digitized signal.

to make programming easier than using machine language. Most programming is done in such high-level languages; other examples of high-level languages include C, C++, and Visual Basic. However, as we have indicated above, only machine language programs can be executed by a computer. Somehow, the Java program needs to be translated to machine language. In the very early history of computers, programmers might do such a translation by hand, but now it is done by another program called a *compiler* (see Figure 1.12). The Java program you

compiler

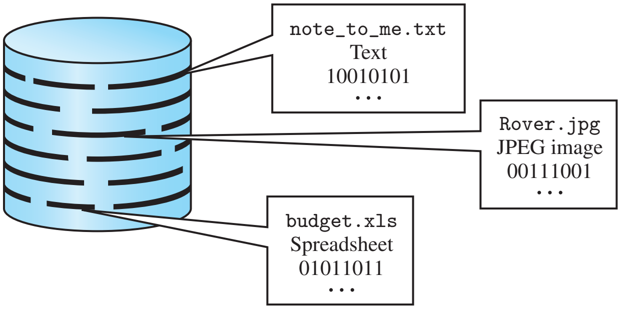


FIGURE 1.11 **Files occupy space on the hard disk. Files consist of many binary digits. A file also has a name and a type. In most systems, the type is given as the latter part of the name; for example, "xls" indicates a spreadsheet created by Excel.**

enter is called the *source code*; the compiler translates it into machine language and stores the result in another file; it is called the *object code*. The object file can be loaded into the computer's main memory and executed by the CPU.

source code

object code

In summary, to create and run a Java program, you need to enter the program using a text editor; execute a special program, the compiler, to translate the Java to machine language; and then load the machine language into main memory and execute it. The process is illustrated in Figure 1.12.

Actually, this process doesn't apply to Java in exactly the same way as it does to most high-level languages. Java is different because one of the goals of the Java language is to be *platform-independent*, that is, to be able to execute on a wide variety of different types of machines *after* it has been compiled. It follows that we cannot translate it to machine language, because each program in machine language only runs on one type of machine. Instead, Java is translated into an intermediate-level language called *Java Bytecode*. Java Bytecode is subsequently interpreted in the exact same way on every computer, so that a Java program does the same thing whether run on an IBM PC, an Apple Macintosh, or whatever.

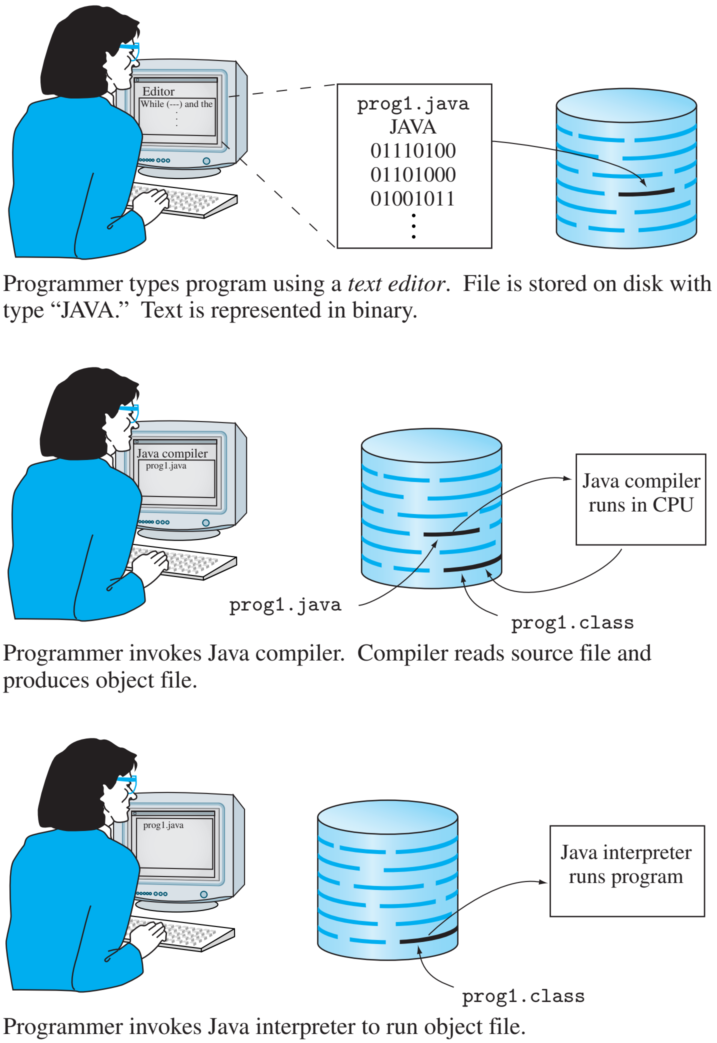
platform-independent

Java Bytecode

So the process of creating and executing a Java program is modified a bit: Once you have the file containing the Java program, you use a compiler to translate it to a program in the Java Bytecode language, and then you use another special program, called an *interpreter*, to *execute* the Java Bytecode. A particularly attractive feature of Java is that many modern Internet browsers (such as Netscape Navigator and Microsoft's Internet Explorer) incorporate Java interpreters.

interpreter

The precise steps you will follow—what editor you use, how to save files, how to invoke the compiler, etc.—depend on the system you are using. The simplest case occurs when you are writing an application on a *command-oriented* system (such as DOS or UNIX). In this case, you place your program in a file called `whatever.java` (`whatever` can be anything you choose, but `java`



Programmer types program using a *text editor*. File is stored on disk with type "JAVA." Text is represented in binary.

Programmer invokes Java compiler. Compiler reads source file and produces object file.

Programmer invokes Java interpreter to run object file.

FIGURE 1.12 Entering, compiling, and executing a Java program.

must be `java`). Then type the command

```
    javac whatever.java
```

to *compile* the program, and then

```
    java whatever
```

to execute it.

Again, you must always *compile* your program first. You can then execute it (as many times as you like).

1.5 Debugging

In reality, the "enter/compile/execute" process rarely runs as smoothly as we've described it. In fact, most of the time the program you enter will fail either to compile correctly or to execute correctly. In that case, you will have to go back to the editor and fix the problem, then again compile and execute. The creation of programs is referred to as the *edit/compile/execute cycle*, because those three steps are repeated many times.

edit/compile/execute cycle

syntactic errors

Your program will fail to compile if it has *syntactic errors*, meaning that it fails to adhere to the grammatical rules of Java. The compiler will be unable to "understand" your program, and it will tell you so, usually in a terse, unfriendly way. Something as simple as omitting a semicolon or entering a letter in uppercase when it should be in lowercase, or vice versa, can easily cause the compiler to reject your program. You must figure out what needs to be changed, use the editor to change it, and then try compiling again. After a few iterations of this process, the program will compile properly into Java Bytecode form.

run-time errors

logic errors

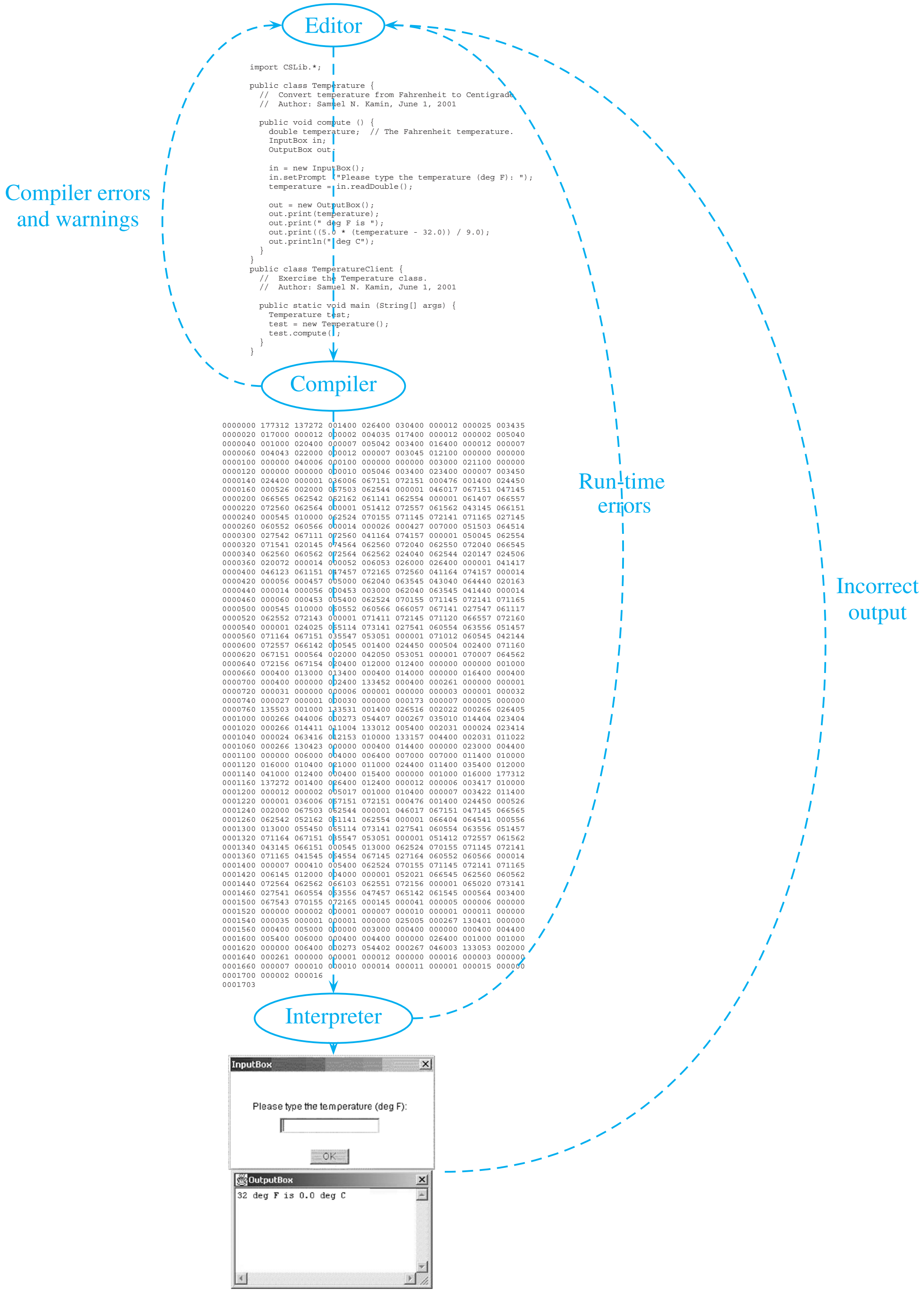
debugging

Just because the program compiles without errors does *not* guarantee that it will work correctly! You may have written something that is grammatically correct but that doesn't do at all what you expected. Errors that occur during the execution of the program are called *run-time errors* or *logic errors*; they need to be fixed, too, just as the syntax errors do. Run-time errors can result from simple typing mistakes or from flaws in the algorithm. In either case, it's back to the editor to revise your program and back to the compiler until the algorithm is correct, your program compiles, and finally it executes to completion. Did it get the right answers? If so, try enough additional data to be sure it really works; if not, find the errors, correct them, and recompile and rerun the program. This whole process of removing bugs (errors) from a program is called *debugging*.

The edit/compile/execute cycle is illustrated in Figure 1.13. Just below the Editor is a Java program. The Compiler translates this Java program into Java Bytecode, shown in the figure as a sequence of numbers, which, of course, it is. The Java Bytecode produced by the compiler is then loaded and interpreted by the Java Interpreter, producing the output shown.

integrated development environment

If you're working in an *integrated development environment*, or *IDE*, instead of a command-oriented system, you're lucky! An IDE allows you to edit, compile, and run programs by clicking on command buttons, rather than by typing commands. The procedures for compiling and executing programs will differ from those just described for command-oriented systems; consult the manual or your local guru for details.



```
import CSLib.*;

public class Temperature {
  //  Convert temperature from Fahrenheit to Centigrade
  //  Author: Samuel N. Kamin, June 1, 2001

  public void compute () {
    double temperature;  // The Fahrenheit temperature.
    InputBox in;
    OutputBox out;

    in = new InputBox();
    in.setPrompt ("Please type the temperature (deg F): ");
    temperature = in.readDouble();

    out = new OutputBox();
    out.print(temperature);
    out.print(" deg F is ");
    out.print((5.0 * (temperature - 32.0)) / 9.0);
    out.println(" deg C");
  }
}
public class TemperatureClient {
  //  Exercise the Temperature class.
  //  Author: Samuel N. Kamin, June 1, 2001

  public static void main (String[] args) {
    Temperature test;
    test = new Temperature();
    test.compute();
  }
}
```

```
0000000 177312 137272 001400 026400 030400 000012 000025 003435
0000020 017000 000012 000002 004035 017400 000012 000002 005040
0000040 001000 020400 000007 005042 003400 016400 000012 000007
0000060 004043 022000 000012 000007 003045 012100 000000 000000
0000100 000000 040006 000100 000000 000000 003000 021100 000000
0000120 000000 000000 000010 005046 003400 023400 000007 003450
0000140 024400 000001 036006 067151 072151 000476 001400 024450
0000160 000526 002000 067503 062544 000001 046017 067151 047145
0000200 066565 062542 052162 061141 062554 000001 061407 066557
0000220 072560 062564 000001 051412 072557 061562 043145 066151
0000240 000545 010000 062524 070155 071145 072141 071165 027145
0000260 060552 060566 000014 000026 000427 007000 051503 064514
0000300 027542 067111 072560 041164 074157 000001 050045 062554
0000320 071541 020145 074564 062560 072040 062550 072040 066545
0000340 062560 060562 072564 062562 024040 062544 020147 024506
0000360 020072 000014 000052 006053 026000 026400 000001 041417
0000400 046123 061151 047457 072165 072560 041164 074157 000014
0000420 000056 000457 005000 062040 063545 043040 064440 020163
0000440 000014 000056 000453 003000 062040 063545 041440 000014
0000460 000060 000453 005400 062524 070155 071145 072141 071165
0000500 000545 010000 060552 060566 066057 067141 027547 061117
0000520 062552 072143 000001 071411 072145 071120 066557 072160
0000540 000001 024025 065114 073141 027541 060554 063556 051457
0000560 071164 067151 035547 053051 000001 071012 060545 042144
0000600 072557 066142 000545 001400 024450 000504 002400 071160
0000620 067151 000564 002000 042050 053051 000001 070007 064562
0000640 072156 067154 020400 012000 012400 000000 000000 001000
0000660 000400 013000 013400 000400 014000 000000 016400 000400
0000700 000400 000000 002400 133452 000400 000261 000000 000001
0000720 000031 000000 000006 000001 000000 000003 000001 000032
0000740 000027 000001 000030 000000 000173 000007 000005 000000
0000760 135503 001000 133531 001400 026516 002022 000266 026405
0001000 000266 044006 000273 054407 000267 035010 014404 023404
0001020 000266 014411 011004 133012 005400 002031 000024 023414
0001040 000024 063416 012153 010000 133157 004400 002031 011022
0001060 000266 130423 000000 000400 014400 000000 023000 004400
0001100 000000 006000 004000 006400 007000 007000 011400 010000
0001120 016000 010400 021000 011000 024400 011400 035400 012000
0001140 041000 012400 000400 015400 000000 001000 016000 177312
0001160 137272 001400 026400 012400 000012 000006 003417 010000
0001200 000012 000002 005017 001000 010400 000007 003422 011400
0001220 000001 036006 067151 072151 000476 001400 024450 000526
0001240 002000 067503 062544 000001 046017 067151 047145 066565
0001260 062542 052162 061141 062554 000001 066404 064541 000556
0001300 013000 055450 065114 073141 027541 060554 063556 051457
0001320 071164 067151 035547 053051 000001 051412 072557 061562
0001340 043145 066151 000545 013000 062524 070155 071145 072141
0001360 071165 041545 064554 067145 027164 060552 060566 000014
0001400 000007 000410 005400 062524 070155 071145 072141 071165
0001420 006145 012000 004000 000001 052021 066545 062560 060562
0001440 072564 062562 066103 062551 072156 000001 065020 073141
0001460 027541 060554 063556 047457 065142 061545 000564 003400
0001500 067543 070155 072165 000145 000041 000005 000006 000000
0001520 000000 000002 000001 000007 000010 000001 000011 000000
0001540 000035 000001 000001 000000 025005 000267 130401 000000
0001560 000400 005000 000000 003000 000400 000000 000400 004400
0001600 005400 006000 000400 004400 000000 026400 001000 001000
0001620 000000 006400 000273 054402 000267 046003 133053 002000
0001640 000261 000000 000001 000012 000000 000016 000003 000000
0001660 000007 000010 000010 000014 000011 000001 000015 000000
0001700 000002 000016
0001703
```

Please type the temperature (deg F):

```
32 deg F is 0.0 deg C
```

FIGURE 1.13 The cycle of editing, compiling, and debugging a program. The Java program shown is the temperature program from Section 3.3 (pages 67 and 68).

1.6 Applications and Applets

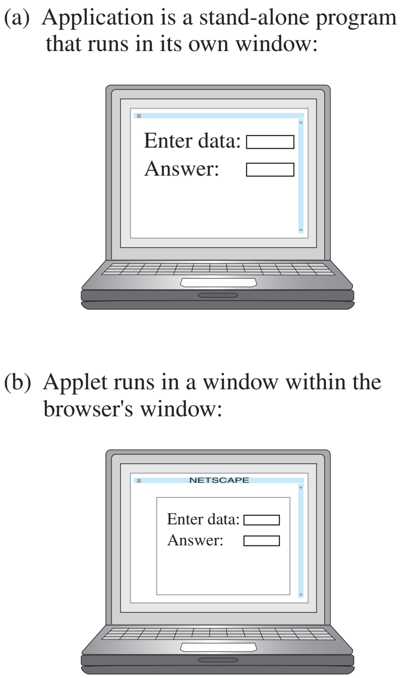
applications

In Java, you can write two kinds of programs: applications and applets. *Applications* are stand-alone programs that run on your own computer. They are no different from the kinds of programs that you might write in other programming languages, such as C++. Applications can read and manipulate data stored on your computer's disk, as well as data that you enter through the keyboard. Applications can also write data on your disk, display output on your screen, and print on your printer. All the programs you normally run, including word processors, spreadsheet programs, and games, are applications.

applets
browser

Unlike applications, *applets* are executed from within a *browser*, such as Netscape Navigator or Internet Explorer. Once you've written an applet, not only can you execute it from within your browser, but also you can make it available for others to execute from within *their* browsers. Correspondingly, you can execute applets that other programmers have written. The ability to execute applets really brings the World Wide Web to life, and it is its ability to write applets that has made Java the sensation that it is.

The difference between applications and applets is illustrated in Figure 1.14.



(a) Application is a stand-alone program that runs in its own window:

(b) Applet runs in a window within the browser's window:

FIGURE 1.14 Applications versus applets.

We will teach you how to write both applications and applets. Applets are more fun, because you can easily show your work to the entire world. On the other hand, since creating and running applets require slightly more work than creating an application (for example, you have to create a web page to run an applet), we will begin by focusing on applications. Applets are covered in detail in Chapter 13. In any case, the differences between them are mainly superficial; an application and an applet, if they do the same computation, will differ only slightly.

Summary

The computer can do extraordinary things in the hands of a skillful programmer. By now you should have some idea of the steps involved in programming. The *problem* must first be clarified and an appropriate *algorithm* developed; the algorithm must be stated in a specific computer language, such as Java, to become a *program*. Object-oriented programming is a modern paradigm that allows the programmer to model the problem in a real-world fashion. Objects interact with one another by sending messages.

The most important internal parts of a computer are the *central processing unit*, *main memory*, and *hard disk*. In addition, *input/output devices* allow the computer to receive input from, and provide results to, the outside world. A program executes only when it and all the data it needs are in the main memory; programs and data can be read from the hard disk when needed, and results can be written out to the hard disk.

Computers use *binary numbers*. Numbers can be used to represent any conceivable kind of data. Text is represented as a sequence of numbers by choosing a code for each letter. Pictures are represented by dividing them into numerous *picture elements*, or *pixels*, and representing each pixel by a number. Video images are represented by placing numerous pictures one after another. Sounds are represented by *sampling* the sound wave at regular intervals. All data are stored in *files* on the hard disk.

To run a program, enter it into a file using an *editor*, then *compile* it to transform it to machine language. *Debugging* is the process whereby errors in a program are detected and fixed. Errors can be *syntactic*, meaning that the compiler cannot understand the program; or they can be *logical*, meaning that the program compiles to machine language but an error occurs when it is executed. Errors that occur during execution of a program will sometimes cause the program to stop prematurely with an (obscure, usually) error message; other times, they will simply give incorrect results with no warning. Programs must be thoroughly tested and debugged to ensure that neither of these errors can occur.

In Java parlance, *applications* are ordinary programs that are run from a computer's desktop, and *applets* are programs that run within a window inside a web browser. Java can be used to write either kind of program.

Exercises

1. To familiarize yourself with the notion of an *algorithm*, write specific, detailed, procedures for the following activities. Use the Java "if" and "while" constructs where appropriate.
   (a) Fill your car with gas.
   (b) Place the initial set of red and black pieces on a standard 8 × 8 checkerboard.
   (c) Generalize the algorithm in part (b) so that it works for a checkerboard of any given size n, so long as n is even.
   (d) Add two three-digit decimal numbers.
   (e) Generalize the algorithm in part (d) to add two decimal numbers of any length n.

2. The central programming structure in any object-oriented language is the *class*. However, some languages use different terminology. Use the Internet to look up each of the following computer languages. For each, determine whether or not it is an object-oriented language and, if it is, the term used for what Java calls a class:
   (a) Fortran
   (b) Cobol
   (c) C
   (d) C++
   (e) JavaScript
   (f) Eiffel
   (g) Visual Basic

3. Using the Internet, find a computer you can buy for between $900 and $1000. Describe the computer, giving the type of its CPU, the amount of main memory and the size of its hard disk.

4. In addition to binary (base 2) and decimal (base 10), computer programmers frequently use *octal* (base 8) representation. Numbers in octal use digits 0–7, and the multiplier for the i^{th} digit is 8^i. For example, $370_8 = 3 \times 8^2 + y \times 8^1 + 0 \times 8^0 = 248_{10}$.
   (a) Convert 370_8 to binary.
   (b) Convert 101111000_2 to octal.
   (c) Octal numbers are popular in computing because there is a simple way to convert them to and from binary. Can you find the algorithm? After doing several more conversions the hard way (translating to and from decimal), you should be able to see it.

5. Continuing with the previous exercise, there is another base commonly used in computing: base 16, or *hexadecimal*. Since this base needs 16 numerals, we use the orginary numerals 0–9 and the letters A–F for the numbers from 10 to 15. For example, $1FF_{16} = 511_{10}(1 \times 16^2 + 15 \times 16^1 + 15 \times 16^0)$. As with octal notation, the main attraction of hexadecimal is that it is easy to convert to and from binary. Redo part (c) of exercise 4 using hexadecimal in place of octal.

6. Although ASCII is a very widely used code for English text, it is quite limited because, as a one-byte code, it can represent only 256 different characters. Java uses the more modern Unicode representation, which is a two-byte (16-bit) code, allowing for 65,536 different characters. Find the Unicode code for these characters:

A: ordinary capital A
ç: c with “cedilla”
π: Lower-case Greek letter “pi”
£: British pounds sterling
¥: Japanese Yen symbol

7. A black-and-white image uses a single bit per pixel. A gray-scale image might permit 16 levels of gray (from pure white to pure black). Many computers allow three options for representing colors: 256 colors (one byte per pixel), “thousands of colors” (two bytes per pixel) and “millions of colors” (four bytes—one word—per pixel). If a 1000 × 1000-pixel image is stored in a computer, how much memory will it occupy if it is represented in each of these ways?