Chapter 1

An Introduction to Computers That Will Actually Help You in Life

- Memory: Not Exactly 0s and 1s
- Memory Organization
- A Very Simple Computer

Java is a programming language that tells computers what to do. This chapter will look at what computes really are, what they can do, and how we use programming languages to control them.

We will begin by exploding the common myth that computers deal only with 0s and 1s. Once we establish what computers really process, we will look at the kind of processing they perform.

This is emphatically *not* an intellectual exercise. Spending a bit of effort here will make your life much easier in the chapters that follow. Many concepts that appear later in this book, such as data typing, referencing, and virtual machines, will make very little sense unless you understand the underlying structure of computers. Without this understanding, learning to program can be confusing and overwhelming. With the right fundamentals, though, it can be enjoyable and stimulating.

Memory: Not Exactly 0s and 1s

No doubt you've heard that computers only process 0s and 1s. This can't possibly be true. Computers are used to count votes in elections, so they must be capable of counting past 1. Computers are also used to model the behavior of subatomic particles whose masses are tiny fractions, so they must be capable of processing fractions as well as whole numbers. They're used for writing documents, so they must be capable of processing text as well as numbers.

On the most fundamental level, computers do not process 0s and 1s, or whole numbers, or fractions, or text. Computers are electronic circuits, so all they really process is electricity. Computer components are designed so that their internal voltages are either approximately zero or approximately 5 or 6 volts. When part of a computer circuit carries a voltage of 5 or 6 volts, we say that it has a value of 1. When part of a circuit carries zero voltage, we say that it has a value of 0. (Fortunately, this is all the electronics knowledge you need to become a master programmer.)

It's all a matter of interpretation. Voltages are interpreted as 0s and 1s. As you'll see later in this chapter and in Chapter 2, "Data," the 0s and 1s are organized into clusters that are interpreted as numbers. More sophisticated parts of the computer interpret those numbers as codes that represent fractions, or text, or colors, or images, or any of the other myriad classes of objects that can be represented in a computer.

A modern computer contains billions of microscopic components, each of which has a value of 0 or 1. Any circuit where we only care about the approximate values of the voltages is known as a *digital circuit*. Computers that are made of digital circuitry are known as *digital computers*.

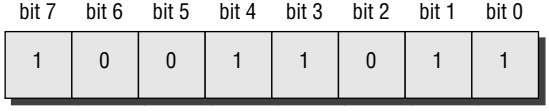
NOTE The opposite of digital is *analog*. In an analog circuit, we care about the exact voltages of the components. Analog circuits are ideal for certain applications, such as radios and microwave ovens, but they don't work so well for computers. Analog computers were used in the 1940s, but they were an evolutionary dead end. All modern computers are digital.

One simple but useful type of digital circuit is known as *memory*. A memory circuit just stores a digital value (0 or 1, because we programmers don't have to think about voltages). A single unit of memory is called a *bit*, which is an abbreviation for "binary digit." You can think of a bit as a microscopic box, the contents of which are available to the rest of the computer. From time to time the computer might change the contents. Bits are usually drawn as shown in Figure 1.1.

FIGURE 1.1:
A bit

Bits are usually organized in groups of eight, known as *bytes*. Figure 1.2 shows a byte that contains an arbitrary combination of 0s and 1s.

FIGURE 1.2:
A byte



Note that the individual bits are numbered from right to left, and that the numbering starts from 0. Computer designers always start numbering things from 0 rather than 1. This is true whether they are numbering bits in a byte, bytes in memory (as we are about to see), or components in an array (as we will see in Chapter 6).

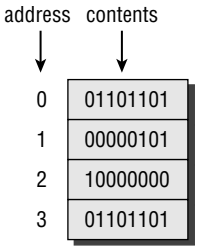
A byte can contain 256 combinations of bit values: 2 possibilities for bit #0 times 2 possibilities for bit #1 times 2 possibilities for bit #3, and so on up through bit #7.

If you looked at a computer through a microscope and saw the byte shown in Figure 1.2, you might wonder what value it contained. You would see the 0s and 1s, but what would they mean? It's a great question that has no good answer. A byte might represent an integral number, a fraction, part of an integer or fraction, a character in a document, a color in a picture, or an instruction in a program. It all depends on the byte's context. As a programmer, you are the one who dictates how each byte will be interpreted.

Memory Organization

Typically, a modern personal computer contains several hundred million bytes of memory. The prefix *mega* (abbreviated *M*) means million, so we could also say that a computer has several hundred megabytes or *MB*. Programs and programmers need a way to distinguish one byte from another. This is done by assigning to each byte a unique number, known as the byte's *address*. Addresses begin at 0. Figure 1.3 shows 4 bytes.

FIGURE 1.3:
Several bytes



If Figure 1.3 showed 512 MB and was drawn to the same scale, it would be about 2,000 miles high.

A single byte is not very versatile, because its value is limited to 256 possibilities. It doesn't matter whether the byte represents a number or a letter or anything else—in computer applications, 256 of *anything* isn't much of a range. For this reason, computers often use groups of bytes. Two bytes, taken together as a unit, can take on 256 times 256 possible values, or 65,536. Four bytes can take on 256 times 256 times 256 times 256 values, or 4,294,967,296. This is where it starts to be useful. Eight bytes can take on approximately 20 quintillion different values.

Memory is usually used in chunks of 1, 2, 4, or 8 bytes. (Later we will see that arrays and objects use chunks of arbitrary size.) The chunks can represent integral numbers, fractions, text, or any other kind of information. From this perspective, we can see that the statement "Computers only deal with 0s and 1s" is true only in a very limited sense.

Think of it this way: A computer is a digital circuit, and we think of its components as having values that represent 0s or 1s. But if we look one level below the digital components, we see only electricity, not numbers. And if we look one level above the digital components, we see that the bits are organized into chunks of 1 or more bytes that represent many types of information.

In addition to various types of data, memory can also store the instructions that operate on data. In the next section, we will look at a very simple computer and see how instructions and data interact.

A Very Simple Computer

This chapter will introduce a very simple computer called SimCom. SimCom is imaginary. Or, to use a more respectable term, it is *virtual*. Nobody has ever built a SimCom, but it is simulated in one of the animated illustrations on the CD-ROM.

The processors that power your own computer, the Pentiums, SPARCs, and so on, are not very different qualitatively from SimCom. *Quantitatively*, however, there is a huge difference: the real processors have vastly more instructions, speed, and memory. SimCom is as simple as a computer can be while still being a useful teaching tool.

The point of this section is not to make you a master SimCom programmer. The point is to use SimCom to introduce certain principles of programming. Later in this book, the same principles will be presented in the context of Java. These principles include

- High-level languages
- Loops
- Referencing
- Two's complement
- Virtual machines

In this section, you will see some typical processor elements that are quite low-level. Modern programming languages like Java deliberately isolate you from having to control these elements. However, it is extremely valuable to know that they exist and what they do on your behalf.

The architecture of SimCom is very simple. There is a bank of 32 bytes of memory; each byte can be used as an instruction or as data. There is one extra byte, called the *register*, which is used like scratch paper. Another component, called the *program counter*, keeps track of which instruction is about to be executed. Figure 1.4 shows the architecture of SimCom.

The arrow in the figure indicates the program counter. The next instruction to be executed will be byte #7. Note that byte addresses start at 0.

When SimCom starts up, it sets the program counter to 0. It then *executes* byte 0. (We'll see what this means in a moment.) Execution may change the register or a byte of memory, and it almost always changes the program counter. Then the whole process repeats: The instruction indicated by the program counter is executed, and the program counter is modified. This continues until SimCom is instructed to halt.

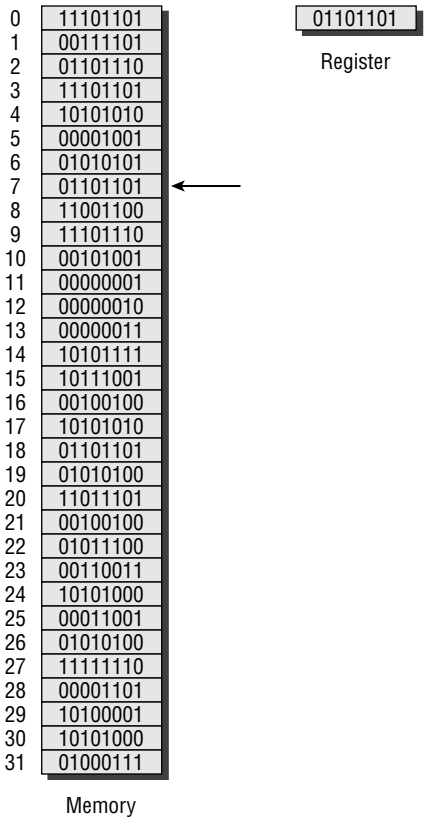


FIGURE 1.4:
SimCom architecture

Bits 7, 6, and 5 of an instruction byte tell SimCom what to do. They are known as the *operation code* or *opcode* bits. Bits 4 through 0 contain additional instructions; they are called the *argument* bits. This division of bits is shown in Figure 1.5.

The SimCom computer has 7 opcodes. They are shown in Table 1.1.

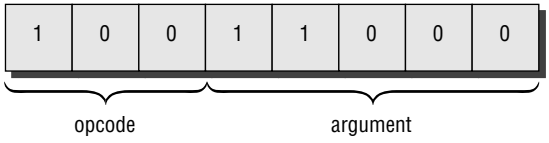


FIGURE 1.5:
Opcode and argument bits

TABLE 1.1: Opcodes

Opcode	Function	Abbreviation
000	Load	LOAD
001	Store	STORE
010	Add	ADD
011	Subtract	SUB
100	Jump to different current instruction	JUMP
101	Jump if register is zero	JUMPZ
110 or 111	Halt	HALT

The 5 argument bits contain a value that is the base-2 address of a memory byte. The LOAD opcode copies the contents of this address into the register. For example, suppose the current instruction is 00000011. The opcode is 000 (LOAD), and the argument is 00011 (which is the base-2 notation for 3). When the instruction is executed, the value in byte #3 is copied into the register. Note that the value 3 is *not* copied into the register. The argument is never used directly; it is always an address whose contents are used.

The STORE opcode copies the contents of the register in the memory byte whose address appears in the argument. For example, 00100001 causes the register to be copied into byte #1.

The ADD opcode adds two values. One value is the value stored in the byte whose address appears in the argument. The other value is the contents of the register. The result of the addition is stored in the register. For example, suppose the register contains 00001100, and byte #1 contains 00000011. The instruction 01000001 causes the contents of byte #1 to be added to the contents of the register, with the result being stored back in the register. Note that the argument (00001) is used *indirectly*, as an address. The value 00001 is not added to the register; rather, 00001 is the address of the byte that gets added to the register.

The SUB opcode is like ADD, except that the value addressed by the argument is subtracted from the register. The result is stored in the register.

After each of these four opcodes is executed, the program counter is incremented by 1. Thus, control flows sequentially through memory. The remaining three opcodes alter this normal flow of control. The JUMP opcode does not change the register or memory; it just stores its argument in the program counter. For example, after executing 10000101, the next instruction to be executed will be the one at byte 00101, which is the base-2 notation for 5.

The JUMPZ opcode inspects the register. If the register contains 00000000, the program counter is set to the instruction's argument. Otherwise, the program counter is just *incremented* (that is, increased by 1) and control flows normally. This is a very powerful opcode,

because it enables the computer to be sensitive to its data and to react differently to different conditions.

Finally, the HALT opcode causes the computer to stop processing.

Let's look at a short program:

```
00000100
01000100
00100100
11000000
```

The first thing to notice about this program is that it's hard to read. Let's translate it to a friendlier format:

```
LOAD  4
ADD   4
STORE 4
HALT
```

The program doubles the value in byte #4. It does this by copying the value into the register, then adding the same value into the register, and then storing the result back in byte #4.

This example shows that anything is better than programming by manipulating 0s and 1s. These spelled-out opcodes and base-10 numbers are a compromise between the binary language of computers and the highly structured and nuanced language of humans. The LOAD 4 notation is known as *assembly language*. In assembly language, a line of code typically corresponds to a single computer instruction, and the programmer must always be aware of the computer's architecture and state. An *assembler* is a program that translates assembly language into binary notation.

Playing with SimCom

Unfortunately we couldn't package a SimCom with every copy of this book, but we have done the next best thing. The first animated illustration on the book's CD is a simulation of a SimCom in action.

NOTE If you don't already have Java installed on your computer, now is the time. If you're not sure how, please refer to Appendix A, "Downloading and Installing Java," which walks you through the entire process. Throughout this book you will be invited to run an animated illustration program, and you will be given a command to type into your machine. It will all make sense after you go through Appendix A.

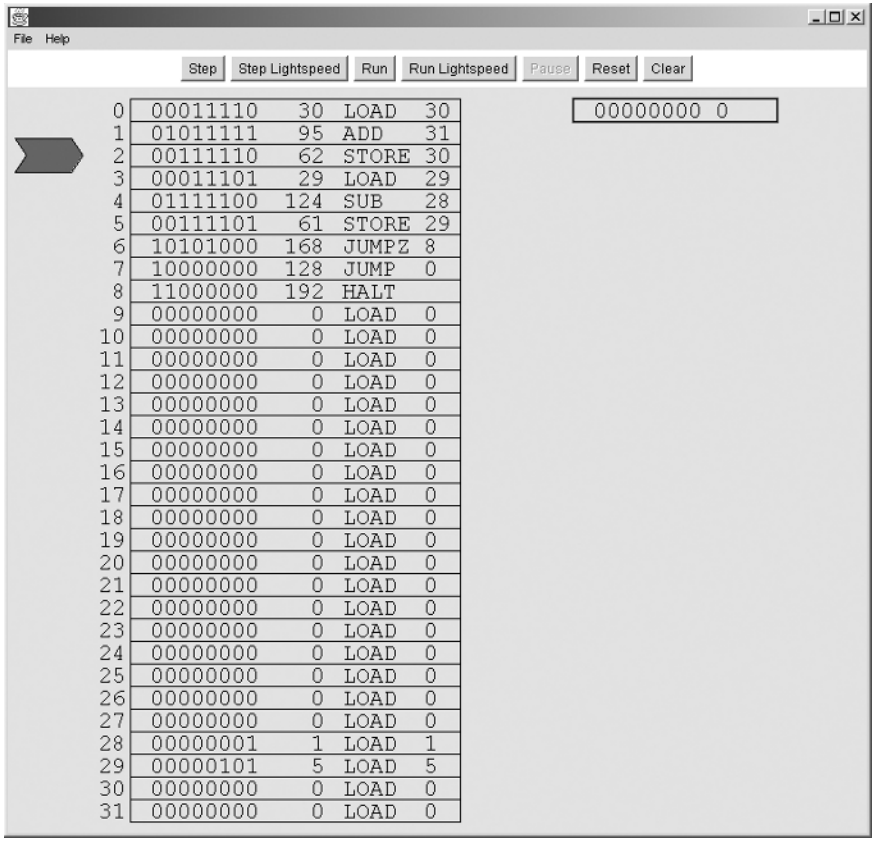
To run the SimCom simulation, type the following at your command prompt:

```
java simcom.SimComFrame
```

The simulation allows you to load and run preexisting programs or create your own programs. Figure 1.6 shows the simulation in action.

Each byte of memory is displayed in three formats: base-2, base-10, and opcode-plus-argument. The register is only displayed in base-2 and base-10; since the register is never executed, there is no value in displaying which instruction *would* be executed. You can change any byte in memory by first clicking inside that byte. This will highlight and select the byte for editing. Then, if you click on the base-10 region, you will get a panel that lets you select a new base-10 value. If you click on the opcode region, you will get a panel that lets you select a new opcode. To change the argument, first click on the argument region of the selected byte. As you move the mouse, the closest byte address will light up. When the address you want is highlighted, click on it to set it as the argument.

FIGURE 1.6:
SimCom in action



Try executing a very simple program. Click File ➢ Scenarios in the File menu, and select Load/Add/Store/. This program adds bytes 10 and 11 (not the numbers 10 and 11, but the contents of the memory bytes whose addresses are 10 and 11), and stores the result in byte 12. Initially, bytes 10 and 11 both contain zero, so to see interesting results you will have to change their values. To see the program in action, click the Step button. This executes the current instruction in slow motion. To run continuously, click the Run button, which plays the animation until a HALT instruction is executed. If you get tired of the slow motion, you can click Step Lightspeed or Run Lightspeed to get instant results. The Reset button reinitializes memory and sets the program counter to zero.

Try storing relatively large values in bytes 10 and 11. The largest value a byte can store is 255. What happens if you try to add 5 + 255?

Change the program so that byte 11 is subtracted from byte 10. What happens if byte 10 contains 5 and byte 11 contains 6?

When you are ready for a more interesting program, click Scenarios ➢ Times 5 in the File menu. This program multiplies the contents of byte 31 by 5 and stores the result in byte 30. Experiment with a few values in byte 31 to convince yourself that it works. Remember to click the Reset button after each run.

This program might seem needlessly complicated. It's too bad the SimCom instruction set doesn't include a multiply opcode, but since it doesn't, wouldn't the following program be more straightforward?

```
LOAD    31
ADD     31
ADD     31
ADD     31
ADD     31
STORE   30
HALT
```

This is definitely more straightforward, but it is also less flexible than the version SimCom uses. That version uses a *loop*, a powerful construct that appears in all programming languages. Note that initially, byte 29 contains 5; this byte is a *loop counter* that controls how many times the loop will be executed. Lines 0 through 3 add whatever is in byte 31 (the value to be quintupled) to whatever is in byte 30 (the accumulating result). Then lines 3 through 5 subtract 1 from the loop counter. If the loop counter reaches zero, line 6 causes a jump to a HALT instruction. If the decremented loop counter has not yet reached zero, line 7 causes a jump back to line 0, which is the beginning of the loop.

Reset the Times 5 program. Change the value in byte 29 (the loop counter) from 5 to 6. Put a reasonable value in byte 31 and run the program. Notice that the program now multiplies

by 6. This is to be expected, because the value in byte 31 has been added one extra time to the accumulated result.

Now you can see how the looping version is more flexible than the repeated-addition version shown earlier. To modify the looping version so that it multiplies by 10 instead of 5, you just have to change the loop counter in byte 29. In the repeated-addition version, you have to make sure you add the right number of ADD 31 lines, and then make sure the STORE 30 and HALT lines are intact. That may not seem unreasonable to you, but what if you want the program to multiply by 30? With the looping version, you just change the loop counter. With the repeated-addition version, you will run out of memory.

As you experiment with the SimCom simulation, you will probably notice a few things:

- Specifying an instruction by selecting an opcode and an argument is much easier than figuring out what the base-10 value should be.
- Even so, SimCom programming isn't very easy.
- When you look at any byte, you can't tell if it is supposed to be an instruction or a value. For example, a byte that contains 100 might mean one hundred, or it might mean SUB 4.

The first two points suggest the need for higher-level programming languages. Hopefully, such languages will support sophisticated operations like multiplication and looping.

The Lessons of SimCom

The point of presenting SimCom in this chapter was to expose you to certain basic functions of programming. Those were high-level languages, loops, referencing, two's complement, and virtual machines. Now that you've been exposed, we can look at how SimCom supports those functions.

Programming with opcodes and arguments is certainly easier than specifying base-10 or (worse yet) base-2 values. But SimCom still forces you to think on the microscopic level. In the Times5 program, you have to remember that byte 29 is the loop counter and byte 30 is the accumulated result. You always have to remember what's going on in the register. High-level languages like Java isolate you from the details of the computer you're programming. (That probably sounds like a good thing, now that you have suffered through SimCom.)

Loops are basic to all programming. Computers are designed to perform repetitive tasks on large data sets, such as printing a paycheck for each employee, displaying each character of a document, or rendering each pixel of a scanned photograph. Loops are difficult to create on SimCom, because everything is hard on SimCom. Java uses simple and powerful looping constructs.

We will cover referencing much later in this book, in Chapter 6, "Arrays." For now, you've had a preview. Remember how SimCom never directly operated with an instruction's argument? The argument was always used as the address of the value to be loaded, added, etc. Now you should be used to the difference between the *address* of a byte and the *value* in that byte. When you program in Java, you don't have to worry about the address of your data, but you still have to think about its location. This will make more sense later on. For now, it's enough to understand the distinction between the value of data and the location of data.

Two's complement is a convention for storing negative numbers. On its surface, SimCom seems to deal only with positive numbers (and zero, of course). But subtraction is supported, and subtraction can lead to negative numbers. If you did the exercise where you modified the LoadAddStore program to make it subtract, you noticed that SimCom thinks 5 minus 6 equals 255. In a way, this is actually correct.

SimCom does not really exist. When you run the animated illustration, there is no actual SimCom computer doing the processing. The program simulates the computer's activity. Thus, SimCom is an imaginary processor that produces real results. As stated earlier in this chapter, an imaginary computer that is simulated on a real one is known as a *virtual computer*. You might have heard of the *JVM*, or *Java Virtual Machine*. Java programs, like SimCom programs, run on a virtual computer.

There is a powerful benefit to this arrangement. When you buy software for your personal computer, you have to check the side of the box to make sure the product works on your platform. If you own a Windows PC, it is useless to buy Macintosh software, just as it is useless to buy SPARC software for a Mac. This is because different manufacturers use different kinds of processors. The binary opcode for addition on one processor type might mean subtract to another type, and might be meaningless to a third type. Thus, software vendors have needed to create a different product for each computer platform they want to support.

Virtual computers do not have this limitation. No matter what kind of computer you're using, SimCom loads when it executes 000, stores when it executes 001, and multiplies by 5 when it executes the Times5 program.

The Java Virtual Machine is much more complicated than SimCom, but the same principle applies. Any Java program will run the same on any hardware. Of course, the JVM itself varies from processor to processor. This is why you had to specify your platform when you downloaded Java. From the JVM's point of view, your platform is known as the *underlying hardware*.

Exercises

NOTE Every chapter in this book ends with exercises that test your understanding of the material and make you think about issues raised in later chapters. The solutions are in Appendix B.

1) A cluster of eight bytes can take on approximately 20 quintillion different values. (One quintillion is a 1 followed by 18 zeroes, or 10 to the 18th power.) Estimate the number of different values that a cluster of 16 bytes can have. Just estimate, do not count. Can you think of anything that comes in such quantities?

2) The SimCom animated illustration is written in Java. When you run the program, how many virtual machines are at work?

3) Write a SimCom program that adds 255 to the value in byte 31 and stores the result in byte 30. Observe the program's behavior. What do you notice?

4) Write a SimCom program that computes the square of the value in byte 31 and stores the result in byte 30. What happens when you try to compute the square of 254?

5) What features could be added to SimCom to make it more useful?