

AP CSA SUMMER WORK

Summer Readings & Assignments:

Java Concepts: Chapter 1 Introduction (**Complete the self-checks**)

Fundamentals of Java: Chapter 1 Background (**Complete Exercises & Written Questions**)

Number Systems: (**Complete Number Systems Conversion WS**)

PLEASE NOTE: You are responsible for completing all summer work. The few days back to school, there will be a quick review on the summer material and multiple assessments to check for understanding.

BACKGROUND

OBJECTIVES

Upon completion of this chapter, you should be able to:

- Give a brief history of computers.
- Describe how hardware and software make up computer architecture.
- Explain the binary representation of data and programs in computers.
- Discuss the evolution of programming languages.
- Describe the software development process.
- Discuss the fundamental concepts of object-oriented programming.

Estimated Time: 2 hours

VOCABULARY

Application software
 Assembly language
 Auxiliary input/output (I/O)
 Auxiliary storage device
 Bit
 Byte
 Central processing unit (CPU)
 Hardware
 Information hiding
 Instance variables
 Internal Memory
 Machine language
 Network connection
 Object-oriented programming
 Primary memory
 RAM
 Secondary memory
 Software
 Software development life cycle (SDLC)
 System software
 Ubiquitous computing
 User interface
 Waterfall model

This is the only chapter in the book that is not about the details of writing Java programs. This chapter discusses computing in general, hardware and software, the representation of information in binary (i.e., as 0s and 1s), and general concepts of object-oriented programming. All this material will give you a broad understanding of computing and a foundation for your study of programming.

1.1 History of Computers

ENIAC, or Electronic Numerical Integrator and Computer, built in the late 1940s, was one of the world's first digital electronic computers. It was a large stand-alone machine that filled a room and used more electricity than all the houses on an average city block. ENIAC contained hundreds of miles of wire and thousands of heat-producing vacuum tubes. The mean time between failures was less than an hour, yet because of its fantastic speed when compared to hand-operated electromechanical calculators, it was immensely useful.

In the early 1950s, IBM sold its first business computer. At the time, it was estimated that the world would never need more than 10 such machines. By comparison, however, its awesome

computational power was a mere 1/2000 of the typical 2-gigahertz laptop computer purchased for about \$1000 in 2010. Today, there are hundreds of millions of laptop and desktop computers in the world. There are also billions of computers embedded in everyday products such as music and video recorders and players, handheld calculators, microwave ovens, cell phones, cars, refrigerators, and even clothing.

The first computers could perform only a single task at a time, and input and output were handled by such primitive means as punch cards and paper tape.

In the 1960s, time-sharing computers, costing hundreds of thousands and even millions of dollars, became popular at organizations large enough to afford them. These computers were powerful enough for 30 people to work on them simultaneously—and each felt as if he or she were the sole user. Each person sat at a teletype connected by wire to the computer. By making a connection through the telephone system, teletypes could even be placed at a great distance from the computer. The teletype was a primitive device by today's standards. It looked like an electric typewriter with a large roll of paper attached. Keystrokes entered at the keyboard were transmitted to the computer, which then echoed them back on the roll of paper. In addition, output from the computer's programs was printed on this roll.

In the 1970s, people began to see the advantage of connecting computers in networks, and the wonders of e-mail and file transfers were born.

In the 1980s, personal computers appeared in great numbers, and soon after, local area networks of interconnected PCs became popular. These networks allowed a local group of computers to communicate and share such resources as disk drives and printers with each other and with large centralized multiuser computers.

The 1990s saw an explosion in computer use. Hundreds of millions of computers appeared on many desktops and in many homes. Most of them are connected through the Internet (Figure 1-1).

FIGURE 1-1

An interconnected world of computers



During the first decade of the twenty-first century, computing has become *ubiquitous* (meaning anywhere and everywhere). Tiny computer chips play the role of brains in cell phones, digital cameras, portable music players, and PDAs (portable digital assistants). Most of these devices now connect to the Internet via wireless technology, giving users unprecedented mobility. The common language of many of these computers is Java.

1.2 Computer Hardware and Software

Computers are machines that process information. They consist of two primary components: hardware and software. **Hardware** consists of the physical devices that you see on your desktop, and **software** consists of the programs that give the hardware useful functionality. The main business of this book, which is programming, concerns software. But before diving into programming, let us take a moment to consider some of the major hardware and software components of a typical computer.

Bits and Bytes

It is difficult to discuss computers without referring to bits and bytes. A *bit*, or *binary digit*, is the smallest unit of information processed by a computer and consists of a single 0 or 1. A *byte* consists of eight adjacent bits. The capacity of computer memory and storage devices is usually expressed in bytes. Some commonly used quantities of bytes are shown in Table 1-1.

TABLE 1-1

Some commonly used quantities of information storage

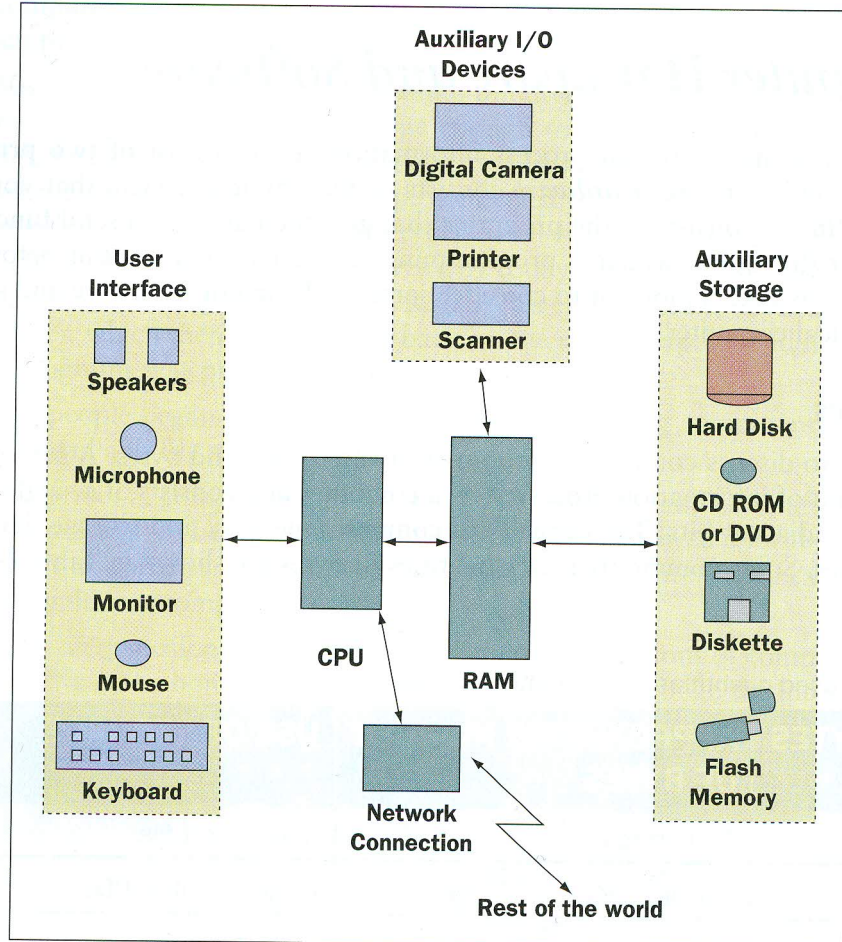
UNIT OF BYTES	APPROXIMATE NUMBER OF BYTES	USE OF STORAGE
Kilobyte	1000 bytes	A small text file
Megabyte	1 million bytes	Large text files, CDs
Gigabyte	1 billion bytes	Video files, RAM, flash memory, hard disk drives, DVDs
Terabyte	1000 gigabytes	File server disks

Computer Hardware

As illustrated in Figure 1-2, a computer consists of six major subsystems.

FIGURE 1-2

A computer's six major subsystems



Listed in order from outside and most visible to inside and most hidden, the six major subsystems are as follows:

- The **user interface**, which supports moment-to-moment communication between a user and the computer
- **Auxiliary input/output (I/O)** devices such as printers and scanners
- **Auxiliary storage devices** for long-term storage of data and programs
- A **network connection** for connecting to the Internet and thus the rest of the world
- **Internal memory**, or **RAM**, for momentary storage of data and programs
- The all important **CPU**, or **central processing unit**

Now we explore each of these subsystems in greater detail.

User Interface

The user interface consists of several devices familiar to everyone who has used a computer. In this book, we assume that our readers have already acquired basic computer literacy and have performed common tasks such as using a word processor or surfing the Internet. The keyboard and mouse are a computer's most frequently used input devices, and the monitor or screen is the principal output device. Also useful, and almost as common, are a microphone for input and speakers for output.

Auxiliary Input/Output (I/O) Devices

In order to communicate with their human users, computers require devices (such as a keyboard) that allow people to input information into the computer. They also require output devices (such as a monitor) that allow people to view information produced by the computer. Input and output devices are often referred to as I/O devices. Computers have not yet produced a paper-free world, so we frequently rely on the output from printers. Scanners are most commonly used to enter images, but in conjunction with appropriate software they can also be used to enter text. Digital cameras can record static images and video for transfer to a computer. Numerous other I/O devices are available for special applications, such as joysticks for games.

Auxiliary Storage Devices

The computer's operating system, the applications we buy, and the documents we write are all stored on devices collectively referred to as auxiliary storage or *secondary memory*. The current capacity of these devices is incredibly large and continues to increase rapidly. In 2009, as these words are being written, the *hard disks* that come encased in a laptop or desktop computer typically store hundreds of billions of bytes of information, or gigabytes (GB) as they are commonly called. External hard disks, with a capacity of one or more terabytes (1000 gigabytes), can be plugged into a personal computer to provide mass storage. In addition to hard disks, there are also several types of *portable storage media*. DVDs, with a capacity of 4.7 to 8 gigabytes, can store a two- to three-hour movie. CDs, with a capacity of 700 megabytes, were originally sized to hold an hour of music, such as the time required for a performance of Beethoven's Ninth Symphony. *Flash memory* sticks with a capacity of several gigabytes are the most convenient portable storage devices. They are used primarily for transporting data between computers that are not interconnected and for making backup copies of crucial computer files.

Network Connection

A network connection is now an essential part of every computer, connecting it to all the resources of the Internet. For home computer users, a modem has long been the most widely used connection device. The term *modem* stands for modulator–demodulator. This term used to refer to a device that converts the digital information (0s and 1s) of the computer to an analog form suitable for transmission on phone lines and vice versa. However, as phone technology becomes increasingly digital, the term *modem* is fast becoming a misnomer. Other devices for connecting to the Internet include so-called cable modems, which use a TV cable or a satellite dish rather than a phone connection; Ethernet cards, which attach directly to local area networks and from there to the Internet; and wireless cards, which transmit digital information through the air and many other objects.

Internal Memory

Although auxiliary storage devices have great capacity, access to their information is relatively slow in comparison to the speed of a computer's central processing unit. For this reason, computers include high-speed internal memory, also called *random access memory* (RAM) or *primary memory*. The contents of RAM are lost every time the computer is turned off, but when

the computer is running, RAM is loaded from auxiliary storage with needed programs and data. Because a byte of RAM costs about 100 times as much as a byte of hard disk storage, personal computers usually contain only about 1 to 4 GB of RAM. Consequently, RAM is often unable to simultaneously hold all the programs and data a person might be using during a computer session. To deal with this situation, the computer swaps programs and data backward and forward between RAM and the hard disk as necessary. Swapping takes time and slows down the apparent speed of the computer from the user's perspective. Often the cheapest way to improve a computer's performance is to install more RAM.

Another smaller piece of internal memory is called *ROM*—short for *read-only memory*. This memory is usually reserved for critical system programs that are used when the computer starts up and that are retained when the computer is shut down.

Finally, most computers have 128 MB or 256 MB of specialized *video RAM* for storing images for graphics and video applications.

Central Processing Unit

The *central processing unit* (*CPU*) does the work of the computer. Given the amazing range of complex tasks performed by computers, one might imagine that the CPU is intrinsically very complex. In fact, the basic functions performed by the CPU consist of the everyday arithmetic operations of addition, subtraction, multiplication, and division, together with some comparison and I/O operations. The complexity lies in the programs that direct the CPU's operations rather than in the individual tasks performed by the CPU itself. It is the programmer's job to determine how to translate a complex task into a series of simple operations, which the computer then executes at blinding speed. One of the authors of this book uses a computer that operates at 2 billion cycles per second (2 GHz), and during each cycle, the CPU executes all or part of a basic operation.

While the basic tasks performed by the CPU are simple, the hardware that makes up the CPU is exceedingly complex. The design of the CPU's hardware is what allows it to perform a series of simple operations at such a high speed. This speed is achieved by packing several million transistors onto a silicon chip roughly the size of a postage stamp. Since 1955, when transistors were first used in computers, hardware engineers have been doubling the speed of computers about every two years, principally by increasing the number of transistors on computer chips. This phenomenon is commonly known as *Moore's Law*. However, basic laws of physics guarantee that the process of miniaturization that allows ever greater numbers of transistors to be packed onto a single chip will soon end. How soon this will be, no one knows.

The *transistor*, the basic building block of the CPU and RAM, is a simple device that can be in one of two states—ON, conducting electricity, or OFF, not conducting electricity. All the information in a computer—programs and data—is expressed in terms of these ONs and OFFs, or 1s and 0s, as they are more conveniently called. From this perspective, RAM is merely a large array of 1s and 0s, and the CPU is merely a device for transforming patterns of 1s and 0s into other patterns of 1s and 0s.

To complete our discussion of the CPU, we describe a typical sequence of events that occurs when a program is executed, or run:

1. The program and data are loaded from auxiliary storage into separate regions of RAM.
2. The CPU copies the program's first instruction from RAM into a decoding unit.
3. The CPU decodes the instruction and sends it to the Arithmetic and Logic Unit (ALU) for execution; for instance, an instruction might tell the ALU to add a number at one location in RAM to one at another location and store the result at a third location.

4. The CPU determines the location of the next instruction and repeats the process of copying, decoding, and executing instructions until the end of the program is reached.
5. After the program has finished executing, the data portion of RAM contains the results of the computation performed by the program.

Needless to say, this description has been greatly simplified. We have, for instance, ignored the use of separate processors for graphics, multicore processors that divide the work among several CPUs, and all issues related to input and output; however, the description provides a view of the computational process that will help you understand what follows.

Computer Software

Computer hardware processes complex patterns of electronic states, or 0s and 1s. Computer software transforms these patterns, allowing them to be viewed as text, images, and so forth. Software is generally divided into two broad categories—*system software* and *application software*.

System Software

System software supports the basic operations of a computer and allows human users to transfer information to and from the computer. This software includes

- The operating system, especially the file system for transferring information to and from disk and schedulers for running multiple programs concurrently
- Communications software for connecting to other computers and the Internet
- Compilers for translating user programs into executable form
- The user interface subsystem, which manages the look and feel of the computer, including the operation of the keyboard, the mouse, and a screen full of overlapping windows

Application Software

Application software allows human users to accomplish specialized tasks. Examples of types of application software include

- Word processors
- Spreadsheets
- Database systems
- Multimedia software for digital music, photography, and video
- Other programs we write

EXERCISE 1.2

1. What is the difference between a bit and a byte?
2. Name two input devices and two output devices.
3. What is the purpose of auxiliary storage devices?
4. What is RAM and how is it used?
5. Discuss the differences between hardware and software.

1.3 Binary Representation of Information and Computer Memory

As we saw in the previous section, computer memory stores patterns of electronic signals, which the CPU manipulates and transforms into other patterns. These patterns in turn can be viewed as strings of binary digits or bits. Programs and data are both stored in memory, and there is no discernible difference between program instructions and data; they are both just sequences of 0s and 1s. To determine what a sequence of bits represents, we must know the context. We now examine how different types of information are represented in binary notation.

Integers

We normally represent numbers in decimal (base 10) notation, whereas the computer uses binary (base 2) notation. Our addition to base 10 is a physiological accident (10 fingers rather than 8, 12, or some other number). The computer's dependence on base 2 is due to the on/off nature of electric current.

To understand base 2, we begin by taking a closer look at the more familiar base 10. What do we really mean when we write a number such as 5403? We are saying that the number consists of 5 thousands, 4 hundreds, 0 tens, and 3 ones. Expressed differently, 5403 looks like this:

$$(5 * 10^3) + (4 * 10^2) + (0 * 10^1) + (3 * 10^0)$$

In this expression, each term consists of a power of 10 times a coefficient between 0 and 9. In a similar manner, we can write expressions involving powers of 2 and coefficients between 0 and 1. For instance, let us analyze the meaning of 10011_2 , where the subscript 2 indicates that we are using a base of 2:

$$\begin{aligned} 10011_2 &= (1 * 2^4) + (0 * 2^3) + (0 * 2^2) + (1 * 2^1) + (1 * 2^0) \\ &= 16 + 0 + 0 + 2 + 1 = 19 \\ &= (1 * 10^1) + (9 * 10^0) \end{aligned}$$

The inclusion of the base as a subscript at the end of a number helps us avoid possible confusion. Here are four numbers that contain the same digits but have different bases and thus different values:

$$\begin{aligned} 1101101_{16} \\ 1101101_{10} \\ 1101101_8 \\ 1101101_2 \end{aligned}$$

Computer scientists use bases 2 (*binary*), 8 (*octal*), and 16 (*hexadecimal*) extensively. Base 16 presents the dilemma of how to represent digits beyond 9. The accepted convention is to use the letters A through F, corresponding to 10 through 15. For example:

$$\begin{aligned} 3BC4_{16} &= (3 * 16^3) + (11 * 16^2) + (12 * 16^1) + (4 * 16^0) \\ &= (3 * 4096) + (11 * 256) + (12 * 16) + 4 \\ &= 15300_{10} \end{aligned}$$

As you can see from these examples, the next time you are negotiating your salary with an employer, you might allow the employer to choose the digits as long as she allows you to pick the base. Table 1-2 shows some base 10 numbers and their equivalents in base 2. An important fact of the base 2 system is that 2^N distinct values can be represented using N bits. For example, four bits represent 2^4 or 16 values 0000, 0001, 0010, ..., 1110, 1111. A more extended discussion of number systems appears in Appendix E of this book.

TABLE 1-2

Some base 10 numbers and their base 2 equivalents

BASE 10	BASE 2
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
43	101011

Floating-Point Numbers

Numbers with a fractional part, such as 354.98, are called *floating-point numbers*. They are a bit trickier to represent in binary than integers. One way is to use the *mantissa/exponent notation*, in which the number is rewritten as a value between 0 and 1, inclusive ($0 \leq x < 1$), times a power of 10. For example:

$$354.98_{10} = 0.35498_{10} * 10^3$$

where the mantissa is 35498, and the exponent is 3, or the number of places the decimal has moved. Similarly, in base 2

$$10001.001_2 = 0.10001001_2 * 2^5$$

with a mantissa of 10001001 and exponent of $5_{10} = 101_2$. In this way we can represent any floating-point number by two separate sequences of bits, with one sequence for the mantissa and the other for the exponent.

Many computers follow the slightly different IEEE standard, in which the mantissa contains one digit before the decimal or binary point. In binary, the mantissa's leading 1 is then suppressed. Originally, this was a 7-bit code, but it has been extended in various ways to 8 bits.

Characters and Strings

To process text, computers must represent characters such as letters, digits, and other symbols on a keyboard. There are many encoding schemes for characters. One popular scheme is called *ASCII* (*American Standard Code for Information Interchange*). In this scheme, each character is represented as a pattern of 8 bits or a byte.

In binary notation, byte values can range from 0000 0000 to 1111 1111, allowing for 256 possibilities. These are more than enough for the following:

- A...Z
- a...z
- 0...9
- +, -, *, /, etc.
- Various unprintable characters such as carriage return, line feed, a ringing bell, and command characters

Table 1-3 shows some characters and their corresponding ASCII bit patterns.

TABLE 1-3

Some characters and their corresponding ASCII bit patterns

CHARACTER	BIT PATTERN	CHARACTER	BIT PATTERN	CHARACTER	BIT PATTERN
A	0100 0001	a	0110 0001	0	0011 0000
B	0100 0010	b	0110 0010	1	0011 0001
...
Z	0101 1010	z	0111 1010	9	0011 1001

Java, however, uses a scheme called *Unicode* rather than ASCII. In this scheme, each character is represented by a pattern of 16 bits, ranging from 0000 0000 0000 0000 to 1111 1111 1111 1111. Unicode allows for 65,536 possibilities and can represent many alphabets simultaneously. Within Unicode, the patterns 0000 0000 0000 0000 to 0000 0000 1111 1111 duplicate the ASCII encoding scheme.

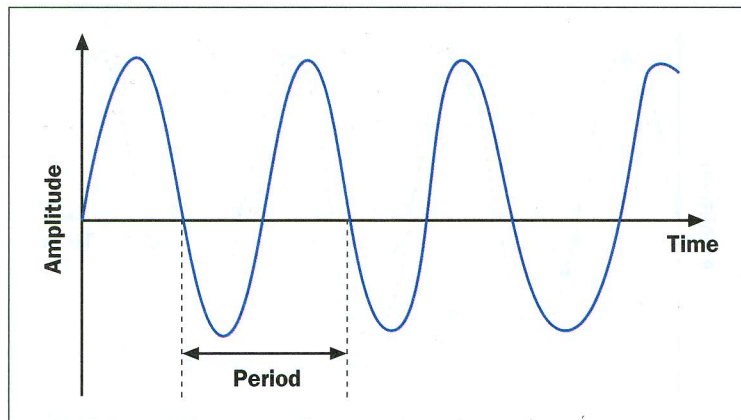
Strings are another type of data used in text processing. Strings are sequences of characters, such as "The cat sat on the mat." The computer encodes each character in ASCII or Unicode and strings them together.

Sound

The information contained in sound is *analog*. Unlike integers and text, which on a computer have a finite range of discrete values, analog information has a continuous range of infinitely many values. The analog information in sound can be plotted as a periodic waveform such as the one shown in Figure 1-3. The amplitude or height of a waveform measures the volume of the sound. The time that a waveform takes to make one complete cycle is called its period. The frequency or number of cycles per second of a sound's waveform measures its pitch. Thus, the higher a wave is, the louder the sound's volume, and the closer together the cycles are, the higher the sound's pitch.

FIGURE 1-3

A sound waveform



An input device for sound must translate this continuous analog information into discrete, digital values. A technique known as sampling takes a reading of the amplitude values on a waveform at regular intervals, as shown in Figure 1-4a. If the intervals are short enough, the digital information can be used to reconstruct a waveform that approximates a sound that most human beings cannot distinguish from the original. Figure 1-4b shows the waveform generated for output from the waveform sampled in Figure 1-4a. The original waveform is shown as a dotted line, whereas the regenerated waveform is shown as a solid line. As you can see, if the sampling rate is too low, some of the measured amplitudes (the heights and depths of the peaks and valleys in the waves) will be inaccurate. The sampling rate must also be high enough to capture the range of frequencies (the waves and valleys themselves), from the lowest to the highest, that most humans can hear. Psychologists and audiophiles agree that this range is from 20 to 22,000 Hertz (cycles per second). Because a sample must capture both the peak and the valley of a cycle, the sampling rate must be double the frequency. Therefore, a standard rate of 44,000 samples per second has been established for sound input. Amplitude is usually measured on a scale from 0 to 65,535.

FIGURE 1-4a

Sampling a waveform

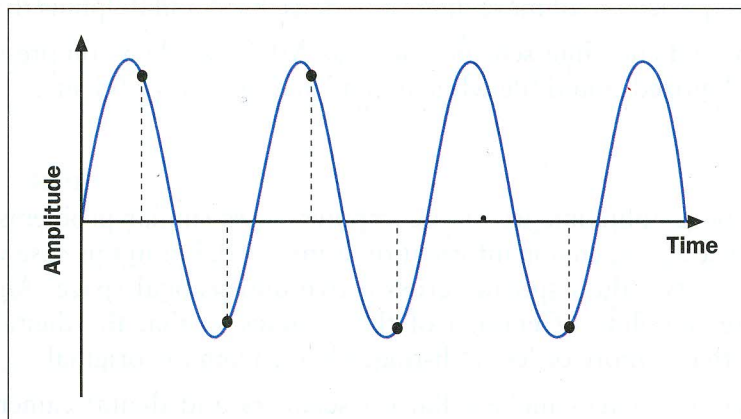
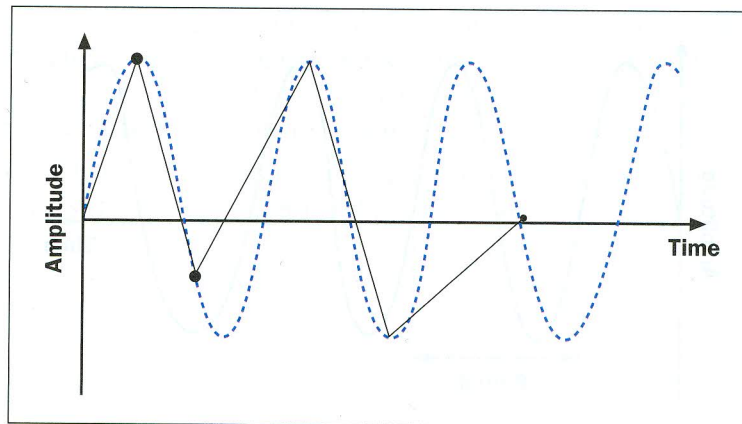


FIGURE 1-4b

Regenerating the sound from the samples



Because of the high sampling rate, the memory requirements for storing sound are much greater than those of text. For example, to digitize an hour of stereo music, the computer must perform the following steps:

- For each stereo channel, every $1/44,000$ of a second, measure the amplitude of the sound on a scale of 0 to 65,535.
- Convert this number to binary using 16 bits.

Thus, 1 hour of stereo music requires

$$\begin{aligned}
 & 2 \text{ channels} * \frac{1 \text{ hour}}{\text{channel}} * \frac{60 \text{ minutes}}{\text{hour}} * \frac{60 \text{ seconds}}{\text{minute}} * \frac{44,000 \text{ samples}}{\text{second}} * \frac{16 \text{ bits}}{\text{sample}} \\
 &= 5,068,800,000 \text{ bits} \\
 &= 633,600,000 \text{ bytes}
 \end{aligned}$$

which is the capacity of a standard CD.

The sampling rate of 44,000 times a second is not arbitrary, but corresponds to the number of samples required to reproduce accurate sounds with a frequency of up to 22,000 cycles per second. Sounds above that frequency are of more interest to dogs, bats, and dolphins than to people.

Many popular sound-encoding schemes, such as MP3, use data-compression techniques to reduce the size of a digitized sound file while minimizing the loss of fidelity.

Images

Representing photographic images on a computer poses similar problems to those encountered with sound. Once again, analog information is involved, but in this case we have an infinite set of color and intensity values spread across a two-dimensional space. And once again, the solution involves the sampling of enough of these values so that the digital information can reproduce an image that is more or less indistinguishable from the original.

Sampling devices for images include flatbed scanners and digital cameras. These devices measure discrete values at distinct points or *pixels* in a two-dimensional grid. In theory, the more pixels that are taken, the more continuous and realistic the resulting image will appear. In practice, however, the human eye cannot discern objects that are closer together than 0.1 mm,

so a sampling rate of 10 pixels per linear millimeter (250 pixels per inch and 62,500 pixels per square inch) would be plenty accurate. Thus, a 3-by-5-inch image would need

$$3 * 5 * 62,500 \text{ pixels/inch}^2 = 937,500 \text{ pixels}$$

For most purposes, however, we can settle for a much lower sampling rate and thus fewer pixels per square inch.

The values sampled are color values, and there are an infinite number of these on the spectrum. If we want a straight black-and-white image, we need only two possible values, or one bit of information, per pixel. For grayscale images, 3 bits allow for 8 shades of gray, while 8 bits allow for 256 shades of gray. A true-color scheme called RGB is based on the fact that the human retina is sensitive to red, green, and blue components. This scheme uses 8 bits for each of the three color components, for a total of 24 bits or 16,777,216 (the number of possible sequences of 24 bits) color values per pixel. No matter which color scheme is used, the sampling device selects a discrete value that is closest to the color and intensity of the image at a given point in space.

The file size of a true-color digitized image can be quite large. For example, the 3-by-5-inch image discussed earlier would need 937,500 pixels * 24 bits/pixel or about 2.5 MB of storage. As with sound files, image files can be saved in a compressed format, such as GIF or JPEG, without much loss of realism.

Video

Video consists of a soundtrack and a set of images called *frames*. The sound for a soundtrack is recorded, digitized, and processed in the manner discussed earlier. The frames are snapshots or images recorded in sequence during a given time interval. If the time intervals between frames are short enough, the human eye will perceive motion in the images when they are replayed. The rate of display required for realistic motion is between 16 and 24 frames per second.

The primary challenge in digitizing video is achieving a suitable data compression scheme. Let's assume that you want to display each frame on a 15-inch laptop monitor. Each frame will then cover about 120 square inches, so even with a conservative memory allocation of 10 kilobytes (KB) of storage per square inch of image, we're looking at 1.2 MB/frame * 16 frames/second = 19 MB/second of storage. A two-hour feature film would need 432,000 seconds * 19 MB/second = 8,208,000 MB without the soundtrack! A typical DVD has space for several gigabytes of data, so our uncompressed video would obviously not fit on a DVD. For this reason, very sophisticated data-compression schemes, such as MPEG, have been developed that allow three-hour films to be placed on a DVD and shorter, smaller-framed video clips to be downloaded and played from the Internet.

Program Instructions

Program instructions are represented as a sequence of bits in RAM. For instance, on some hypothetical computer, the instruction to add two numbers already located in RAM and store their sum at some third location in RAM might be represented as follows:

```
0000 1001 / 0100 0000 / 0100 0010 / 0100 0100
```

where

- The first group of 8 bits represents the ADD command and is called the *operation code*, or *opcode* for short.

- In other words, our instruction translates as follows: add the number at location 64 to the number at location 66 and store the sum at location 68.

We can envision a computer's memory as a gigantic sequence of bytes. A byte's location in memory is called its *address*. Addresses are numbered from 0 to 1 less than the number of bytes of memory installed on that computer, say, $32M - 1$, where M stands for *megabyte*.

FIGURE 1-5

Address	Memory
---------	--------

[illegible]

The several possible meanings include these:

- If it is a string, then the meaning is “Hi”.
- If it is a binary encoded integer, then the meaning is 18537_{10} .
- If it is a program instruction, then it might mean ADD, depending on the type of computer.

In addition to these data values, the computer also stores contextual information that allows it to interpret them.

EXERCISE 1.3

1. Translate 11100011_2 to a base 10 number.
2. Translate $45B_{16}$ to a base 10 number.
3. What is the difference between Unicode and ASCII?
4. Assume that 4 bits are used to represent the intensities of red, green, and blue. How many total colors are possible in this scheme?
5. An old-fashioned computer has just 16 bits available to represent an address of a memory location. How many total memory locations can be addressed in this machine?



Computer Ethics

THE ACM CODE OF ETHICS

The Association for Computing Machinery (ACM) is the flagship organization for computing professionals. The ACM supports publications of research results and new trends in computer science, sponsors conferences and professional meetings, and provides standards for computer scientists as professionals. The standards concerning the conduct and professional responsibility of computer scientists have been published in the ACM Code of Ethics. The code is intended as a basis for ethical decision making and for judging the merits of complaints about violations of professional ethical standards.

The code lists several general moral imperatives for computer professionals:

- Contribute to society and human well-being.
- Avoid harm to others.
- Be honest and trustworthy.
- Be fair and take action not to discriminate.
- Honor property rights, including copyrights and patents.
- Give proper credit for intellectual property.
- Respect the privacy of others.
- Honor confidentiality.

The code also lists several more specific professional responsibilities:

- Strive to achieve the highest quality, effectiveness, and dignity in both the process and products of professional work.
- Acquire and maintain professional competence.
- Know and respect existing laws pertaining to professional work.
- Accept and provide appropriate professional review.
- Give comprehensive and thorough evaluations of computer systems and their impacts, including analysis of possible risks.
- Honor contracts, agreements, and assigned responsibilities.
- Improve public understanding of computing and its consequences.
- Access computing and communication resources only when authorized to do so.

In addition to these principles, the code offers a set of guidelines that provide professionals with explanations of various issues contained in the principles. The complete text of the ACM Code of Ethics is available at the ACM's Web site, <http://www.acm.org>.

1.4 Programming Languages

Question: “If a program is just some very long pattern of electronic states in a computer’s memory, then what is the best way to write a program?” The history of computing provides several answers to this question in the form of generations of programming languages.

Generation 1 (Late 1940s to Early 1950s)—Machine Languages

Early on, when computers were new, they were very expensive, and programs were very short. To be executed by computer hardware, these programs had to be coded in *machine language*, whose only symbols are the binary digits 1 and 0. Programmers toggled switches on the front of the computer to enter programs and data directly into RAM in the form of 0s and 1s. Later, devices were developed to read the 0s and 1s into memory from punched cards and paper tape. There were several problems with this machine language-coding technique:

- Coding was error-prone; entering just a single 0 or 1 incorrectly was enough to make a program run improperly or not at all.
- Coding was tedious and slow.
- It was extremely difficult to modify programs.
- It was nearly impossible for one person to decipher another’s program.
- A program was not portable to a different type of computer because each type had its own unique machine language.

Needless to say, this technique is no longer used!

Generation 2 (Early 1950s to Present)—Assembly Languages

Instead of the binary notation of machine language, *assembly language* uses mnemonic symbols to represent instructions and data. For instance, here is a machine language instruction followed by its assembly language equivalent:

```
0011 1001 / 1111 0110 / 1111 1000 / 1111 1010
ADD          A,          B,          C
```

This code translates as follows:

1. Add the number at memory location 246 (which we refer to as A)
2. to the number at memory location 248 (which we refer to as B)
3. and store the result at memory location 250 (which we refer to as C).

Each *assembly language* instruction corresponds exactly to one machine language instruction. The standard procedure for using assembly language consists of several steps:

1. Write the program in assembly language.
2. Translate the program into a machine language program—this is done by a computer program called an *assembler*.
3. Load and run the machine language program—this is done by another program called a *loader*.

When compared to machine language, assembly language is

- More programmer friendly
- Still unacceptably (by today's standards) tedious to use, difficult to modify, and so forth
- No more portable than machine language because each type of computer still has its own unique assembly language

Assembly language is used as little as possible by programmers today, although sometimes it is used when memory or processing speed are at a premium. Thus, every student of computer science probably learns at least one assembly language.

Generation 3 (Mid-1950s to Present)—High-Level Languages

Early examples of *high-level languages* are FORTRAN and COBOL, which are still in widespread use. Later examples are BASIC, C, and Pascal. Recent examples include Smalltalk, C++, Python, and Java. All these languages are designed to be human friendly—easy to write, easy to read, and easy to understand—at least when compared to assembly language. For example, all high-level languages support the use of algebraic notation, such as the expression $x + (y * z)$.

Each instruction in a high-level language corresponds to many instructions in machine language. Translation to machine language is done by a program called a *compiler*. Generally, a program written in a high-level language is portable, but it must be recompiled for each different type of computer on which it is going to run. Java is a notable exception because it is a high-level language that does not need to be recompiled for each type of computer. We learn more about this in Chapter 2. The vast majority of software written today is written in high-level languages.

EXERCISE 1.4

1. State two of the difficulties of programming with machine language.
2. State two features of assembly language.
3. What is a loader, and what is it used for?
4. State one difference between a high-level language and assembly language.

1.5 The Software Development Process

High-level programming languages help programmers write high-quality software in much the same sense as good tools help carpenters build high-quality houses, but there is much more to programming than writing lines of code, just as there is more to building houses than pounding nails. The “more” consists of organization and planning and various diagrammatic conventions for expressing those plans. To this end, computer scientists have developed a view of the software development process known as the *software development life cycle (SDLC)*. We now present a useful version of this life cycle called the *waterfall model*.

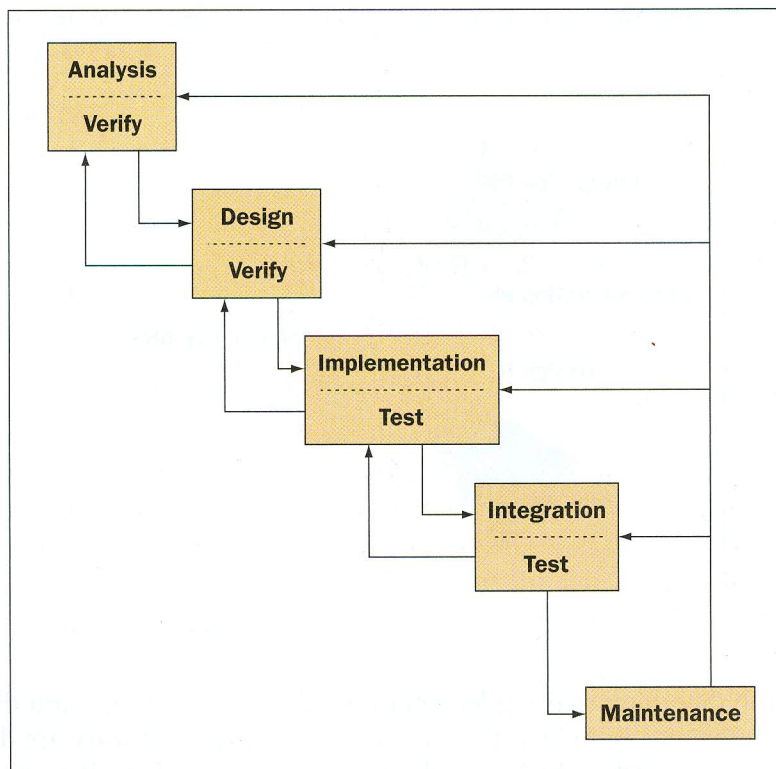
The waterfall model consists of several phases:

1. *Customer request*—In this phase, the programmers receive a broad statement of a problem that is potentially amenable to a computerized solution. This step is also called the *user requirements phase*.
2. *Analysis*—The programmers determine what the program will do. This is sometimes viewed as a process of clarifying the specifications for the problem.
3. *Design*—The programmers determine how the program will do its task.
4. *Implementation*—The programmers write the program. This step is also called the *coding phase*.
5. *Integration*—Large programs have many parts. In the integration phase, these parts are brought together into a smoothly functioning whole, usually not an easy task.
6. *Maintenance*—Programs usually have a long life; a life span of 5 to 15 years is common for software. During this time, known as the maintenance phase, requirements change and minor or major modifications must be made.

The interaction between the phases is shown in Figure 1-6. Note that the figure resembles a waterfall, in which the results of each phase flow down to the next. Mistakes can be detected in each phase during testing or verification. A mistake detected in the verification or testing in one phase often requires the developer to back up and redo some of the work in the previous phase. Modifications made during maintenance also require backing up to earlier phases.

FIGURE 1-6

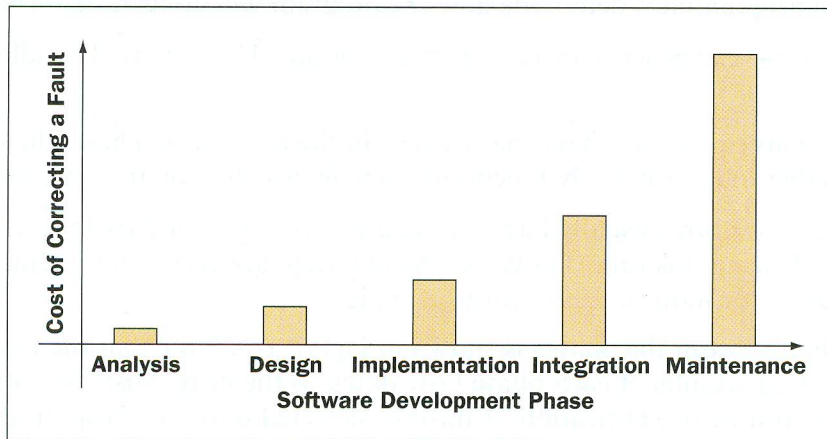
The waterfall model of the software development life cycle



Programs rarely work as hoped the first time they are run; hence, they should be subjected to extensive and careful testing. Many people think that testing is an activity that applies only to the implementation and integration phases; however, the outputs of each phase should be scrutinized carefully. In fact, mistakes found early are much less expensive to correct than those found late. Figure 1-7 illustrates some relative costs of repairing mistakes when found in different phases.

FIGURE 1-7

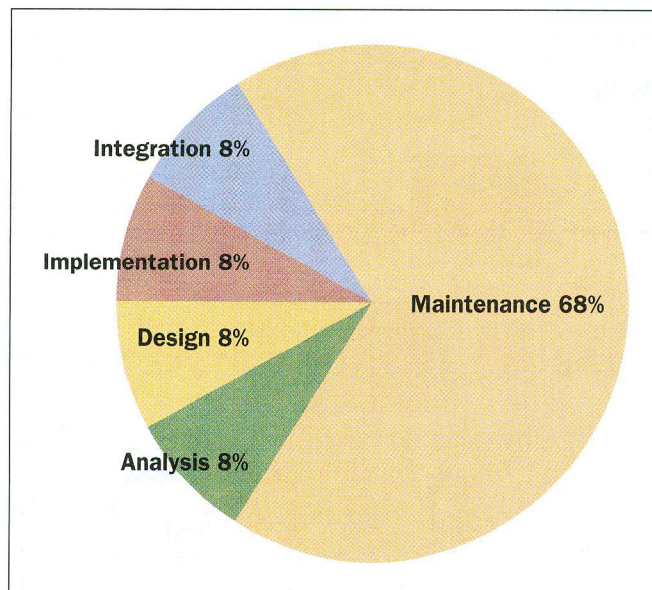
Relative costs of repairing mistakes when found in different phases



Finally, the cost of developing software is not spread equally over the phases. The percentages shown in Figure 1-8 are typical.

FIGURE 1-8

Percentage of total cost incurred in each phase of the development process



Most people probably think that implementation takes the most time and therefore costs the most. However, maintenance is, in fact, the most expensive aspect of software development. The cost of maintenance can be reduced by careful analysis, design, and implementation.

As you read this book and begin to sharpen your programming skills, you should remember two points:

1. There is more to software development than hacking out code.
2. If you want to reduce the overall cost of software development, write programs that are easy to maintain. This requires thorough analysis, careful design, and good coding style. We have more to say about coding style throughout the book.

EXERCISE 1.5

1. What happens during the Analysis and Design phases of the waterfall model of software development?
2. Which phase of the waterfall model of software development incurs the highest cost to developers?
3. Why would a programmer back up to an earlier phase in the waterfall model of software development?
4. In which phase of the waterfall model of software development is the detection and correction of errors the least expensive?

For a thorough discussion of the software development process and software engineering in general, see Shari Pfleeger and Joanne Atlee, *Software Engineering*, 4th Edition (Prentice-Hall, 2009).

1.6 Basic Concepts of Object-Oriented Programming

The high-level programming languages mentioned earlier fall into two major groups, and these two groups utilize two different approaches to programming. The first group, consisting of the older languages (COBOL, FORTRAN, BASIC, C, and Pascal), uses what is called a *procedural approach*. Inadequacies in the procedural approach led to the development of the *object-oriented approach* and to several newer languages (Smalltalk, C++, Python, and Java). There is little point in trying to explain the differences between these approaches in an introductory programming text, but suffice it to say that the object-oriented approach is widely considered the superior of the two. There are also several other approaches to programming, but that, too, is a topic for a more advanced text.

Most programs in the real world contain hundreds of thousands of lines of code. Writing such programs is a highly complex task that can only be accomplished by breaking the code into communicating components. This is an application of the well-known principle of “divide and conquer” that has been applied successfully to many human endeavors. There are various strategies for subdividing a program, and these depend on the type of programming language used. We now give an overview of the process in the context of *object-oriented programming* (OOP)—that is, programming with objects. Along the way, we introduce fundamental OOP concepts, such as class, inheritance, and polymorphism. Each of these concepts is also discussed in greater detail later in the book. For best results, reread this section as you encounter each concept for a second time.

We proceed by way of an extended analogy. Suppose that it is your task to plan an expedition in search of the lost treasure of Balbor. Your overall approach might consist of the following steps:

1. *Planning*—You determine the different types of team members needed, including leaders, pathfinders, porters, and trail engineers. You then determine how many you need of each, and then define the responsibilities of each member in terms of
 - A list of the resources required, including the materials and knowledge needed by each member
 - The rules of behavior, which define how each team member behaves in and responds to various situations
2. *Execution*—You recruit the team members and assemble them at the starting point, send the team on its way, and sit back and wait for the outcome. (There is no sense in endangering your own life, too.)
3. *Outcome*—If the planning was done well, you will be rich; otherwise, prepare for disappointment.

How does planning an expedition relate to OOP? We give the answer in Table 1-4. The left side of the table describes various aspects of the expedition, and the right side lists corresponding aspects of OOP. Do not expect to understand all the new terms now. We explore them with many other examples in the rest of this book.

TABLE 1-4

Comparing an expedition to OOP

THE WORLD OF THE EXPEDITION	THE WORLD OF OOP
The trip must be planned.	Computer software is created in a process called programming .
The team is composed of different types of team members, and each type is characterized by its list of resources and rules of behavior.	A program is composed of different types of software components called classes . A class defines or describes a list of data resources called instance variables and rules of behavior called methods . Combining the description of resources and behaviors into a single software entity is called encapsulation .
First the trip must be planned. Then it must be set in motion.	First a program must be written. Then it must be run, or executed.
When the expedition is in progress, the team is composed of individual members and not types. Each member is, of course, an instance of a particular type.	An executing program is composed of interacting objects, and each object's resources (instance variables) and rules of behavior (methods) are described in a particular class. An object is said to be an instance of the class that describes its resources and behavior.
At the beginning of the expedition, team members must be recruited.	While a program is executing, it creates, or instantiates, objects as needed.

TABLE 1-4 (continued)

Comparing an expedition to OOP

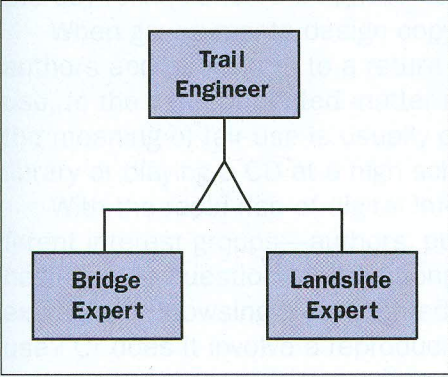
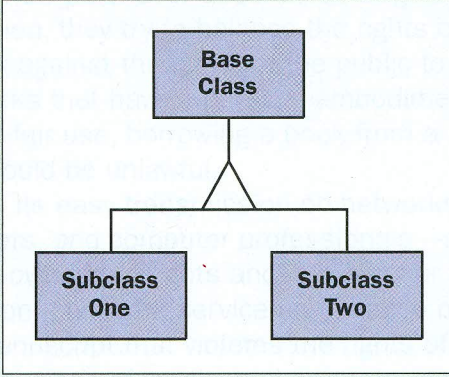
THE WORLD OF THE EXPEDITION	THE WORLD OF OOP
Team members working together accomplish the mission of the expedition. They do this by asking each other for services.	Objects working together accomplish the mission of the program. They do this by asking each other for services or, in the language of OOP, by sending messages to each other.
When a team member receives a request for service, she follows the instructions in a corresponding rule of behavior.	When an object receives a message, it refers to its class to find a corresponding rule or method to execute.
If someone who is not a pathfinder wants to know where north is, she does not need to know anything about compasses. She merely asks one of the pathfinders, who are well-known providers of this service. Even if she did ask a pathfinder for his compass, he would refuse. Thus, team members tell others about the services they provide but never anything about the resources they use to provide these services.	If an object A needs a service that it cannot provide for itself, then A requests the service from some well-known provider B. However, A knows nothing of B's data resources and never asks for access to them. This principle of providing access to services but not to data resources is called information hiding .
The expedition includes general-purpose trail engineers plus two specialized subtypes. All trail engineers share common skills, but some specialize in bridge building and others in clearing landslides. Thus, there is a hierarchy of engineers.	Classes are organized into a hierarchy also. The class at the root, or base, of the hierarchy defines methods and instance variables that are shared by its subclasses, those below it in the hierarchy. Each subclass then defines additional methods and instance variables. This process of sharing is called inheritance .
 <pre> graph TD TE[Trail Engineer] --> BE[Bridge Expert] TE --> LE[Landslide Expert] </pre>	 <pre> graph TD BC[Base Class] --> SO[Subclass One] BC --> ST[Subclass Two] </pre>
At the end of the day, the leader tells each member to set up camp. All members understand this request, but their responses depend on their types. Each type responds in a manner consistent with its specific responsibilities.	Different types of objects can understand the same message. This is referred to as polymorphism . However, an object's response depends on the class to which it belongs.

TABLE 1-4 (continued)

Comparing an expedition to OOP

THE WORLD OF THE EXPEDITION	THE WORLD OF OOP
During the trip, everyone is careful not to ask an individual to do something for which he is not trained—that is, for which he does not have a rule of behavior.	When writing a program, we never send a message to an object unless its class has a corresponding method.
One can rely on team members to improvise and resolve ambiguities and contradictions in rules.	In contrast, a computer does exactly what the program specifies—neither more nor less. Thus, programming errors and oversights, no matter how small, are usually disastrous. Therefore, programmers need to be excruciatingly thorough and exact when writing programs.

EXERCISE 1.6

1. In what way is programming like planning the construction of a house?
2. An object-oriented program is a set of objects that interact by sending messages to each other. Explain.
3. What is a class, and how does it relate to objects in an object-oriented program?
4. Explain the concept of inheritance with an example.
5. Explain the concept of information hiding with an example.

REVIEW Questions

WRITTEN QUESTIONS

Write a brief answer to each of the following questions.

1. What are the three major hardware components of a computer?
2. Name three input devices.
3. Name two output devices.
4. What is the difference between application software and system software?
5. Name a first-generation programming language, a second-generation programming language, and a third-generation programming language.

FILL IN THE BLANK

Complete the following sentences by writing the correct word or words in the blanks provided.

1. All information used by a computer is represented using _____ notation.
2. The _____ phase of the software life cycle is also called the *coding phase*.
3. More than half of the cost of developing software goes to the _____ phase of the software life cycle.

4. ACM stands for _____.
5. Copyright law is designed to give fair use to the public and to protect the rights of _____ and _____.

PROJECTS

PROJECT 1-1

Take some time to become familiar with the architecture of the computer you will use for this course. Describe your hardware and software using the following guidelines:

- What hardware components make up your system?
- How much memory does your system have?
- What are the specifications of your CPU? (Do you know its speed and what kind of microprocessor it has?)
- What operating system are you using? What version of that operating system is your computer currently running?
- What major software applications are loaded on your system?

CRITICAL *Thinking*

You have just written some software that you would like to sell. Your friend suggests that you copyright your software. Discuss why this might be a good idea.

CHAPTER CONTENTS

1.1 What Is Programming? 2

1.2 The Anatomy of a Computer 3

RANDOM FACT 1.1: The ENIAC and the Dawn of Computing 7

1.3 Translating Human-Readable Programs to Machine Code 8

1.4 The Java Programming Language 10

1.5 Becoming Familiar with Your Computer 12

PRODUCTIVITY HINT 1.1: Understand the File System 15

PRODUCTIVITY HINT 1.2: Have a Backup Strategy 16

1.6 Compiling a Simple Program 17

SYNTAX 1.1: Method Call 21

COMMON ERROR 1.1: Omitting Semicolons 22

ADVANCED TOPIC 1.1: Alternative Comment Syntax 22

1.7 Errors 23

COMMON ERROR 1.2: Misspelling Words 24

1.8 The Compilation Process 25

1.1 What Is Programming?

You have probably used a computer for work or fun. Many people use computers for everyday tasks such as balancing a checkbook or writing a term paper. Computers are good for such tasks. They can handle repetitive chores, such as totaling up numbers or placing words on a page, without getting bored or exhausted. Computers also make good game machines because they can play sequences of sounds and pictures, involving the human user in the process.

The flexibility of a computer is quite an amazing phenomenon. The same machine can balance your checkbook, print your term paper, and play a game. In contrast, other machines carry out a much narrower range of tasks—a car drives and a toaster toasts.

A computer must be programmed to perform tasks. Different tasks require different programs.

A computer program executes a sequence of very basic operations in rapid succession.

To achieve this flexibility, the computer must be *programmed* to perform each task. A computer itself is a machine that stores data (numbers, words, pictures), interacts with devices (the monitor screen, the sound system, the printer), and executes programs. Programs are sequences of instructions and decisions that the computer carries out to achieve a task. One program balances checkbooks; a different program, perhaps designed and constructed by a different company, processes words; and a third program, probably from yet another company, plays a game.

Today's computer programs are so sophisticated that it is hard to believe that they are all composed of extremely primitive operations.

A typical operation may be one of the following:

- Put a red dot onto this screen position.
- Send the letter A to the printer.
- Get a number from this location in memory.
- Add up two numbers.
- If this value is negative, continue the program at that instruction.

A computer program contains the instruction sequences for all tasks that it can execute.

A computer program tells a computer, in minute detail, the sequence of steps that are needed to complete a task. A program contains a huge number of simple operations, and the computer executes them at great speed. The computer has no intelligence—it simply executes instruction sequences that have been prepared in advance.

To use a computer, no knowledge of programming is required. When you write a term paper with a word processor, that software package has been programmed by the manufacturer and is ready for you to use. That is only to be expected—you can drive a car without being a mechanic and toast bread without being an electrician.

A primary purpose of this book is to teach you how to design and implement computer programs. You will learn how to formulate instructions for all tasks that your programs need to execute.

Keep in mind that programming a sophisticated computer game or word processor requires a team of many highly skilled programmers, graphic artists, and other professionals. Your first programming efforts will be more mundane. The concepts and skills you learn in this book form an important foundation, but you should not expect to immediately produce professional software. A typical college program in computer science or software engineering takes four years to complete; this book is intended as an introductory course in such a program.

Many students find that there is an immense thrill even in simple programming tasks. It is an amazing experience to see the computer carry out a task precisely and quickly that would take you hours of drudgery.

SELF CHECK

1. What is required to play a music CD on a computer?
2. Why is a CD player less flexible than a computer?
3. Can a computer program develop the initiative to execute tasks in a better way than its programmers envisioned?

1.2 The Anatomy of a Computer

To understand the programming process, you need to have a rudimentary understanding of the building blocks that make up a computer. This section will describe a personal computer. Larger computers have faster, larger, or more powerful components, but they have fundamentally the same design.

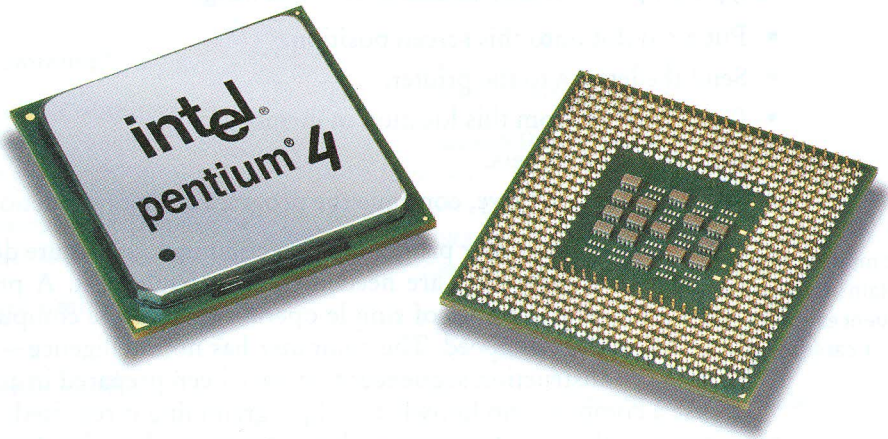


Figure 1 Central Processing Unit

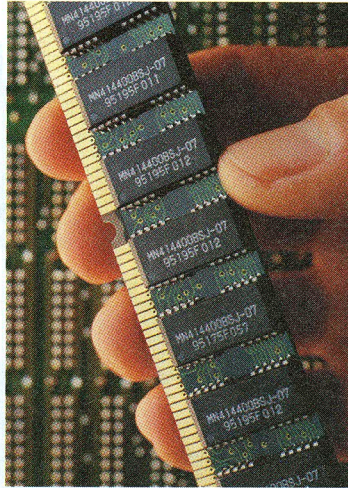
At the heart of the computer lies the central processing unit (CPU).

At the heart of the computer lies the *central processing unit* (CPU) (see Figure 1). It consists of a single *chip* (integrated circuit) or a small number of chips. A computer chip is a component with a plastic or metal housing, metal connectors, and inside wiring made principally from silicon. For a CPU chip, the inside wiring is enormously complicated. For example, the Pentium 4 chip (a popular CPU for personal computers at the time of this writing) contains over 50 million structural elements called *transistors*—the elements that enable electrical signals to control other electrical signals, making automatic computing possible. The CPU locates and executes the program instructions; it carries out arithmetic operations such as addition, subtraction, multiplication, and division; and it fetches data from storage and input/output devices and sends data back.

Data and programs are stored in primary storage (memory) and secondary storage (such as a hard disk).

The computer keeps data and programs in *storage*. There are two kinds of storage. *Primary storage*, also called *random-access memory* (RAM) or simply *memory*, is fast but expensive; it is made from memory chips (see Figure 2). Primary storage has two disadvantages. It is comparatively expensive, and it loses all its data when the power is turned off. *Secondary storage*, usually a *hard disk* (see Figure 3), provides less expensive storage that persists without electricity. A hard disk consists of rotating platters, which are coated with a magnetic material, and read/write heads, which can detect and change the patterns of varying magnetic flux on the platters. This is essentially the same recording and playback process that is used in audio or video tapes.

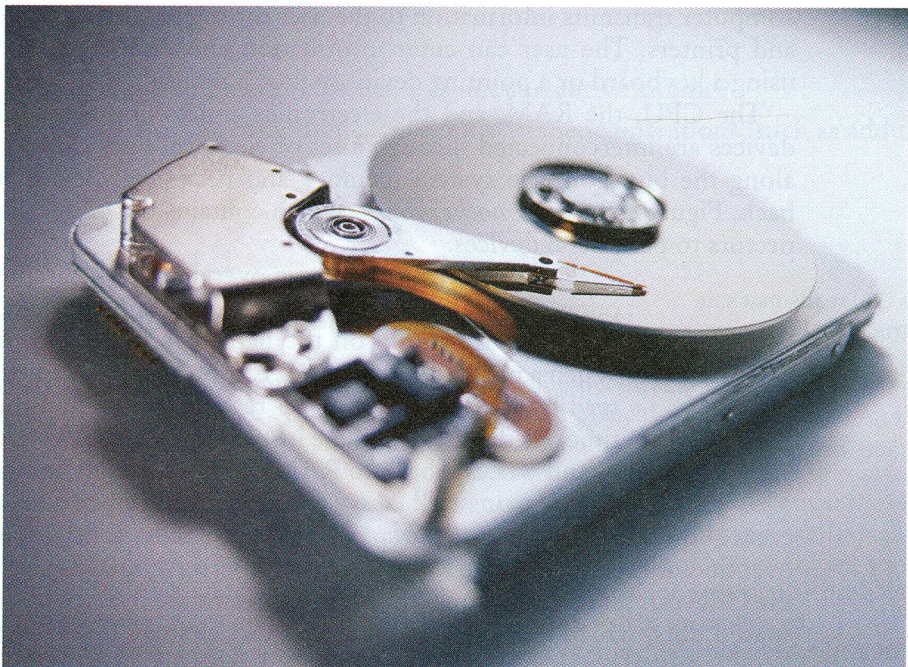
Some computers are self-contained units, whereas others are interconnected through *networks*. Home computers are usually intermittently connected to the Internet via a dialup or broadband connection. The computers in your computer lab are probably permanently connected to a local area network. Through the network cabling, the computer can read programs from central storage locations or

**Figure 2**

A Memory Module with Memory Chips

send data to other computers. For the user of a networked computer, it may not even be obvious which data reside on the computer itself and which are transmitted through the network.

Most computers have *removable storage* devices that can access data or programs on media such as floppy disks, tapes, or compact discs (CDs).

**Figure 3** A Hard Disk

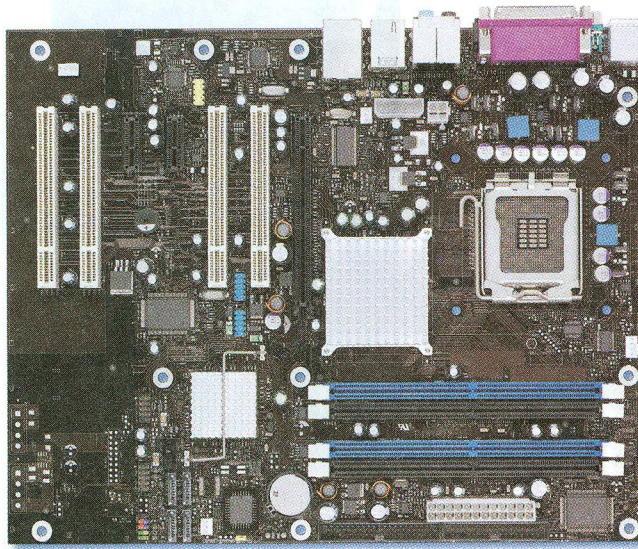


Figure 4 A Motherboard

To interact with a human user, a computer requires other peripheral devices. The computer transmits information to the user through a display screen, loudspeakers, and printers. The user can enter information and directions to the computer by using a keyboard or a pointing device such as a mouse.

The CPU, the RAM, and the electronics controlling the hard disk and other devices are interconnected through a set of electrical lines called a *bus*. Data travel along the bus from the system memory and peripheral devices to the CPU and back. Figure 4 shows a *motherboard*, which contains the CPU, the RAM, and connectors to peripheral devices.

The CPU reads machine instructions from memory. The instructions direct it to communicate with memory, secondary storage, and peripheral devices.

Figure 5 gives a schematic overview of the architecture of a computer. Program instructions and data (such as text, numbers, audio, or video) are stored on the hard disk, on a CD, or on a network. When a program is started, it is brought into memory where it can be read by the CPU. The CPU reads the program one instruction at a time. As directed by these instructions, the CPU reads data, modifies it, and writes it back to RAM or to secondary storage. Some program instructions will cause the CPU to interact with the devices that control the display screen or the speaker. Because these actions happen many times over and at great speed, the human user will perceive images and sound. Similarly, the CPU can send instructions to a printer to mark the paper with patterns of closely spaced dots, which a human recognizes as text characters and pictures. Some program instructions read user input from the keyboard or mouse. The program analyzes the nature of these inputs and then executes the next appropriate instructions.

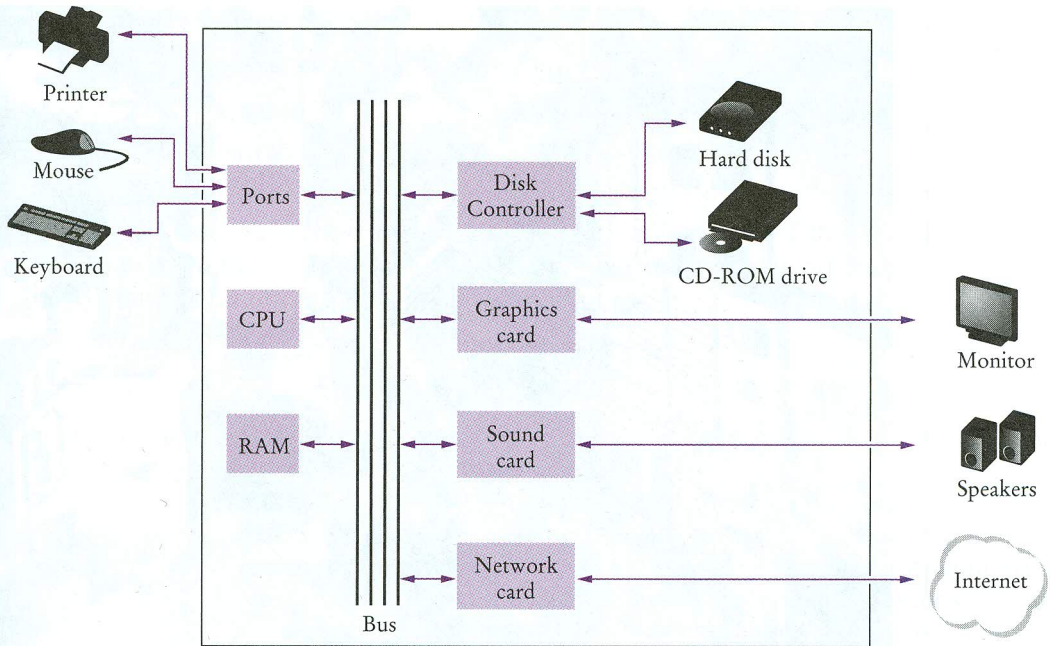


Figure 5 Schematic Diagram of a Computer

SELF CHECK

4. Where is a program stored when it is not currently running?
5. Which part of the computer carries out arithmetic operations, such as addition and multiplication?

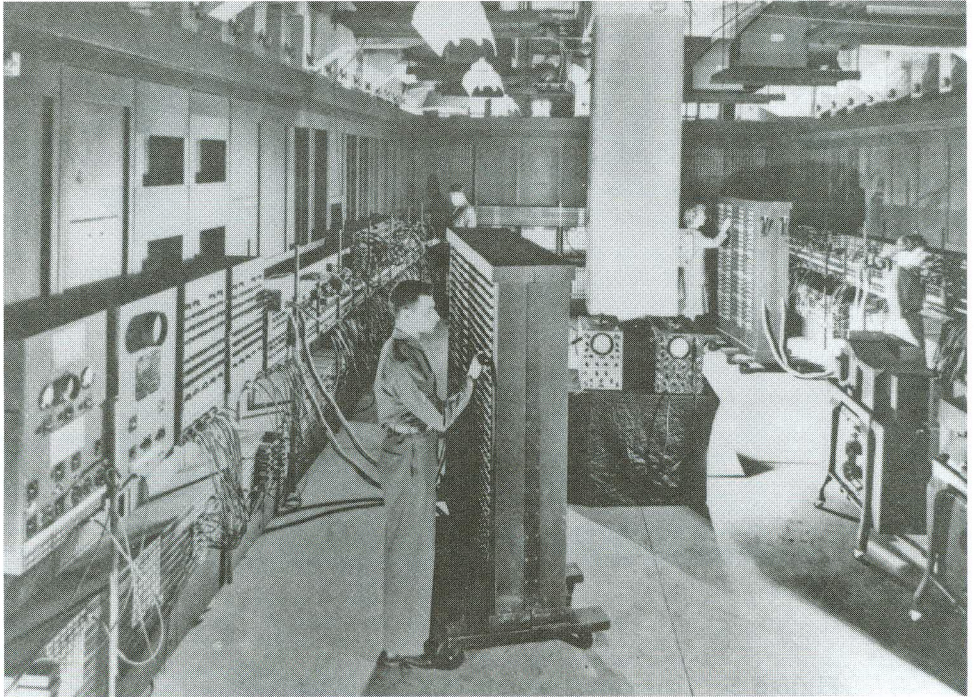


RANDOM FACT 1.1

The ENIAC and the Dawn of Computing

The ENIAC (electronic *numerical integrator and computer*) was the first usable electronic computer. It was designed by J. Presper Eckert and John Mauchly at the University of Pennsylvania and was completed in 1946. Instead of transistors, which were not invented until two years after it was built, the ENIAC contained about 18,000 *vacuum tubes* in many cabinets housed in a large room (see The ENIAC figure). Vacuum tubes burned out at the rate of several tubes per day. An attendant with a shopping cart full of tubes constantly made the rounds and replaced defective ones. The computer was programmed by connecting wires on panels. Each wiring configuration would set up the computer for a particular problem. To have the computer work on a different problem, the wires had to be replugged.

Work on the ENIAC was supported by the U.S. Navy, which was interested in computations of ballistic tables that would give the trajectory of a projectile, depending on the wind resistance, initial velocity, and atmospheric conditions. To compute the trajectories, one must



The ENIAC

find the numerical solutions of certain differential equations; hence the name “numerical integrator”. Before machines like ENIAC were developed, humans did this kind of work, and until the 1950s the word “computer” referred to these people. The ENIAC was later used for peaceful purposes, such as the tabulation of U.S. census data.

1.3 Translating Human-Readable Programs to Machine Code

Generally, machine code depends on the CPU type. However, the instruction set of the Java virtual machine (JVM) can be executed on many CPUs.

On the most basic level, computer instructions are extremely primitive. The processor executes *machine instructions*. CPUs from different vendors, such as the Intel Pentium or the Sun SPARC, have different sets of machine instructions. To enable Java applications to run on multiple CPUs without modification, Java programs contain machine instructions for a so-called “Java virtual machine” (JVM), an idealized CPU that is simulated by a program run on the actual CPU.

The difference between actual and virtual machine instructions is not important—all you need to know is that machine instructions are very simple, are encoded as numbers and stored in memory, and can be executed very quickly.

A typical sequence of machine instructions is

1. Load the contents of memory location 40.
2. Load the value 100.
3. If the first value is greater than the second value, continue with the instruction that is stored in memory location 240.

Actually, machine instructions are encoded as numbers so that they can be stored in memory. On the Java virtual machine, this sequence of instruction is encoded as the sequence of numbers

```
21 40
16 100
163 240
```

When the virtual machine fetches this sequence of numbers, it decodes them and executes the associated sequence of commands.

Because machine instructions are encoded as numbers, it is difficult to write programs in machine code.

How can you communicate the command sequence to the computer? The most direct method is to place the actual numbers into the computer memory. This is, in fact, how the very earliest computers worked. However, a long program is composed of thousands of individual commands, and it is tedious and error-prone to look up the numeric codes for all commands and manually place the codes into memory. As we said before, computers are really good at automating tedious and error-prone activities, and it did not take long for computer programmers to realize that computers could be harnessed to help in the programming process.

High-level languages allow you to describe tasks at a higher conceptual level than machine code.

In the mid-1950s, *high-level* programming languages began to appear. In these languages, the programmer expresses the idea behind the task that needs to be performed, and a special computer program, called a *compiler*, translates the high-level description into machine instructions for a particular processor.

For example, in Java, the high-level programming language that you will use in this book, you might give the following instruction:

```
if (intRate > 100)
    System.out.println("Interest rate error");
```

This means, “If the interest rate is over 100, display an error message”. It is then the job of the compiler program to look at the sequence of characters `if (intRate > 100)` and translate that into

```
21 40 16 100 163 240 . . .
```

A compiler translates programs written in a high-level language into machine code.

Compilers are quite sophisticated programs. They translate logical statements, such as the `if` statement, into sequences of computations, tests, and jumps. They assign memory locations for *variables*—items of information identified by symbolic names—like `intRate`. In this course, we will generally take the existence of a compiler for granted.

If you decide to become a professional computer scientist, you may well learn more about compiler-writing techniques later in your studies.

SELF CHECK

6. What is the code for the Java virtual machine instruction “Load the contents of memory location 100”?
7. Does a person who uses a computer for office work ever run a compiler?

1.4 The Java Programming Language

Java was originally designed for programming consumer devices, but it was first successfully used to write Internet applets.

In 1991, a group led by James Gosling and Patrick Naughton at Sun Microsystems designed a programming language that they code-named “Green” for use in consumer devices, such as intelligent television “set-top” boxes. The language was designed to be simple and architecture neutral, so that it could be executed on a variety of hardware. No customer was ever found for this technology.

Gosling recounts that in 1994 the team realized, “We could write a really cool browser. It was one of the few things in the client/server mainstream that needed some of the weird things we’d done: architecture neutral, real-time, reliable, secure”. Java was introduced to an enthusiastic crowd at the SunWorld exhibition in 1995.

Java was designed to be safe and portable, benefiting both Internet users and students.

Since then, Java has grown at a phenomenal rate. Programmers have embraced the language because it is simpler than its closest rival, C++. In addition, Java has a rich *library* that makes it possible to write portable programs that can bypass proprietary operating systems—a feature that was eagerly sought by those who wanted to be independent of those proprietary systems and was bitterly fought by their vendors. A “micro edition” and an “enterprise edition” of the Java library make Java programmers at home on hardware ranging from smart cards and cell phones to the largest Internet servers.

Because Java was designed for the Internet, it has two attributes that make it very suitable for beginners: safety and portability. If you visit a web page that contains Java code (so-called *applets*—see Figure 6 for an example), the code automatically starts running. It is important that you can trust that applets are inherently safe. If an applet could do something evil, such as damaging data or reading personal information on your computer, then you would be in real danger every time you browsed the Web—an unscrupulous designer might put up a web page containing dangerous code that would execute on your machine as soon as you visited the page. The Java language has an assortment of security features that guarantees that no evil applets can run on your computer. As an added benefit, these features also help you to learn the language faster. The Java virtual machine can catch many kinds of beginners’ mistakes and report them accurately. (In contrast, many beginners’ mistakes in the C++ language merely produce programs that act in random and confusing ways.) The other benefit of Java is portability. The same Java program will run, without change, on Windows, UNIX, Linux, or the Macintosh. This too is a requirement for applets. When you visit a web page, the web server that serves up

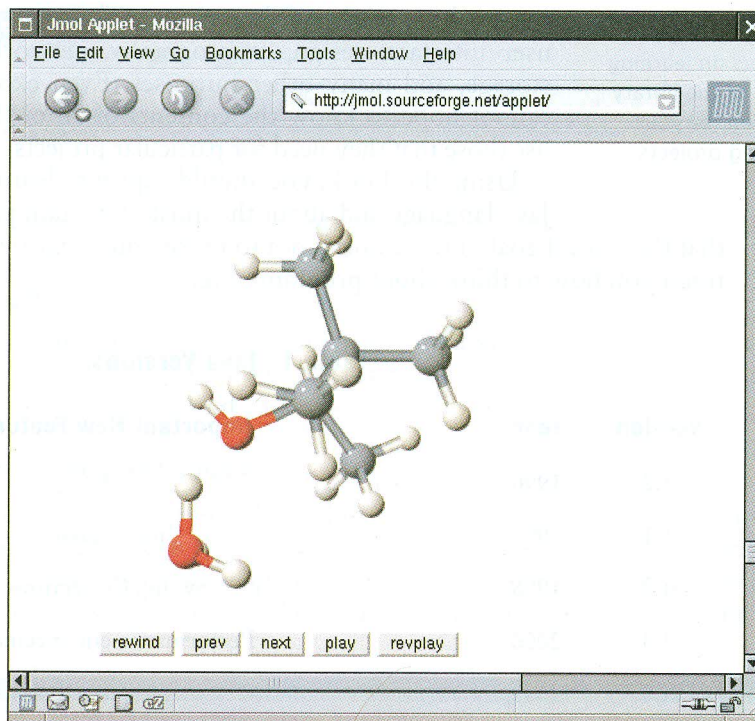


Figure 6 An Applet for Visualizing Molecules ([1])

the page contents has no idea what computer you are using to browse the Web. It simply returns you the portable code that was generated by the Java compiler. The virtual machine on your computer executes that portable code. Again, there is a benefit for the student. You do not have to learn how to write programs for different operating systems.

At this time, Java is firmly established as one of the most important languages for general-purpose programming as well as for computer science instruction. However, although Java is a good language for beginners, it is not perfect, for three reasons.

Because Java was not specifically designed for students, no thought was given to making it really simple to write basic programs. A certain amount of technical machinery is necessary in Java to write even the simplest programs. This is not a problem for professional programmers, but it is a drawback for beginning students. As you learn how to program in Java, there will be times when you will be asked to be satisfied with a preliminary explanation and wait for complete details in a later chapter.

Java was revised and extended many times during its life—see Table 1. In this book, we assume that you have Java version 5 or later.

Finally, you cannot hope to learn all of Java in one semester. The Java language itself is relatively simple, but Java contains a vast set of *library packages* that are

Java has a very large library. Focus on learning those parts of the library that you need for your programming projects.

required to write useful programs. There are packages for graphics, user interface design, cryptography, networking, sound, database storage, and many other purposes. Even expert Java programmers cannot hope to know the contents of all of the packages—they just use those that they need for particular projects.

Using this book, you should expect to learn a good deal about the Java language and about the most important packages. Keep in mind that the central goal of this book is not to make you memorize Java minutiae, but to teach you how to think about programming.

Table 1 Java Versions

Version	Year	Important New Features
1.0	1996	
1.1	1997	Inner classes
1.2	1998	Swing, Collections
1.3	2000	Performance enhancements
1.4	2002	Assertions, XML
5	2004	Generic classes, enhanced for loop, auto-boxing, enumerations
6	2006	Library improvements

SELF CHECK

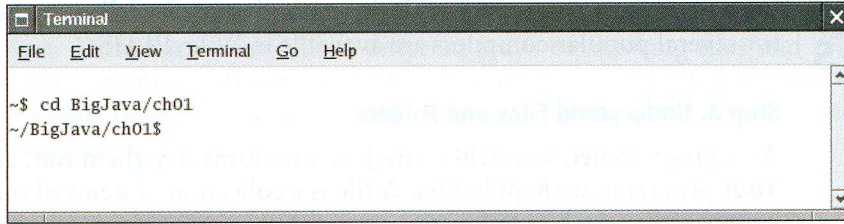
8. What are the two most important benefits of the Java language?
9. How long does it take to learn the entire Java library?

1.5 Becoming Familiar with Your Computer

Set aside some time to become familiar with the computer system and the Java compiler that you will use for your class work.

You may be taking your first programming course as you read this book, and you may well be doing your work on an unfamiliar computer system. Spend some time familiarizing yourself with the computer. Because computer systems vary widely, this book can only give an outline of the steps you need to follow. Using a new and unfamiliar computer system can be frustrating, especially if you are

on your own. Look for training courses that your campus offers, or ask a friend to give you a brief tour.

**Figure 7**

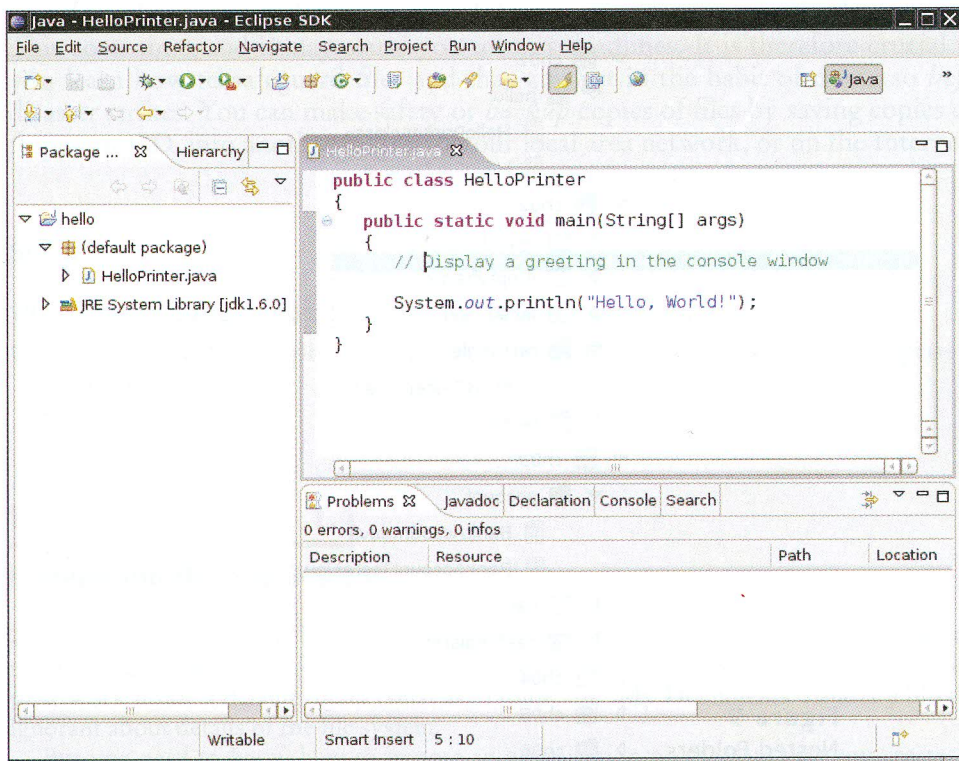
A Shell Window

Step 1. Log In

If you use your home computer, you probably don't need to worry about this step. Computers in a lab, however, are usually not open to everyone. You may need an account name or number and a password to gain access to such a system.

Step 2. Locate the Java Compiler

Computer systems differ greatly in this regard. On some systems you must open a *shell window* (see Figure 7) and type commands to launch the compiler. Other systems have an *integrated development environment* in which you can write and test your programs (see Figure 8). Many university labs have information sheets and

**Figure 8** An Integrated Development Environment



tutorials that walk you through the tools that are installed in the lab. Instructions for several popular compilers are available in WileyPLUS.

Step 3. Understand Files and Folders

As a programmer, you will write Java programs, try them out, and improve them. Your programs are kept in *files*. A file is a collection of items of information that are kept together, such as the text of a word-processing document or the instructions of a Java program. Files have names, and the rules for legal names differ from one system to another. Some systems allow spaces in file names; others don't. Some distinguish between upper- and lowercase letters; others don't. Most Java compilers require that Java files end in an *extension*—.java; for example, Test.java. Java file names cannot contain spaces, and the distinction between upper- and lowercase letters is important.

Files are stored in *folders* or *directories*. These file containers can be *nested*. That is, a folder can contain not only files but also other folders, which themselves can contain more files and folders (see Figure 9). This hierarchy can be quite large, especially on networked computers, where some of the files may be on your local disk, others elsewhere on the network. While you need not be concerned with

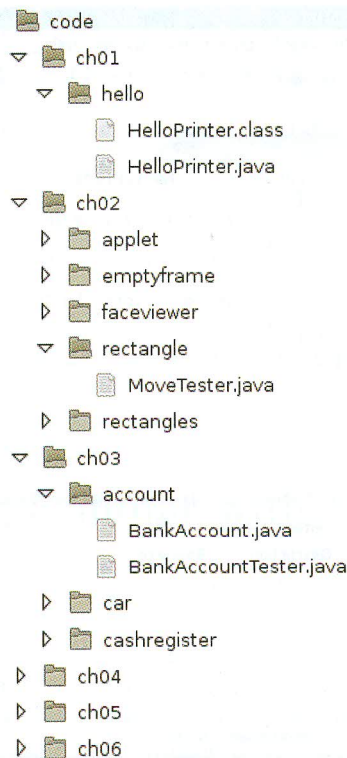


Figure 9
Nested Folders

every branch of the hierarchy, you should familiarize yourself with your local environment. Different systems have different ways of showing files and directories. Some use a graphical display and let you move around by clicking the mouse on folder icons. In other systems, you must enter commands to visit or inspect different locations.

Step 4. Write a Simple Program

In the next section, we will introduce a very simple program. You will need to learn how to type it in, how to run it, and how to fix mistakes.

Step 5. Save Your Work

Develop a strategy for keeping backup copies of your work before disaster strikes.

You will spend many hours typing Java program code and improving it. The resulting program files have some value, and you should treat them as you would other important property. A conscientious safety strategy is particularly important for computer files. They are more fragile than paper documents or other more tangible objects. It is easy to delete a file accidentally, and occasionally files are lost because of a computer malfunction. Unless you keep a copy, you must then retype the contents. Because you probably won't remember the entire file, you will likely find yourself spending almost as much time as you did to enter and improve it in the first place. This costs time, and it may cause you to miss deadlines. It is therefore crucial that you learn how to safeguard files and that you get in the habit of doing so *before* disaster strikes. You can make safety or *backup* copies of files by saving copies on a floppy or CD, into another folder, to your local area network, or on the Internet.

SELF CHECK

10. How are programming projects stored on a computer?
11. What do you do to protect yourself from data loss when you work on programming projects?

PRODUCTIVITY HINT 1.1

Understand the File System

In recent years, computers have become easier to use for home or office users. Many inessential details are now hidden from casual users. For example, many casual users simply place all their work inside a default folder (such as "Home" or "My Documents") and are blissfully ignorant about details of the file system.

But you need to know how to impose an organization on the data that you create. You also need to be able to locate and inspect files that are required for translating and running Java programs.



If you are not comfortable with files and folders, be sure to set aside some time to learn about these concepts. Enroll in a short course, or take a web tutorial. Many free tutorials are available on the Internet, but unfortunately their locations change frequently. Search the Web for “files and folders tutorial” and pick a tutorial that goes beyond the basics.

PRODUCTIVITY HINT 1.2



Have a Backup Strategy

Come up with a strategy for your backups *now*, before you lose any data. Here are a few pointers to keep in mind.

- *Select a backup medium.* Floppy disks are the traditional choice, but they can be unreliable. CD media are more reliable and hold far more information, but they are more expensive. An increasingly popular form of backup is Internet file storage. Many people use two levels of backup: a folder on the hard disk for quick and dirty backups, and a CD-ROM for higher security. (After all, a hard disk can crash—a particularly common problem with laptop computers.)
- *Back up often.* Backing up a file takes only a few seconds, and you will hate yourself if you have to spend many hours recreating work that you easily could have saved.
- *Rotate backups.* Use more than one set of disks or folders for backups, and rotate them. That is, first back up onto the first backup destination, then to the second and third, and then go back to the first. That way you always have three recent backups. Even if one of the floppy disks has a defect, or you messed up one of the backup directories, you can use one of the others.
- *Back up source files only.* The compiler translates the files that you write into files consisting of machine code. There is no need to back up the machine code files, because you can recreate them easily by running the compiler again. Focus your backup activity on those files that represent your effort. That way your backups won't fill up with files that you don't need.
- *Pay attention to the backup direction.* Backing up involves copying files from one place to another. It is important that you do this right—that is, copy from your work location to the backup location. If you do it the wrong way, you will overwrite a newer file with an older version.
- *Check your backups once in a while.* Double-check that your backups are where you think they are. There is nothing more frustrating than finding out that the backups are not there when you need them. This is particularly true if you use a backup program that stores files on an unfamiliar device (such as data tape) or in a compressed format.
- *Relax before restoring.* When you lose a file and need to restore it from backup, you are likely to be in an unhappy, nervous state. Take a deep breath and think through the recovery process before you start. It is not uncommon for an agitated computer user to wipe out the last backup when trying to restore a damaged file.

1.6 Compiling a Simple Program

You are now ready to write and run your first Java program. The traditional choice for the very first program in a new programming language is a program that displays a simple greeting: “Hello, World!”. Let us follow that tradition. Here is the “Hello, World!” program in Java.

ch01/hello/HelloPrinter.java

```
1 public class HelloPrinter
2 {
3     public static void main(String[] args)
4     {
5         // Display a greeting in the console window
6
7         System.out.println("Hello, World!");
8     }
9 }
```

Output

Hello, World!

We will examine this program in a minute. For now, you should make a new program file and call it `HelloPrinter.java`. Enter the program instructions and compile and run the program, following the procedure that is appropriate for your compiler.

Java is case sensitive. You must be careful about distinguishing between upper- and lowercase letters.

Java is *case sensitive*. You must enter upper- and lowercase letters exactly as they appear in the program listing. You cannot type `MAIN` or `PrintLn`. If you are not careful, you will run into problems—see Common Error 1.2.

On the other hand, Java has *free-form layout*. You can use any number of spaces and line breaks to separate words. You can cram as many words as possible into each line,

```
public class HelloPrinter{public static void main(String[]
args){// Display a greeting in the console window
System.out.println("Hello, World!");}}
```

You can even write every word and symbol on a separate line,

```
public
class
HelloPrinter
{
public
static
void
main
(
. . .
```


Lay out your programs so that they are easy to read.

However, good taste dictates that you lay out your programs in a readable fashion. We will give you recommendations for good layout throughout this book. Appendix A contains a summary of our recommendations.

When you run the test program, the message

Hello, World!

will appear somewhere on the screen (see Figures 10 and 11). The exact location depends on your programming environment.

Now that you have seen the program working, it is time to understand its makeup. The first line,

```
public class HelloPrinter
```

Classes are the fundamental building blocks of Java programs.

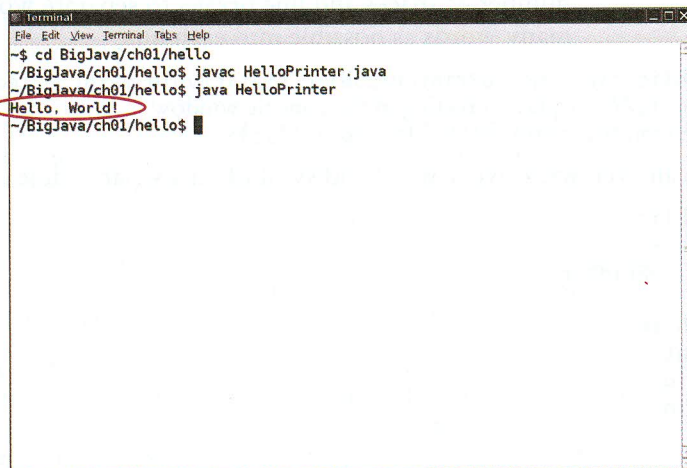
starts a new *class*. Classes are a fundamental concept in Java, and you will begin to study them in Chapter 2. In Java, every program consists of one or more classes.

The keyword `public` denotes that the class is usable by the “public”. You will later encounter private features. At this point, you should

simply regard the

```
public class ClassName
{
    . . .
}
```

as a necessary part of the “plumbing” that is required to write any Java program. In Java, every source file can contain at most one public class, and the name of the public class must match the name of the file containing the class. For example, the class `HelloPrinter` *must* be contained in a file `HelloPrinter.java`. It is very important that the names *and the capitalization* match exactly. You can get strange error messages if you call the class `HELLOPrinter` or the file `helloprinter.java`.



```
Terminal
File Edit View Terminal Tabs Help
~$ cd BigJava/ch01/hello
~/BigJava/ch01/hello$ javac HelloPrinter.java
~/BigJava/ch01/hello$ java HelloPrinter
Hello, World!
~/BigJava/ch01/hello$
```

Figure 10 Running the `HelloPrinter` Program in a Console Window

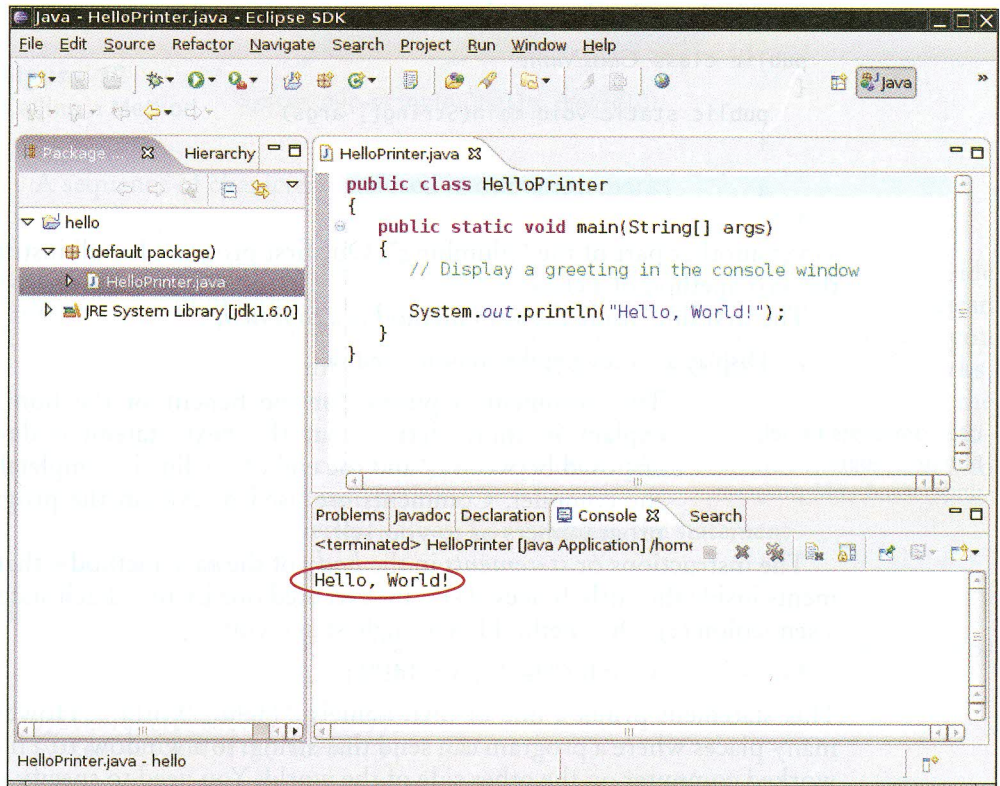


Figure 11

Running the HelloPrinter Program in an Integrated Development Environment

Every Java application contains a class with a main method. When the application starts, the instructions in the main method are executed.

Each class contains definitions of methods. Each method contains a sequence of instructions.

The construction

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

defines a *method* called main. A method contains a collection of programming instructions that describe how to carry out a particular task. Every Java application must have a main method. Most Java programs contain other methods besides main, and you will see in Chapter 3 how to write other methods.

The *parameter* String[] args is a required part of the main method. (It contains *command line arguments*, which we will not discuss until Chapter 11.) The keyword static indicates that the main method does not operate on an *object*. (As you will see in Chapter 2, most methods in Java do operate on objects, and static methods are not common in large Java programs. Nevertheless, main must always be static, because it starts running before the program can create objects.)

At this time, simply consider

```
public class ClassName
{
    public static void main(String[] args)
    {
        . . .
    }
}
```

as yet another part of the “plumbing”. Our first program has all instructions inside the main method of a class.

The first line inside the main method is a *comment*

```
// Display a greeting in the console window
```

Use comments to help human readers understand your program.

This comment is purely for the benefit of the human reader, to explain in more detail what the next statement does. Any text enclosed between `//` and the end of the line is completely ignored by the compiler. Comments are used to explain the program to other programmers or to yourself.

The instructions or *statements* in the *body* of the main method—that is, the statements inside the curly braces (`{}`)—are executed one by one. Each statement ends in a semicolon (`;`). Our method has a single statement:

```
System.out.println("Hello, World!");
```

This statement prints a line of text, namely “Hello, World!”. However, there are many places where a program can send that string: to a window, to a file, or to a networked computer on the other side of the world. You need to specify that the destination for the string is the *system output*—that is, a console window. The console window is represented in Java by an object called `out`. Just as you needed to place the main method in a `HelloPrinter` class, the designers of the Java library needed to place the `out` object into a class. They placed it in the `System` class, which contains useful objects and methods to access system resources. To use the `out` object in the `System` class, you must refer to it as `System.out`.

To use an object, such as `System.out`, you specify what you want to do to it. In this case, you want to print a line of text. The `println` method carries out this task.

You do not have to implement this method—the programmers who wrote the Java library already did that for us—but you do need to *call* the method.

Whenever you call a method in Java, you need to specify three items (see Figure 12):

A method is called by specifying an object, the method name, and the method parameters.

1. The object that you want to use (in this case, `System.out`)
2. The name of the method you want to use (in this case, `println`)
3. A pair of parentheses, containing any other information the method needs (in this case, `"Hello, World!"`). The technical term for this information is a *parameter* for the method. Note that the two periods in `System.out.println` have different meanings. The first period means “locate the `out` object in the `System` class”. The second period means “apply the `println` method to that object”.

Figure 12

Calling a Method

Object Method Parameters

`System.out.println("Hello, World!")`

A sequence of characters enclosed in quotation marks

`"Hello, World!"`

A string is a sequence of characters enclosed in quotation marks.

is called a *string*. You must enclose the contents of the string inside quotation marks so that the compiler knows you literally mean `"Hello, World!"`. There is a reason for this requirement. Suppose you need to print the word *main*. By enclosing it in quotation marks, `"main"`, the compiler knows you mean the sequence of characters `m a i n`, not the method named `main`. The rule is simply that you must enclose all text strings in quotation marks, so that the compiler considers them plain text and does not try to interpret them as program instructions.

You can also print numerical values. For example, the statement

```
System.out.println(3 + 4);
```

displays the number 7.

The `println` method prints a string or a number and then starts a new line. For example, the sequence of statements

```
System.out.println("Hello");
System.out.println("World!");
```

prints two lines of text:

```
Hello
World!
```

There is a second method, called `print`, that you can use to print an item without starting a new line. For example, the output of the two statements

```
System.out.print("00");
System.out.println(3 + 4);
```

is the single line

```
007
```

SYNTAX 1.1 Method Call

object.methodName(parameters)

Example:

```
System.out.println("Hello, Dave!")
```

Purpose:

To invoke a method on an object and supply any additional parameters

SELF CHECK

12. How would you modify the `HelloPrinter` program to print the words “Hello,” and “World!” on two lines?
13. Would the program continue to work if you omitted the line starting with `///
14. What does the following set of statements print?`

```
System.out.print("My lucky number is");  
System.out.println(3 + 4 + 5);
```

COMMON ERROR 1.1**Omitting Semicolons**

In Java every statement must end in a semicolon. Forgetting to type a semicolon is a common error. It confuses the compiler, because the compiler uses the semicolon to find where one statement ends and the next one starts. The compiler does not use line breaks or closing braces to recognize the end of statements. For example, the compiler considers

```
System.out.println("Hello")  
System.out.println("World!");
```

a single statement, as if you had written

```
System.out.println("Hello") System.out.println("World!");
```

Then it doesn't understand that statement, because it does not expect the word `System` following the closing parenthesis after "Hello". The remedy is simple. Scan every statement for a terminating semicolon, just as you would check that every English sentence ends in a period.

ADVANCED TOPIC 1.1**Alternative Comment Syntax**

In Java there are two methods for writing comments. You already learned that the compiler ignores anything that you type between `//` and the end of the current line. The compiler also ignores any text between a `/*` and `*/`.

```
/* A simple Java program */
```

The `//` comment is easier to type if the comment is only a single line long. If you have a comment that is longer than a line, then the `/* . . . */` comment is simpler:

```
/*  
    This is a simple Java program that you can use to try out  
    your compiler and virtual machine.  
*/
```

It would be somewhat tedious to add the `//` at the beginning of each line and to move them around whenever the text of the comment changes.

In this book, we use `//` for comments that will never grow beyond a line, and `/* . . . */` for longer comments. If you prefer, you can always use the `//` style. The readers of your code will be grateful for *any* comments, no matter which style you use.

1.7 Errors

Experiment a little with the `HelloPrinter` program. What happens if you make a typing error such as

```
System.ouch.println("Hello, World!");  
System.out.println("Hello, World!");  
System.out.println("Hello, Word!");
```

A syntax error is a violation of the rules of the programming language. The compiler detects syntax errors.

In the first case, the compiler will complain. It will say that it has no clue what you mean by `ouch`. The exact wording of the error message is dependent on the compiler, but it might be something like “Undefined symbol `ouch`”. This is a *compile-time error* or *syntax error*. Something is wrong according to the language rules and the compiler finds it. When the compiler finds one or more errors, it refuses to

translate the program to Java virtual machine instructions, and as a consequence you have no program that you can run. You must fix the error and compile again. In fact, the compiler is quite picky, and it is common to go through several rounds of fixing compile-time errors before compilation succeeds for the first time.

If the compiler finds an error, it will not simply stop and give up. It will try to report as many errors as it can find, so you can fix them all at once. Sometimes, however, one error throws it off track. This is likely to happen with the error in the second line. Because the closing quotation mark is missing, the compiler will think that the `);` characters are still part of the string. In such cases, it is common for the compiler to emit bogus error reports for neighboring lines. You should fix only those error messages that make sense to you and then recompile.

The error in the third line is of a different kind. The program will compile and run, but its output will be wrong. It will print

```
Hello, Word!
```

A logic error causes a program to take an action that the programmer did not intend. You must test your programs to find logic errors.

This is a *run-time error* or *logic error*. The program is syntactically correct and does something, but it doesn’t do what it is supposed to do. The compiler cannot find the error. You, the programmer, must flush out this type of error. Run the program, and carefully look at its output.

During program development, errors are unavoidable. Once a program is longer than a few lines, it requires superhuman concentration to enter it correctly without slipping up once. You will find yourself omitting semicolons or quotes more often than you would like, but the compiler will track down these problems for you.

Logic errors are more troublesome. The compiler will not find them—in fact, the compiler will cheerfully translate any program as long as its syntax is correct—but

the resulting program will do something wrong. It is the responsibility of the program author to test the program and find any logic errors. Testing programs is an important topic that you will encounter many times in this book. Another important aspect of good craftsmanship is *defensive programming*: structuring programs and development processes in such a way that an error in one part of a program does not trigger a disastrous response.

The error examples that you saw so far were not difficult to diagnose or fix, but as you learn more sophisticated programming techniques, there will also be much more room for error. It is an uncomfortable fact that locating all errors in a program is very difficult. Even if you can observe that a program exhibits faulty behavior, it may not at all be obvious what part of the program caused it and how you can fix it. Special software tools (so-called *debuggers*) let you trace through a program to find *bugs*—that is, logic errors. In Chapter 6 you will learn how to use a debugger effectively.

Note that these errors are different from the types of errors that you are likely to make in calculations. If you total up a column of numbers, you may miss a minus sign or accidentally drop a carry, perhaps because you are bored or tired. Computers do not make these kinds of errors.

This book uses a three-part error management strategy. First, you will learn about common errors and how to avoid them. Then you will learn defensive programming strategies to minimize the likelihood and impact of errors. Finally, you will learn debugging strategies to flush out those errors that remain.

SELF CHECK

15. Suppose you omit the `//` characters from the `HelloPrinter.java` program but not the remainder of the comment. Will you get a compile-time error or a run-time error?
16. How can you find logic errors in a program?

COMMON ERROR 1.2



Misspelling Words

If you accidentally misspell a word, then strange things may happen, and it may not always be completely obvious from the error messages what went wrong. Here is a good example of how simple spelling errors can cause trouble:

```
public class HelloPrinter
{
    public static void Main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

This class defines a method called `Main`. The compiler will not consider this to be the same as the `main` method, because `Main` starts with an uppercase letter and the Java language is case sensitive. Upper- and lowercase letters are considered to be completely different from each other, and to the compiler `Main` is no better match for `main` than `rain`. The compiler will cheerfully compile your `Main` method, but when the Java virtual machine reads the compiled file, it will complain about the missing `main` method and refuse to run the program. Of course, the message “missing main method” should give you a clue where to look for the error.

If you get an error message that seems to indicate that the compiler is on the wrong track, it is a good idea to check for spelling and capitalization. All Java keywords use only lowercase letters. Names of classes usually start with an uppercase letter; names of methods and variables with a lowercase letter. If you misspell the name of a symbol (for example, `ouch` instead of `out`), the compiler will complain about an “undefined symbol”. That error message is usually a good clue that you made a spelling error.

1.8 The Compilation Process

Some Java development environments are very convenient to use. Enter the code in one window, click on a button to compile, and click on another button to execute your program. Error messages show up in a second window, and the program runs in a third window. With such an environment you are completely shielded from the details of the compilation process. On other systems you must carry out every step manually, by typing commands into a shell window.

An editor is a program for entering and modifying text, such as a Java program.

No matter which compilation environment you use, you begin your activity by typing in the program statements. The program that you use for entering and modifying the program text is called an *editor*. Remember to *save* your work to disk frequently, because otherwise the text editor stores the text only in the computer’s memory. If

something goes wrong with the computer and you need to restart it, the contents of the primary memory (including your program text) are lost, but anything stored on the hard disk is permanent even if you need to restart the computer.

The Java compiler translates source code into class files that contain instructions for the Java virtual machine.

When you compile your program, the compiler translates the Java *source code* (that is, the statements that you wrote) into *class files*, which consist of virtual machine instructions and other information that is required for execution. The class files have the extension `.class`. For example, the virtual machine instructions for the `HelloPrinter` program are stored in a file `HelloPrinter.class`. As already

mentioned, the compiler produces a class file only after you have corrected all syntax errors.

The class file contains the translation of only the instructions that you wrote. That is not enough to actually run the program. To display a string in a window, quite a bit of low-level activity is necessary. The authors of the `System` and `PrintStream` classes (which define the `out` object and the `println` method) have implemented all necessary actions and placed the required class files into a *library*. A

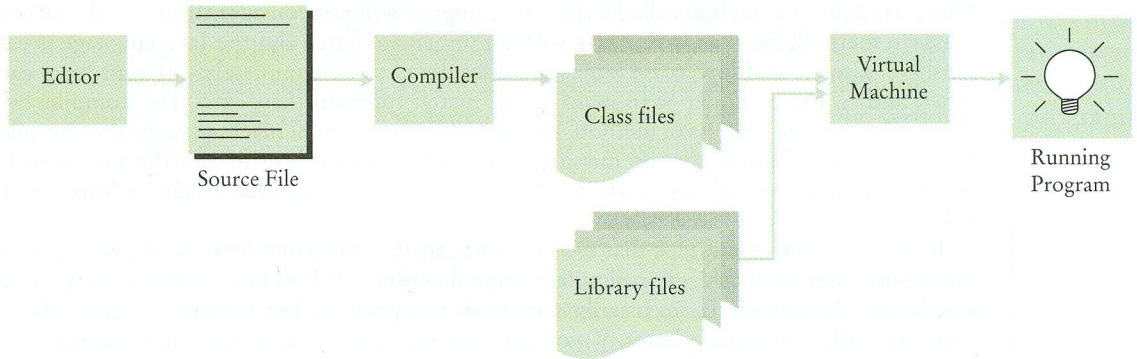


Figure 13 From Source Code to Running Program

library is a collection of code that has been programmed and translated by someone else, ready for you to use in your program.

The Java virtual machine loads the instructions for the program that you wrote, starts your program, and loads the necessary library files as they are required.

The steps of compiling and running your program are outlined in Figure 13.

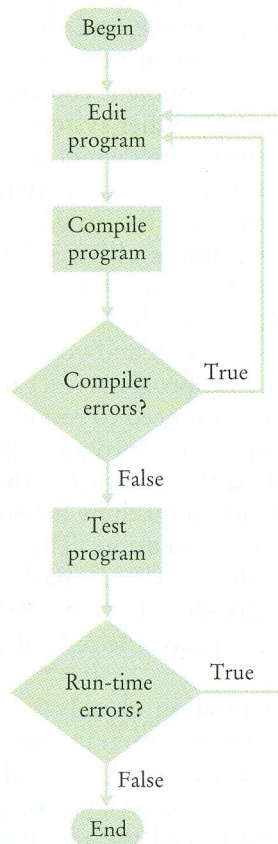


Figure 14

The Edit-Compile-Test Loop

The Java virtual machine loads program instructions from class files and library files.

Programming activity centers around these steps. Start in the editor, writing the source file. Compile the program and look at the error messages. Go back to the editor and fix the syntax errors. When the compiler succeeds, run the program. If you find a run-time error, you must look at the source code in the editor to try to determine the reason.

Once you find the cause of the error, fix it in the editor. Compile and run again to see whether the error has gone away. If not, go back to the editor. This is called the *edit-compile-test loop* (see Figure 14). You will spend a substantial amount of time in this loop when working on programming assignments.

SELF CHECK

17. What do you expect to see when you load a class file into your text editor?
18. Why can't you test a program for run-time errors when it has compiler errors?

CHAPTER SUMMARY

1. A computer must be programmed to perform tasks. Different tasks require different programs.
2. A computer program executes a sequence of very basic operations in rapid succession.
3. A computer program contains the instruction sequences for all tasks that it can execute.
4. At the heart of the computer lies the central processing unit (CPU).
5. Data and programs are stored in primary storage (memory) and secondary storage (such as a hard disk).
6. The CPU reads machine instructions from memory. The instructions direct it to communicate with memory, secondary storage, and peripheral devices.
7. Generally, machine code depends on the CPU type. However, the instruction set of the Java virtual machine (JVM) can be executed on many CPUs.
8. Because machine instructions are encoded as numbers, it is difficult to write programs in machine code.
9. High-level languages allow you to describe tasks at a higher conceptual level than machine code.
10. A compiler translates programs written in a high-level language into machine code.
11. Java was originally designed for programming consumer devices, but it was first successfully used to write Internet applets.

12. Java was designed to be safe and portable, benefiting both Internet users and students.
13. Java has a very large library. Focus on learning those parts of the library that you need for your programming projects.
14. Set aside some time to become familiar with the computer system and the Java compiler that you will use for your class work.
15. Develop a strategy for keeping backup copies of your work before disaster strikes.
16. Java is case sensitive. You must be careful about distinguishing between upper- and lowercase letters.
17. Lay out your programs so that they are easy to read.
18. Classes are the fundamental building blocks of Java programs.
19. Every Java application contains a class with a `main` method. When the application starts, the instructions in the `main` method are executed.
20. Each class contains definitions of methods. Each method contains a sequence of instructions.
21. Use comments to help human readers understand your program.
22. A method is called by specifying an object, the method name, and the method parameters.
23. A string is a sequence of characters enclosed in quotation marks.
24. A syntax error is a violation of the rules of the programming language. The compiler detects syntax errors.
25. A logic error causes a program to take an action that the programmer did not intend. You must test your programs to find logic errors.
26. An editor is a program for entering and modifying text, such as a Java program.
27. The Java compiler translates source code into class files that contain instructions for the Java virtual machine.
28. The Java virtual machine loads program instructions from class files and library files.

FURTHER READING

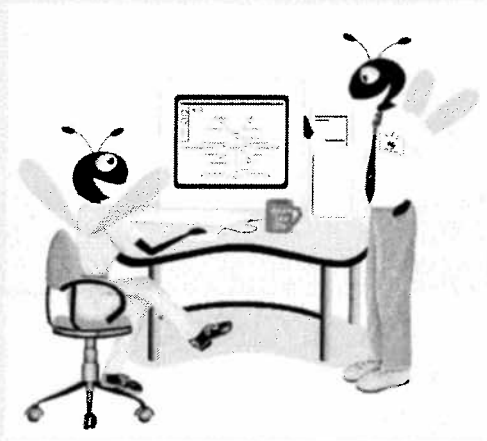
1. <http://jmol.sourceforge.net/applet/> The web site for the jmol applet for visualizing molecules.



Number Systems (on CD)

Objectives

- To understand basic number systems concepts such as base, positional value, and symbol value.
- To understand how to work with numbers represented in the binary, octal, and hexadecimal number systems
- To be able to abbreviate binary numbers as octal numbers or hexadecimal numbers.
- To be able to convert octal numbers and hexadecimal numbers to binary numbers.
- To be able to convert back and forth between decimal numbers and their binary, octal, and hexadecimal equivalents.
- To understand binary arithmetic, and how negative binary numbers are represented using two's complement notation.



Here are only numbers ratified.

William Shakespeare

Nature has some sort of arithmetic-geometrical coordinate system, because nature has all kinds of models. What we experience of nature is in models, and all of nature's models are so beautiful.

It struck me that nature's system must be a real beauty, because in chemistry we find that the associations are always in beautiful whole numbers—there are no fractions.

Richard Buckminster Fuller

Outline

- E.1 Introduction**
- E.2 Abbreviating Binary Numbers as Octal Numbers and Hexadecimal Numbers**
- E.3 Converting Octal Numbers and Hexadecimal Numbers to Binary Numbers**
- E.4 Converting from Binary, Octal, or Hexadecimal to Decimal**
- E.5 Converting from Decimal to Binary, Octal, or Hexadecimal**
- E.6 Negative Binary Numbers: Two's Complement Notation**

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

E.1 Introduction

In this appendix, we introduce the key number systems that Java programmers use, especially when they are working on software projects that require close interaction with “machine-level” hardware. Projects like this include operating systems, computer networking software, compilers, database systems, and applications requiring high performance.

When we write an integer such as 227 or -63 in a Java program, the number is assumed to be in the decimal (base 10) number system. The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The lowest digit is 0 and the highest digit is 9—one less than the base of 10. Internally, computers use the binary (base 2) number system. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest digit is 1—one less than the base of 2.

As we will see, binary numbers tend to be much longer than their decimal equivalents. Programmers who work in assembly languages and in high-level languages like Java that enable programmers to reach down to the “machine level,” find it cumbersome to work with binary numbers. So two other number systems the octal number system (base 8) and the hexadecimal number system (base 16)—are popular primarily because they make it convenient to abbreviate binary numbers.

In the octal number system, the digits range from 0 to 7. Because both the binary number system and the octal number system have fewer digits than the decimal number system, their digits are the same as the corresponding digits in decimal.

The hexadecimal number system poses a problem because it requires sixteen digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15. Thus in hexadecimal we can have numbers like 876 consisting solely of decimal-like digits, numbers like 8A55F consisting of digits and letters, and numbers like FFE consisting solely of letters. Occasionally, a hexadecimal number spells a common word such as FACE or FEED—this can appear strange to programmers accustomed to working with numbers.

Each of these number systems uses positional notation—each position in which a digit is written has a different positional value. For example, in the decimal number 937 (the 9, the 3, and the 7 are referred to as symbol values), we say that the 7 is written in the ones position, the 3 is written in the tens position, and the 9 is written in the hundreds position.

Notice that each of these positions is a power of the base (base 10), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. E.3).

Binary digit	Octal digit	Decimal digit	Hexadecimal digit
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (decimal value of 10)
			B (decimal value of 11)
			C (decimal value of 12)
			D (decimal value of 13)
			E (decimal value of 14)
			F (decimal value of 15)

Fig. E.1 Digits of the binary, octal, decimal and hexadecimal number systems.

Attribute	Binary	Octal	Decimal	Hexadecimal
Base	2	8	10	16
Lowest digit	0	0	0	0
Highest digit	1	7	9	F

Fig. E.2 Comparing the binary, octal, decimal and hexadecimal number systems.

Positional values in the decimal number system			
Decimal digit	9	3	7
Position name	Hundreds	Tens	Ones
Positional value	100	10	1
Positional value as a power of the base (10)	10 ²	10 ¹	10 ⁰

Fig. E.3 Positional values in the decimal number system.

For longer decimal numbers, the next positions to the left would be the thousands position (10 to the 3rd power), the ten-thousands position (10 to the 4th power), the hundred-thousands position (10 to the 5th power), the millions position (10 to the 6th power), the ten-millions position (10 to the 7th power), and so on.

In the binary number 101, we say that the rightmost 1 is written in the ones position, the 0 is written in the twos position, and the leftmost 1 is written in the fours position. Notice that each of these positions is a power of the base (base 2), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig E.4).

For longer binary numbers, the next positions to the left would be the eights position (2 to the 3rd power), the sixteens position (2 to the 4th power), the thirty-twos position (2 to the 5th power), the sixty-fours position (2 to the 6th power), and so on.

In the octal number 425, we say that the 5 is written in the ones position, the 2 is written in the eights position, and the 4 is written in the sixty-fours position. Notice that each of these positions is a power of the base (base 8), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. E.5).

For longer octal numbers, the next positions to the left would be the five-hundred-and-twelves position (8 to the 3rd power), the four-thousand-and-ninety-sixes position (8 to the 4th power), the thirty-two-thousand-seven-hundred-and-sixty eights position (8 to the 5th power), and so on.

In the hexadecimal number 3DA, we say that the A is written in the ones position, the D is written in the sixteens position, and the 3 is written in the two-hundred-and-fifty-sixes position. Notice that each of these positions is a power of the base (base 16), and that these powers begin at 0 and increase by 1 as we move left in the number (Fig. E.6).

Positional values in the binary number system

Binary digit	1	0	1
Position name	Fours	Twos	Ones
Positional value	4	2	1
Positional value as a power of the base (2)	2^2	2^1	2^0

Fig. E.4 Positional values in the binary number system.

Positional values in the octal number system

Decimal digit	4	2	5
Position name	Sixty-fours	Eights	Ones
Positional value	64	8	1
Positional value as a power of the base (8)	8^2	8^1	8^0

Fig. E.5 Positional values in the octal number system.

Positional values in the hexadecimal number system			
Decimal digit	3	D	A
Position name	Two-hundred-and-fifty-sixes	Sixteens	Ones
Positional value	256	16	1
Positional value as a power of the base (16)	16²	16¹	16⁰

Fig. E.6 Positional values in the hexadecimal number system.

For longer hexadecimal numbers, the next positions to the left would be the four-thousand-and-ninety-sixes position (16 to the 3rd power), the sixty-five-thousand-five-hundred-and-thirty-six position (16 to the 4th power), and so on.

E.2 Abbreviating Binary Numbers as Octal Numbers and Hexadecimal Numbers

The main use for octal and hexadecimal numbers in computing is for abbreviating lengthy binary representations. Figure E.7 highlights the fact that lengthy binary numbers can be expressed concisely in number systems with higher bases than the binary number system.

Decimal number	Binary representation	Octal representation	Hexadecimal representation
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F
16	10000	20	10

Fig. E.7 Decimal, binary, octal, and hexadecimal equivalents.

A particularly important relationship that both the octal number system and the hexadecimal number system have to the binary system is that the bases of octal and hexadecimal (8 and 16 respectively) are powers of the base of the binary number system (base 2). Consider the following 12-digit binary number and its octal and hexadecimal equivalents. See if you can determine how this relationship makes it convenient to abbreviate binary numbers in octal or hexadecimal. The answer follows the numbers.

Binary Number	Octal equivalent	Hexadecimal equivalent
100011010001	4321	8D1

To see how the binary number converts easily to octal, simply break the 12-digit binary number into groups of three consecutive bits each, and write those groups over the corresponding digits of the octal number as follows

100	011	010	001
4	3	2	1

Notice that the octal digit you have written under each group of three bits corresponds precisely to the octal equivalent of that 3-digit binary number as shown in Fig. E.7.

The same kind of relationship may be observed in converting numbers from binary to hexadecimal. In particular, break the 12-digit binary number into groups of four consecutive bits each and write those groups over the corresponding digits of the hexadecimal number as follows

1000	1101	0001
8	D	1

Notice that the hexadecimal digit you wrote under each group of four bits corresponds precisely to the hexadecimal equivalent of that 4-digit binary number as shown in Fig. E.7.

E.3 Converting Octal Numbers and Hexadecimal Numbers to Binary Numbers

In the previous section, we saw how to convert binary numbers to their octal and hexadecimal equivalents by forming groups of binary digits and simply rewriting these groups as their equivalent octal digit values or hexadecimal digit values. This process may be used in reverse to produce the binary equivalent of a given octal or hexadecimal number.

For example, the octal number 653 is converted to binary simply by writing the 6 as its 3-digit binary equivalent 110, the 5 as its 3-digit binary equivalent 101, and the 3 as its 3-digit binary equivalent 011 to form the 9-digit binary number 110101011.

The hexadecimal number FAD5 is converted to binary simply by writing the F as its 4-digit binary equivalent 1111, the A as its 4-digit binary equivalent 1010, the D as its 4-digit binary equivalent 1101, and the 5 as its 4-digit binary equivalent 0101 to form the 16-digit 1111101011010101.

E.4 Converting from Binary, Octal, or Hexadecimal to Decimal

Because we are accustomed to working in decimal, it is often convenient to convert a binary, octal, or hexadecimal number to decimal to get a sense of what the number is “really” worth. Our diagrams in Section E.1 express the positional values in decimal. To convert a number to decimal from another base, multiply the decimal equivalent of each digit by its

positional value, and sum these products. For example, the binary number 110101 is converted to decimal 53 as shown in Fig. E.8.

To convert octal 7614 to decimal 3980, we use the same technique, this time using appropriate octal positional values as shown in Fig. E.9.

To convert hexadecimal AD3B to decimal 44347, we use the same technique, this time using appropriate hexadecimal positional values as shown in Fig. E.10.

E.5 Converting from Decimal to Binary, Octal, or Hexadecimal

The conversions of the previous section follow naturally from the positional notation conventions. Converting from decimal to binary, octal, or hexadecimal also follows these conventions.

Suppose we wish to convert decimal 57 to binary. We begin by writing the positional values of the columns right to left until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Converting a binary number to decimal						
Positional values:	32	16	8	4	2	1
Symbol values:	1	1	0	1	0	1
Products:	$1 \cdot 32 = 32$	$1 \cdot 16 = 16$	$0 \cdot 8 = 0$	$1 \cdot 4 = 4$	$0 \cdot 2 = 0$	$1 \cdot 1 = 1$
Sum:	$= 32 + 16 + 0 + 4 + 0 + 1 = 53$					

Fig. E.8 Converting a binary number to decimal.

Converting an octal number to decimal				
Positional values:	512	64	8	1
Symbol values:	7	6	1	4
Products	$7 \cdot 512 = 3584$	$6 \cdot 64 = 384$	$1 \cdot 8 = 8$	$4 \cdot 1 = 4$
Sum:	$= 3584 + 384 + 8 + 4 = 3980$			

Fig. E.9 Converting an octal number to decimal.

Converting a hexadecimal number to decimal				
Positional values:	4096	256	16	1
Symbol values:	A	D	3	B
Products	$A \cdot 4096 = 40960$	$D \cdot 256 = 3328$	$3 \cdot 16 = 48$	$B \cdot 1 = 11$

Fig. E.10 Converting a hexadecimal number to decimal.

Converting a hexadecimal number to decimal

$$\text{Sum:} \quad = 40960 + 3328 + 48 + 11 = 44347$$

Fig. E.10 Converting a hexadecimal number to decimal.

Positional values: **64 32 16 8 4 2 1**

Then we discard the column with positional value 64 leaving:

Positional values: **32 16 8 4 2 1**

Next we work from the leftmost column to the right. We divide 32 into 57 and observe that there is one 32 in 57 with a remainder of 25, so we write 1 in the 32 column. We divide 16 into 25 and observe that there is one 16 in 25 with a remainder of 9 and write 1 in the 16 column. We divide 8 into 9 and observe that there is one 8 in 9 with a remainder of 1. The next two columns each produce quotients of zero when their positional values are divided into 1 so we write 0s in the 4 and 2 columns. Finally, 1 into 1 is 1 so we write 1 in the 1 column. This yields:

Positional values:	32	16	8	4	2	1
Symbol values:	1	1	1	0	0	1

and thus decimal 57 is equivalent to binary 111001.

To convert decimal 103 to octal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write:

Positional values: **512 64 8 1**

Then we discard the column with positional value 512, yielding:

Positional values: **64 8 1**

Next we work from the leftmost column to the right. We divide 64 into 103 and observe that there is one 64 in 103 with a remainder of 39, so we write 1 in the 64 column. We divide 8 into 39 and observe that there are four 8s in 39 with a remainder of 7 and write 4 in the 8 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder so we write 7 in the 1 column. This yields:

Positional values:	64	8	1
Symbol values:	1	4	7

and thus decimal 103 is equivalent to octal 147.

To convert decimal 375 to hexadecimal, we begin by writing the positional values of the columns until we reach a column whose positional value is greater than the decimal number. We do not need that column, so we discard it. Thus, we first write

Positional values: **4096 256 16 1**

Then we discard the column with positional value 4096, yielding:

Positional values: **256 16 1**

Next we work from the leftmost column to the right. We divide 256 into 375 and observe that there is one 256 in 375 with a remainder of 119, so we write 1 in the 256 column. We divide 16 into 119 and observe that there are seven 16s in 119 with a remainder of 7 and write 7 in the 16 column. Finally, we divide 1 into 7 and observe that there are seven 1s in 7 with no remainder so we write 7 in the 1 column. This yields:

Positional values: **256 16 1**
 Symbol values: **1 7 7**

and thus decimal 375 is equivalent to hexadecimal 177.

E.6 Negative Binary Numbers: Two's Complement Notation

The discussion in this appendix has been focussed on positive numbers. In this section, we explain how computers represent negative numbers using *two's complement notation*. First we explain how the two's complement of a binary number is formed, and then we show why it represents the negative value of the given binary number.

Consider a machine with 32-bit integers. Suppose

```
int value = 13;
```

The 32-bit representation of **value** is

```
00000000 00000000 00000000 00001101
```

To form the negative of **value** we first form its *one's complement* by applying Java's bit-wise complement operator (~):

```
onesComplementOfValue = ~value;
```

Internally, **~value** is now **value** with each of its bits reversed—ones become zeros and zeros become ones as follows:

```
value:  
00000000 00000000 00000000 00001101  
  
~value (i.e., value's ones complement):  
11111111 11111111 11111111 11110010
```

To form the two's complement of **value** we simply add one to **value's** one's complement. Thus

```
Two's complement of value:  
11111111 11111111 11111111 11110011
```

Now if this is in fact equal to -13, we should be able to add it to binary 13 and obtain a result of 0. Let us try this:

```
00000000 00000000 00000000 00001101  
+11111111 11111111 11111111 11110011  
-----  
00000000 00000000 00000000 00000000
```


The carry bit coming out of the leftmost column is discarded and we indeed get zero as a result. If we add the one's complement of a number to the number, the result would be all 1s. The key to getting a result of all zeros is that the two's complement is 1 more than the one's complement. The addition of 1 causes each column to add to 0 with a carry of 1. The carry keeps moving leftward until it is discarded from the leftmost bit, and hence the resulting number is all zeros.

Computers actually perform a subtraction such as

```
x = a - value;
```

by adding the two's complement of **value** to **a** as follows:

```
x = a + (~value + 1);
```

Suppose **a** is 27 and **value** is 13 as before. If the two's complement of **value** is actually the negative of **value**, then adding the two's complement of **value** to **a** should produce the result 14. Let us try this:

a (i.e., 27)	00000000 00000000 00000000 00011011
+ (~value + 1)	+11111111 11111111 11111111 11110011

	00000000 00000000 00000000 00001110

which is indeed equal to 14.

SUMMARY

- When we write an integer such as 19 or 227 or -63 in a Java program, the number is automatically assumed to be in the decimal (base 10) number system. The digits in the decimal number system are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. The lowest digit is 0 and the highest digit is 9—one less than the base of 10.
- Internally, computers use the binary (base 2) number system. The binary number system has only two digits, namely 0 and 1. Its lowest digit is 0 and its highest digit is 1—one less than the base of 2.
- The octal number system (base 8) and the hexadecimal number system (base 16) are popular primarily because they make it convenient to abbreviate binary numbers.
- The digits of the octal number system range from 0 to 7.
- The hexadecimal number system poses a problem because it requires sixteen digits—a lowest digit of 0 and a highest digit with a value equivalent to decimal 15 (one less than the base of 16). By convention, we use the letters A through F to represent the hexadecimal digits corresponding to decimal values 10 through 15.
- Each number system uses positional notation—each position in which a digit is written has a different positional value.
- A particularly important relationship that both the octal number system and the hexadecimal number system have to the binary system is that the bases of octal and hexadecimal (8 and 16 respectively) are powers of the base of the binary number system (base 2).
- To convert an octal number to a binary number, simply replace each octal digit with its three-digit binary equivalent.

Number System Conversions (Summer Work)

I. Convert the following Binary numbers to Octal, Decimal, and Hexadecimal

	Binary	Hexadecimal	Octal	Decimal
1	11010111			
2	10101011			
3	11010001			
4	01111110			
5	11001010			
6	00110111			
7	10010010			

II. Convert the following Decimal numbers to Binary, Octal, and Hexadecimal

	Decimal	Hexadecimal	Octal	Binary
1	500			
2	350			
3	64			
4	1732			
5	895			
6	45			
7	188			

III. Convert the following Octal numbers to Hexadecimal and Decimal

	Octal	Hexadecimal	Decimal
1	77		
2	256		
3	1024		
4	2555		
5	123		

Number System Conversions (Summer Work)

IV. Convert the following Hexadecimal numbers to Decimal

	Hexadecimal	Decimal
1	A12F	
2	B64E	
3	CAD	
4	8914	
5	1C2B	