AP Computer Science

CHAPTER 1

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.
- Understand 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
- Bit
- Byte
- Central processing unit (CPU)
- Hardware
- Information hiding
- Object-oriented programming
- Primary memory
- Secondary memory
- Software
- Software development life cycle (SDLC)
- System software
- Ubiquitous computing
- 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 standalone 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 Pentium personal computer purchased for about $1000 in 2006. Today, there are hundreds of millions of computers in the world,

most of which are PCs. There are also billions of computers embedded in everyday products such as handheld calculators, microwave ovens, cell phones, cars, refrigerators, and 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.

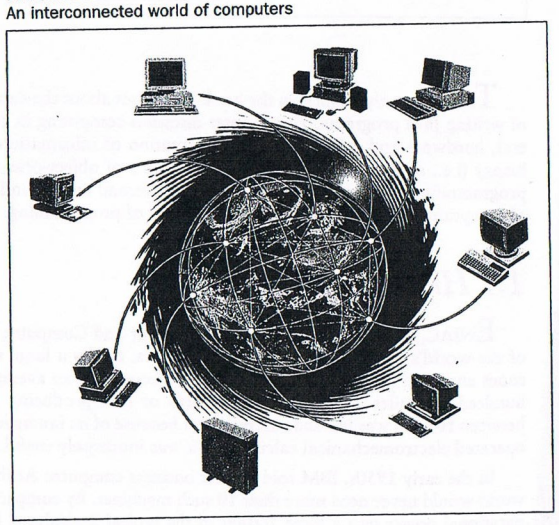
In the 1960s, time-sharing computers, costing hundreds of thousands and even millions of dollars, became popular at organizations large enough to afford them. These computers were powerful enough for 30 people to work on them simultaneously—and each felt as if he or she were the sole user. Each person 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, PCs appeared in great numbers, and soon after, local area networks of interconnected PCs became popular. These networks allowed a local group of PCs 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 21st 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). Many of these devices now connect to the Internet via wireless technology, giving users unprecedented mobility.

And the common language of many of these computers is Java.

1.2 Computer Hardware and Software

Computers can be viewed as 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 PC.

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.

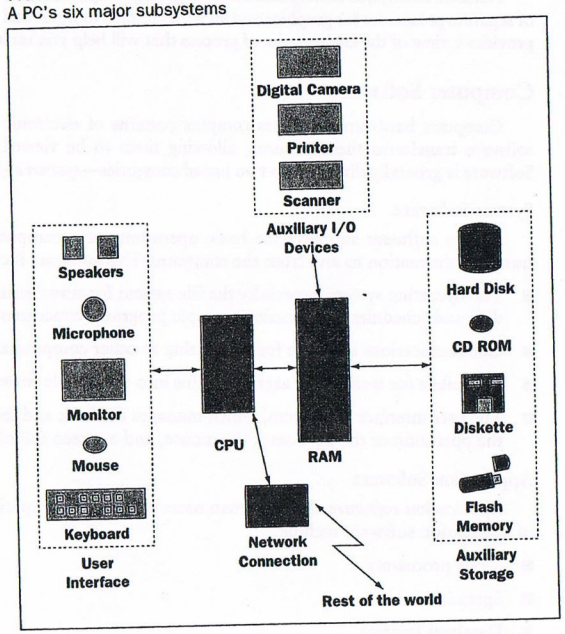
TABLE 1-1
Some commonly used quantities of information storage

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

Computer Hardware

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

FIGURE 1-2
A PC'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 PC. 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 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 2006, as these words are being written, *hard disks* typically store tens of billions of bytes of information, or gigabytes (GB) as they are commonly called. In addition to hard disks, which are permanently encased within computers, there are several *portable storage media*. Most computer software is now purchased on *CD-ROMs*. CD stands for compact disk and ROM stands for read-only memory. The term *ROM* is becoming somewhat misleading in this context as PCs are increasingly being equipped with CD devices that can read and write. Most CDs have a capacity of about 700 million bytes (megabytes or MB), enough for a little over an hour of music or a small PC software package. Currently, CDs are being supplanted by *DVDs*, which have about 10 times a CD's capacity. *Flash memory* sticks with a capacity of 100MB to 2GB are the most convenient portable storage devices. They support input and output and 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 PC, connecting it to all the resources of the Internet. For home computer users, a modem has long been the most widely used connection device. *Modem* stands for modulator–demodulator and refers to the fact that the device converts the digital information (0s and 1s) of the computer to an analog form suitable for transmission on phone lines and vice versa. Of course, as phone technology becomes increasingly digital, the term *modem* is fast becoming a misnomer. Other devices for connecting to the Internet include 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, PCs usually contain only about 512MB to 1GB 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 64MB or 128MB 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, but such is not the case. 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 CPU itself, and it is the programmer's job to determine how to translate a complex task into an enormous 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 1 billion cycles per second (1 GHz), and during each cycle, the CPU executes all or part of a basic operation.

Perhaps we have gone too far in downplaying the complexity of the CPU. To be fair, it too is highly complex—not in terms of the basic operations it performs, but rather in terms of how it achieves its incredible 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 18 months, 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 disk 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, it may 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 copy, decode, and execute 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 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 addiction 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, or expressed differently:

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

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:

$$1101101_{16}$$
$$1101101_{10}$$
$$1101101_8$$
$$1101101_2$$

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 ($0 \le x < 1$), 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 characters

- A...Z
- a...z
- 0...9
- +, -, *, /, etc.
- And 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

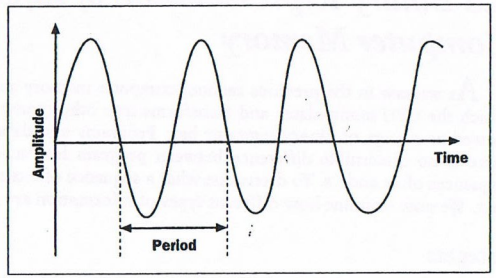
Java, however, uses a scheme called *Unicode* rather than ASCII. In this scheme, each character is represented by a pattern of 16 bits, ranging from 0000 0000 0000 0000 to 1111 1111 1111 1111. Unicode allows for 65,536 possibilities and can represent many alphabets simultaneously. Within Unicode, the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

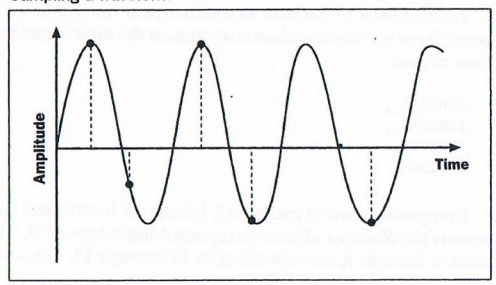
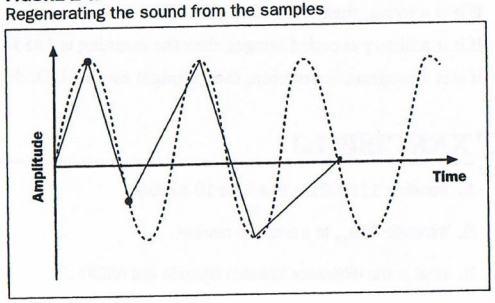


FIGURE 1-4b
Regenerating the sound from the samples



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

$$2\ \text{channels} * \frac{1\ \text{hour}}{\text{channel}} * \frac{60\ \text{minutes}}{\text{hour}} * \frac{60\ \text{seconds}}{\text{minute}} * \frac{44{,}000\ \text{samples}}{\text{second}} * \frac{16\ \text{bits}}{\text{sample}}$$

$$= 5{,}068{,}800{,}000\ \text{bits}$$
$$= 633{,}600{,}000\ \text{bytes}$$

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 are 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 of course 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 2.5MB 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 1.2MB/frame * 16 frames/second = 19MB/second of storage. A 2-hour feature film would need 432,000 seconds * 19MB/second = 8,208,000MB without the soundtrack! A typical DVD has space for several gigabytes of data, so our uncompressed video would obviously not fit on a DVD. Needless to say, very sophisticated data compression schemes, such as MPEG, have been developed that allow 3-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
- The second group of 8 bits represents the location (64_{10}) in memory of the first operand
- The third group of 8 bits represents the location (66_{10}) in memory of the second operand
- The fourth group of 8 bits represents the location (68_{10}) at which to store the sum

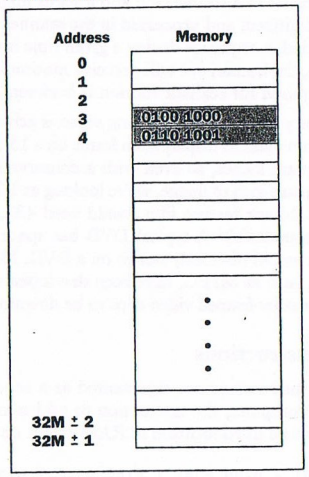
In other words, add the number at location 64 to the number at location 66 and store the sum at location 68.

Computer Memory

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*.

A group of contiguous bytes can represent a number, a string, a picture, a chunk of sound, a program instruction, or whatever, as determined by context. For example, let us consider the meaning of the two bytes starting at location 3 in Figure 1-5.

FIGURE 1-5
A 32MB RAM



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.

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 to 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. 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
```

meaning

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 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++, 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

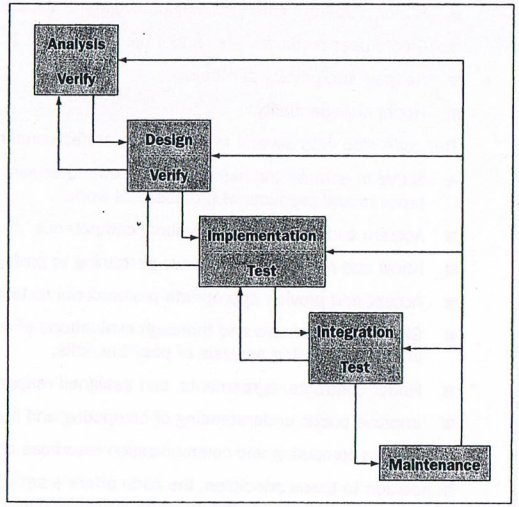
High-level programming languages help programmers write high-quality software in much the same sense as good tools help carpenters build high-quality houses, but there is much more to programming than writing lines of code, just as there is more to building houses than pounding nails. The "more" consists 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 particular 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, requirements change and minor or major modifications must be made.

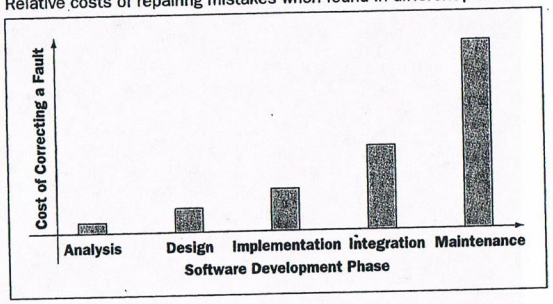
The interaction between the phases is shown in Figure 1-6. Note that the figure resembles a waterfall, in which the results of each phase flow down to the next. A mistake detected in one phase often requires the developer to back up and redo some of the work in the previous phase. Modifications made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

FIGURE 1-6
The waterfall model of the software development life cycle



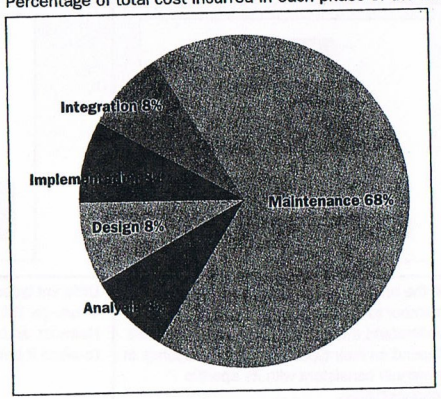
Programs rarely work as hoped the first time they are run; hence, they should be subjected to extensive and careful testing. Many people think that testing is an activity that applies only to the implementation and integration phases; however, the outputs of each phase should be scrutinized carefully. 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 software development process?
2. Which phase of the software development process incurs the highest cost to developers?
3. How does the waterfall model of software development work?
4. In which phase of the software development process 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 Stephen R. Schach, *Software Engineering with Java* (Chicago: Irwin, 1997).

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++, 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 everyone considers the object-oriented approach to be 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 in an attempt to associate something already familiar with something new. Like all analogies, this one is imperfect but ideally useful. Imagine that it is your task to plan an expedition in search of the lost treasure of Balbor. How familiar can

this be, you ask? Well, that depends on your taste in books, movies, and video games. 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 define the responsibilities of each member in terms of
   - A list of the resources used—these include the materials and knowledge needed by each member
   - The rules of behavior followed—these define how the team member behaves and responds in various situations

   Finally, you decide how many of each type will be needed.
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 this analogy relate to OOP? We give the answer in Table 1-4. On the left side of the table we describe various aspects of the expedition, and on the right side are listed 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
Expedition analogy 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 comprised 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
Expedition analogy to OOP

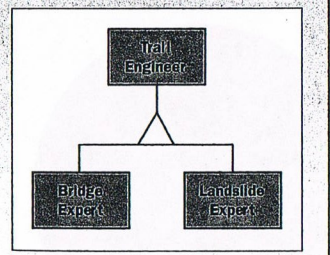
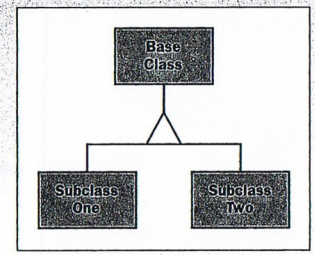
THE WORLD OF THE EXPEDITION	THE WORLD OF OOP
Team members working together accomplish the mission of the expedition. They do this by asking each other for services.	Objects working together accomplish the mission of the program. They do this by asking each other for services or, in the language of OOP, 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*.
	
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
Expedition analogy 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?
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.

Computer Ethics

COPYRIGHT, INTELLECTUAL PROPERTY, AND DIGITAL INFORMATION

For hundreds of years, copyright law has existed to regulate the use of intellectual property. At stake are the rights of authors and publishers to a return on their investment in works of the intellect, which include printed matter (books, articles, etc.), recorded music, film, and video. More recently, copyright law has been extended to include software and other forms of digital information. For example, copyright law protects the software used with this book. This prohibits the purchaser from reproducing the software for sale or free distribution to others. If the software is stolen or "pirated" in this way, the perpetrator can be prosecuted and punished by law. However, copyright law also allows for "fair use"—the purchaser may make backup copies of the software for personal use. When the purchaser sells the software to another user, the seller thereby relinquishes the right to use it, and the new purchaser acquires this right.

When governments design copyright legislation, governments try to balance the rights of authors and publishers to a return on their work against the rights of the public to fair use. In the case of printed matter and other works that have a physical embodiment, the meaning of fair use is usually clear. Without fair use, borrowing a book from a library or playing a CD at a high school dance would be unlawful.

With the rapid rise of digital information and its easy transmission on networks, different interest groups—authors, publishers, users, and computer professionals—are beginning to question the traditional balance of ownership rights and fair use. For example, is browsing a copyrighted manuscript on a network service an instance of fair use? Or does it involve a reproduction of the manuscript that violates the rights of the author or publisher? Is the manuscript a physical piece of intellectual property when browsed or just a temporary pattern of bits in a computer's memory? When you listen to an audio clip on a network are you violating copyright, or only when you download the clip to your hard drive? Users and technical experts tend to favor free access to any information placed on a network. Publishers and to a lesser extent authors tend to worry that their work, when placed on a network, will be resold for profit.

Legislators struggling with the adjustment of copyright law to a digital environment face many of these questions and concerns. Providers and users of digital information should also be aware of the issues. For more information about these topics, visit the Creative Commons Web site at http://creativecommons.org/.

SUMMARY

In this chapter, you learned:

- The modern computer age began in the late 1940s with the development of ENIAC. Business computing became practical in the 1950s, and time-sharing computers advanced computing in large organizations in the 1960s and 1970s. The 1980s saw the development and first widespread sales of personal computers, and the 1990s saw personal computers connected in networks.
- Modern computers consist of two primary components: hardware and software. Computer hardware is the physical component of the system. Computer software consists of programs that enable us to use the hardware.
- All information used by a computer is represented in binary form. This information includes numbers, text, images, sound, and program instructions.
- Programming languages have been developed in the course of three generations: generation 1 is machine language, generation 2 is assembly language, and generation 3 is high-level language.
- The software development process consists of several standard phases: customer request, analysis, design, implementation, integration, and maintenance.
- Object-oriented programming is a style of programming that can lead to better quality software. Breaking code into easily handled components simplifies the job of writing a large program.

VOCABULARY *Review*

Define the following terms:

Application software	Information hiding	Software development life cycle (SDLC)
Bit	Object-oriented programming	System software
Byte	Primary memory	Ubiquitous computing
Central processing unit (CPU)	Secondary memory	Waterfall model
Hardware	Software	

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.