1 Introduction

Chapter Objectives

- Describe the relationship between hardware and software.
- Define various types of software and how they are used.
- Identify the core hardware components of a computer and explain their roles.
- Explain how the hardware components interact to execute programs and manage data.
- Describe how computers are connected into networks to share information.
- Introduce the Java programming language.
- Describe the steps involved in program compilation and execution.
- Present an overview of object-oriented principles.

This book is about writing well-designed software. To understand software, we must first have a fundamental understanding of its role in a computer system. Hardware and software cooperate in a computer system to accomplish complex tasks. The purpose of various hardware components and the way those components are connected into networks are important prerequisites to the study of software development. This chapter first discusses basic computer processing and then begins our exploration of software development by introducing the Java *programming language and the principles of object-oriented programming.*

1.1 Computer Processing

All computer systems, whether it’s a desktop, laptop, tablet, smartphone, gaming console, or a special-purpose device such as a car’s navigation system, share certain characteristics. The details vary, but they all process data in similar ways. While the majority of this book deals with the development of software, we’ll begin with an overview of computer processing to set the context. It’s important to establish some fundamental terminology and see how key pieces of a computer system interact.

A computer system is made up of hardware and software. The **hardware** components of a computer system are the physical, tangible pieces that support the computing effort. They include chips, boxes, wires, keyboards, speakers, disks, memory cards, universal serial bus (USB) flash drives (also called jump drives), cables, plugs, printers, mice, monitors, routers, and so on. If you can physically touch it and it can be considered part of a computer system, then it is computer hardware.

Key Concept
A computer system consists of hardware and software that work in concert to help us solve problems.

The hardware components of a computer are essentially useless without instructions to tell them what to do. A **program** is a series of instructions that the hardware executes one after another. **Software** consists of programs and the data that programs use. Software is the intangible counterpart to the physical hardware components. Together they form a tool that we can use to help solve problems.

The key hardware components in a computer system are

- central processing unit (CPU)
- input/output (I/O) devices
- main memory
- secondary memory devices

Each of these hardware components is described in detail in the next section. For now, let's simply examine their basic roles. The ***central processing unit* (CPU)** is a device that executes the individual commands of a program. **Input/output (I/O) devices**, such as the keyboard, mouse, trackpad, and monitor, allow a human being to interact with the computer.

Programs and data are held in storage devices called memory, which fall into two categories: main memory and secondary memory. **Main memory** is the storage device that holds the software while it is being processed by the CPU. **Secondary memory** devices store software in a relatively permanent manner. The most important secondary memory device of a typical computer system is the hard disk that resides inside the main computer box. A USB flash drive is also an important secondary memory device. A typical USB flash drive cannot store nearly as much information as a hard disk. USB flash drives have the advantage of portability; they can be removed temporarily or moved from computer to computer as needed.

Figure 1.1 shows how information moves among the basic hardware components of a computer. Suppose you have an executable program you wish to run. The program is stored on some secondary memory device, such as a hard disk. When you instruct the computer to execute your program, a copy of the program is brought in from secondary memory and stored in main memory. The CPU reads the individual program instructions from main memory. The CPU then executes the instructions one at a time until the program ends. The data that the instructions use, such as two numbers that will be added together, also are stored in main memory. They are either brought in from secondary memory or read from an input device such as the keyboard. During execution, the program may display information to an output device such as a monitor.

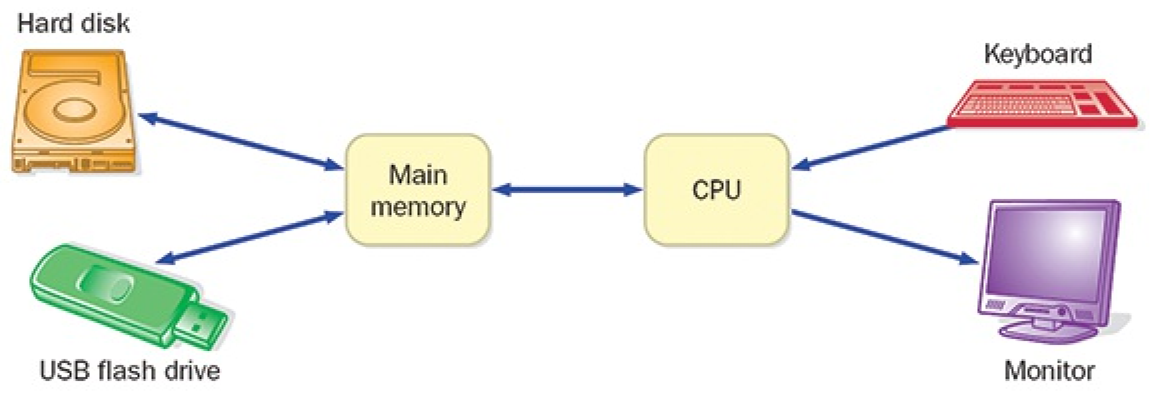


Figure 1.1 A simplified view of a computer system

Key Concept
The CPU reads the program instructions from main memory, executing them one at a time until the program ends.

The process of executing a program is fundamental to the operation of a computer. All computer systems basically work in the same way.

Software Categories

Software can be classified into many categories using various criteria. At this point, we will simply differentiate between system programs and application programs.

The **operating system** is the core software of a computer. It performs two important functions. First, it provides a **user interface** that allows the user to interact with the machine. Second, the operating system manages computer resources such as the CPU and main memory. It determines when programs are allowed to run, where they are loaded into memory, and how hardware devices communicate. It is the operating system's job to make the computer easy to use and to ensure that it runs efficiently.

Key Concept
The operating system provides a user interface and manages computer resources.

Several popular operating systems are in use today. The Windows operating system was developed for personal computers by Microsoft, which has captured the lion's share of the operating systems market. Various versions of the Unix operating system are also quite popular, especially in larger computer systems. A version of Unix called Linux was developed as an open-source project, which means that many people contributed to its development and its code is freely available. Because of that Linux has become a particular favorite among some users. Mac OS is an operating system used for computing systems developed by Apple Computers.

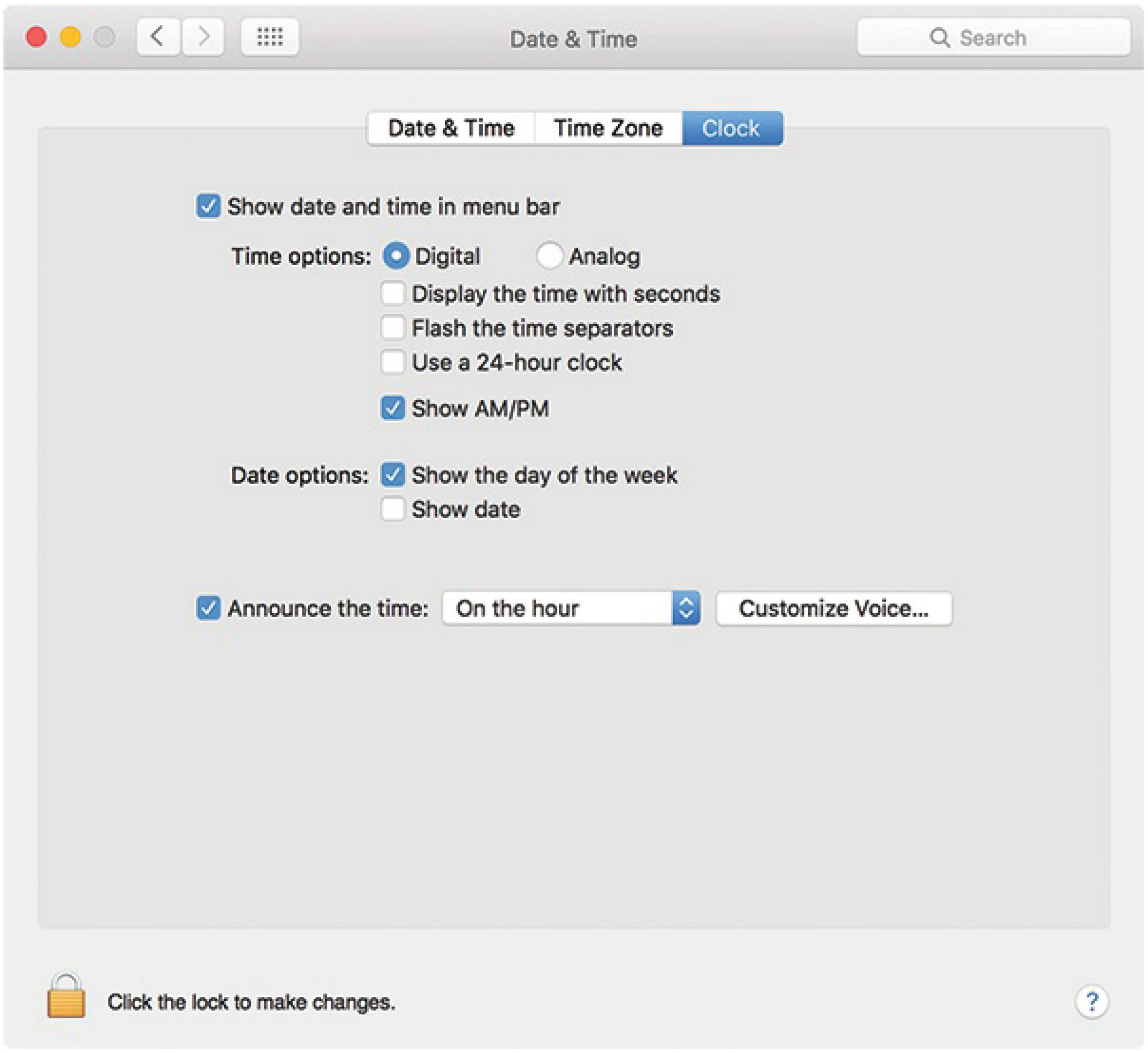
Operating systems are often specialized for mobile devices such as smartphones and tablets. The iOS operating system from Apple is used on the iPhone, iPad, and iPod. It is similar in functionality and appearance to the desktop Mac OS, but tailored for the smaller devices. The Android operating system developed by Google currently dominates the smartphone market.

An **application** (often shortened in conversation to "app") is a generic term for just about any software other than the operating system. Word processors, missile control systems, database managers, Web browsers, and games all can be considered application programs. Each application program has its own user interface that allows the user to interact with that particular program.

The user interface for most modern operating systems and applications is a **graphical user interface (GUI**, pronounced "gooey"), which, as the name implies, makes use of graphical screen elements. Among many others, these elements include

- **windows**, which are used to separate the screen into distinct work areas
- **icons**, which are small images that represent computer resources, such as a file
- **menus, checkboxes, and radio buttons**, which provide the user with selectable options
- **sliders**, which allow the user to select from a range of values
- **buttons**, which can be "pushed" with a mouse click to indicate a user selection

A mouse or trackpad is the primary input device used with GUIs; thus, GUIs are sometimes called *point-and-click interfaces*. The screenshot in **Figure 1.2** shows an example of a GUI.



✓ Show date and time in menu bar

Time options: ● Digital ○ Analog

☐ Display the time with seconds

☐ Flash the time separators

☐ Use a 24-hour clock

✓ Show AM/PM

Date options: ✓ Show the day of the week

☐ Show date

✓ Announce the time: On the hour

Click the lock to make changes.

Figure 1.2 An example of a graphical user interface (GUI)

Key Concept
As far as the user is concerned, the interface is the program.

The interface to an application or operating system is an important part of the software because it is the only part of the program with which the user interacts directly. To the user, the interface *is* the program. Throughout this book, we discuss the design and implementation of graphical user interfaces.

The focus of this book is the development of high-quality application programs. We explore how to design and write software that will perform calculations, make decisions, and present results textually or graphically. We use the Java programming language throughout the text to demonstrate various computing concepts.

Digital Computers

Two fundamental techniques are used to store and manage information: analog and digital. **Analog** information is continuous, in direct proportion to the source of the information. For example, an alcohol thermometer is an analog device for measuring temperature. The alcohol rises in a tube in direct proportion to the temperature outside the tube. Another example of analog information is an electronic signal used to represent the vibrations of a sound wave. The signal's voltage varies in direct proportion to the original sound wave. A stereo amplifier sends this kind of electronic signal to its speakers, which vibrate to reproduce the sound. We use the term analog because the signal is directly analogous to the information it represents. **Figure 1.3** graphically depicts a sound wave captured by a microphone and represented as an electronic signal.

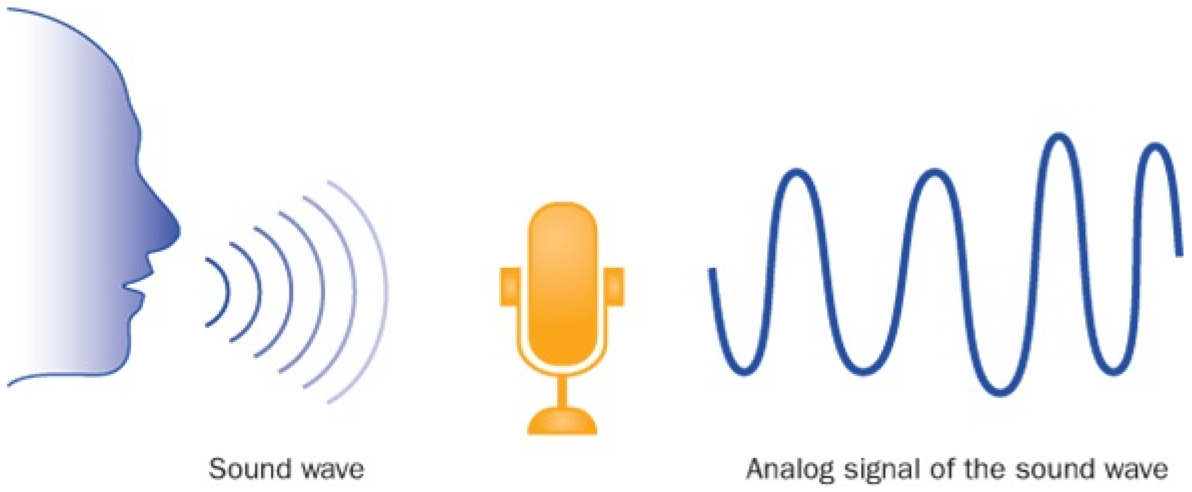


Figure 1.3 A sound wave and an electronic analog signal that represents the wave

Key Concept
Digital computers store information by breaking it into pieces and representing each piece as a number.

Digital technology breaks information into discrete pieces and represents those pieces as numbers. The music on a compact disc is stored digitally, as a series of numbers. Each number represents the voltage level of one specific instance of the recording. Many of these measurements are taken in a short period of time, perhaps 44,000 measurements every second. The number of measurements per second is called the *sampling rate*. If samples are taken often enough, the discrete voltage measurements can be used to generate a continuous analog signal that is “close enough” to the original. In most cases, the goal is to create a reproduction of the original signal that is good enough to satisfy the human senses.

Figure 1.4 shows the sampling of an analog signal. When analog information is converted to a digital format by breaking it into pieces, we say it has been *digitized*. Because the changes that occur in a signal between samples are lost, the sampling rate must be sufficiently fast.

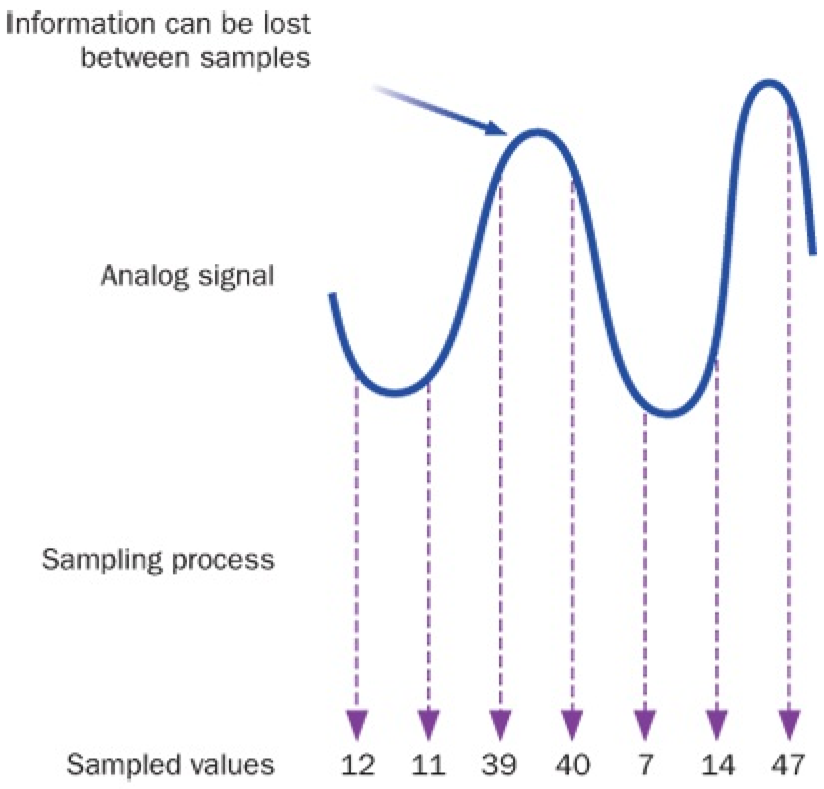


Figure 1.4 Digitizing an analog signal by sampling

Sampling is only one way to digitize information. For example, a sentence of text is stored on a computer as a series of numbers, where each number represents a single character in the sentence. Every letter, digit, and punctuation symbol has been assigned a number. Even the space character is assigned a number. Consider the following sentence:

Hi, Heather.

The characters of the sentence are represented as a series of 12 numbers, as shown in **Figure 1.5**. When a character is repeated, such as the uppercase `'H'`, the same representation number is used. Note that the uppercase version of a letter is stored as a different number from the lowercase version, such as the `'H'` and `'h'` in the word Heather. They are considered distinct characters.

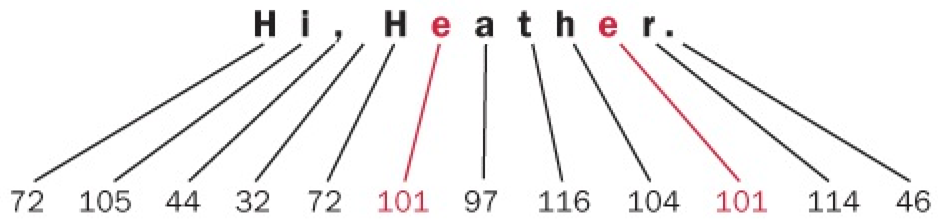


Figure 1.5 Text is stored by mapping each character to a number

Modern electronic computers are digital. Every kind of information, including text, images, numbers, audio, video, and even program instructions, is broken into pieces. Each piece is represented as a number. The information is stored by storing those numbers.

Binary Numbers

A digital computer stores information as numbers, but those numbers are not stored as **decimal** values. All information in a computer is stored and managed as **binary** values. Unlike the decimal system, which has 10 digits (0 through 9), the binary number system has only two digits (0 and 1). A single binary digit is called a *bit*.

All number systems work according to the same rules. The *base value* of a number system dictates how many digits we have to work with and indicates the place value of each digit in a number. The decimal number system is base 10, whereas the binary number system is base 2. **Appendix B** contains a detailed discussion of number systems.

Key Concept
Binary is used to store and move information in a computer because the devices that store and manipulate binary data are inexpensive and reliable.

Modern computers use binary numbers because the devices that store and move information are less expensive and more reliable if they have to represent only one of two possible values. Other than this characteristic, there is nothing special about the binary number system. Computers have been created that use other number systems to store and move information, but they aren’t as convenient.

Some computer memory devices, such as hard drives, are magnetic in nature. Magnetic material can be polarized easily to one extreme or the other, but intermediate levels are difficult to distinguish. Therefore, magnetic devices can be used to represent binary values quite effectively—a magnetized area represents a binary 1 and a demagnetized area represents a binary 0. Other computer memory devices are made up of tiny electrical circuits. These devices are easier to create and are less likely to fail if they have to switch between only two states. We’re better off reproducing millions of these simple devices than creating fewer, more complicated ones.

Binary values and digital electronic signals go hand in hand. They improve our ability to transmit information reliably along a wire. As we’ve seen, an analog signal has continuously varying voltage with infinitely many states, but a digital signal is *discrete*, which means the voltage changes dramatically between one extreme (such as +5 volts) and the other (such as –5 volts). At any point, the voltage of a digital signal is considered to be either “high,” which represents a binary 1, or “low,” which represents a binary 0. **Figure 1.6** compares these two types of signals.

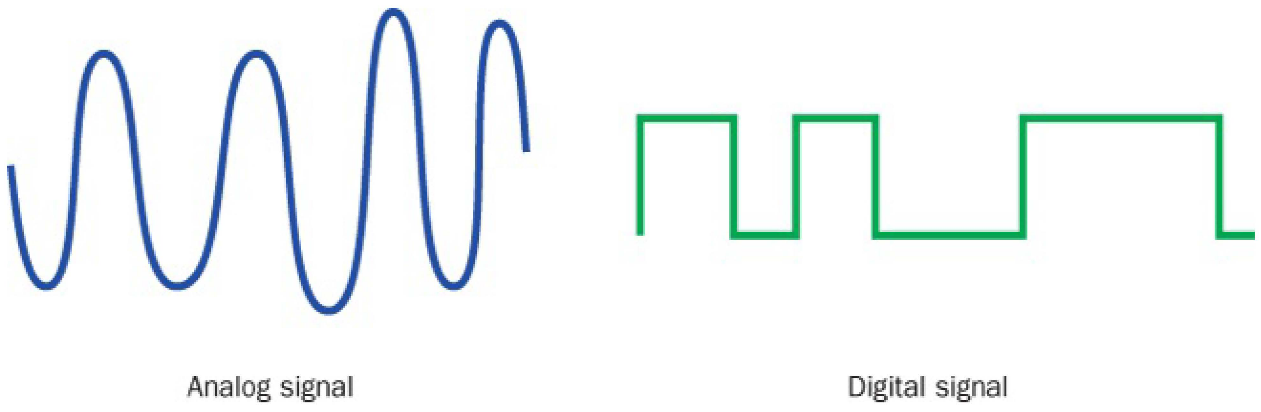


Figure 1.6 An analog signal vs. a digital signal

As a signal moves down a wire, it gets weaker and degrades due to environmental conditions. That is, the voltage levels of the original signal change slightly. The trouble with an analog signal is that as it fluctuates, it loses its original information. Since the information is directly analogous to the signal, any change in the signal changes the information. The changes in an analog signal cannot be recovered because the degraded signal is just as valid as the original. A digital signal degrades just as an analog signal does, but because the digital signal is originally at one of two extremes, it can be reinforced before any information is lost. The voltage may change slightly from its original value, but it still can be interpreted correctly as either high or low.

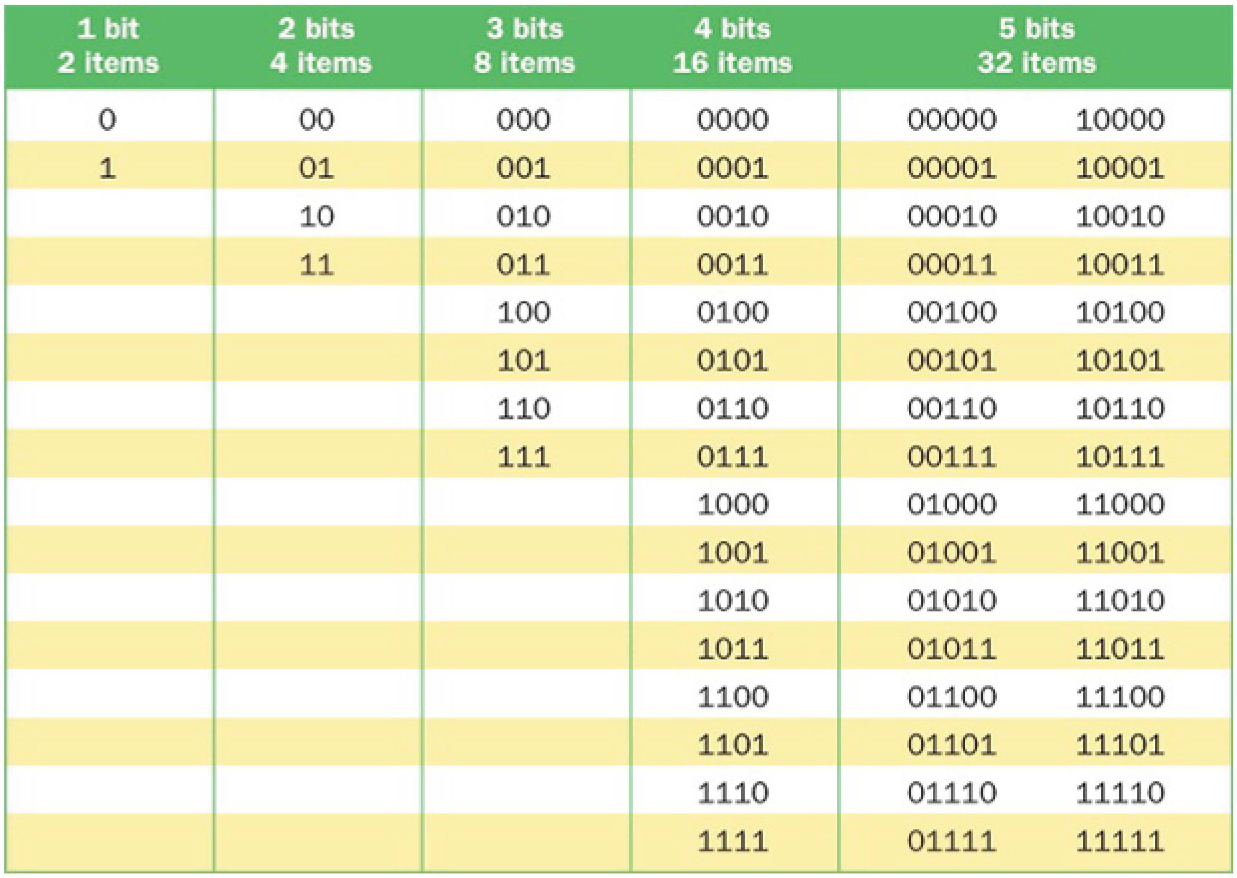
The number of bits we use in any given situation determines the number of unique items we can represent. A single bit has two possible values, 0 and 1, and therefore can represent two possible items or situations. If we want to represent the state of a light bulb (off or on), one bit will suffice, because we can interpret 0 as the light bulb being off and 1 as the light bulb being on. If we want to represent more than two things, we need more than one bit.

Two bits, taken together, can represent four possible items because there are exactly four permutations of two bits: 00, 01, 10, and 11. Suppose we want to represent the gear that a car is in (park, drive, reverse, or neutral). We would need only two bits, and could set up a mapping between the bit permutations and the gears. For instance, we could say that 00 represents park, 01 represents drive, 10 represents reverse, and 11 represents neutral. In this case, it wouldn't matter if we switched that mapping around, though in some cases the relationships between the bit permutations and what they represent are important.

Three bits can represent eight unique items, because there are eight permutations of three bits. Similarly, four bits can represent 16 items, five bits can represent 32 items, and so on. **Figure 1.7** shows the relationship between the number of bits used and the number of items they can represent. In general, N bits can represent 2^N unique items. For every bit added, the number of items that can be represented doubles.

Key Concept

There are exactly 2^N permutations of N bits. Therefore, N bits can represent up to 2^N unique items.



1 bit 2 items	2 bits 4 items	3 bits 8 items	4 bits 16 items	5 bits 32 items	
0	00	000	0000	00000	10000
1	01	001	0001	00001	10001
	10	010	0010	00010	10010
	11	011	0011	00011	10011
		100	0100	00100	10100
		101	0101	00101	10101
		110	0110	00110	10110
		111	0111	00111	10111
			1000	01000	11000
			1001	01001	11001
			1010	01010	11010
			1011	01011	11011
			1100	01100	11100
			1101	01101	11101
			1110	01110	11110
			1111	01111	11111

Figure 1.7 The number of bits used determines the number of items that can be represented

We've seen how a sentence of text is stored on a computer by mapping characters to numeric values. Those numeric values are stored as binary numbers. Suppose we want to represent character strings in a language that contains 256 characters and symbols. We would need to use eight bits to store each character because there are 256 unique permutations of eight bits (2^8 equals 256). Each bit permutation, or binary value, is mapped to a specific character.

How many bits would be needed to represent 195 countries of the world? Seven wouldn't be enough, because 2^7 equals 128. Eight bits would be enough, but some of the 256 permutations would not be mapped to a country.

Ultimately, representing information on a computer boils down to the number of items there are to represent and determining the way those items are mapped to binary values.

Self-Review Questions

(see answers in **Appendix L**)

SR 1.1 What is hardware? What is software?
SR 1.2 What are the two primary functions of an operating system?
SR 1.3 The music on a CD is created using a sampling rate of 44,000 measurements per second. Each measurement is stored as a number that represents a specific voltage level. How many such numbers are used to store a three-minute long song? How many such numbers does it take to represent one hour of music?
SR 1.4 What happens to information when it is stored digitally?
SR 1.5 How many unique items can be represented with the following?

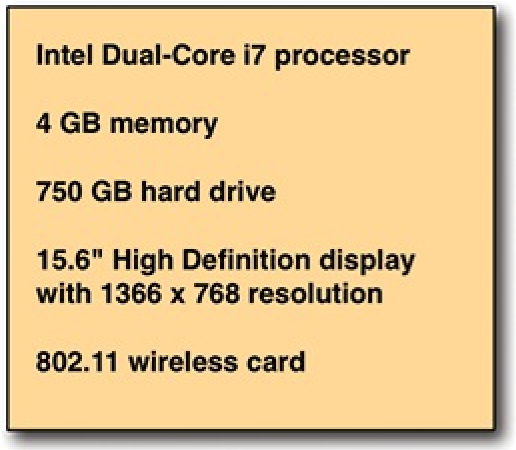
a. 2 bits
b. 4 bits
c. 5 bits

d. 7 bits

SR 1.6 Suppose you want to represent each of the 50 states of the United States using a unique permutation of bits. How many bits would be needed to store each state representation? Why?

1.2 Hardware Components

Let’s examine the hardware components of a computer system in more detail. Consider the desktop computer described in **Figure 1.8**. What does it all mean? Is the system capable of running the software you want it to? How does it compare with other systems? These terms are explained throughout this section.


Intel Dual-Core i7 processor

4 GB memory

750 GB hard drive

15.6" High Definition display with 1366 x 768 resolution

802.11 wireless card

Figure 1.8 The hardware specification of a particular computer

Computer Architecture

The architecture of a house defines its structure. Similarly, we use the term **computer architecture** to describe how the hardware components of a computer are put together. **Figure 1.9** illustrates the basic architecture of a generic computer system. Information travels between components across a group of wires called a **bus**.

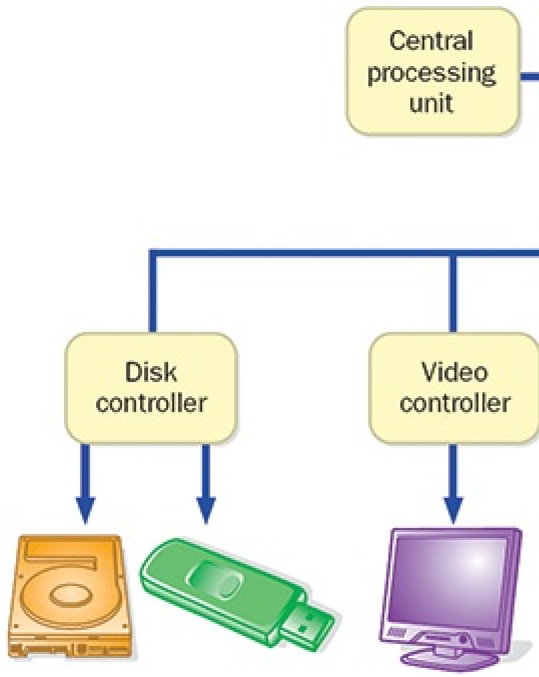


Figure 1.9 Basic computer architecture

Key Concept
The core of a computer is made up of main memory, which stores programs and data, and the CPU, which executes program instructions one at a time.

The CPU and the main memory make up the core of a computer. As we mentioned earlier, main memory stores programs and data that are in active use, and the CPU methodically executes program instructions one at a time. The CPU in the computer described in **Figure 1.8** is manufactured by the Intel Corporation, which supplies processors for many computer systems.

Suppose we have a program that computes the average of a list of numbers. The program and the numbers must reside in main memory while the program runs. The CPU reads one program instruction from main memory and executes it. If an instruction needs data, such as a number in the list, to perform its task, the CPU reads that information as well. This process repeats until the program ends. The average, when computed, is stored in main memory to await further processing or long-term storage in secondary memory.

Almost all devices in a computer system other than the CPU and main memory are called **peripherals**; they operate at the periphery, or outer edges, of the system (although they may be in the same box). Users don't interact directly with the CPU or main memory. Although they form the essence of the machine, the CPU and main memory would not be useful without peripheral devices.

Controllers are devices that coordinate the activities of specific peripherals. Every device has its own particular way of formatting and communicating data, and part of the controller's role is to handle these idiosyncrasies and isolate them from the rest of the computer hardware. Furthermore, the controller often handles much of the actual transmission of information, allowing the CPU to focus on other activities.

Input/output (I/O) devices and secondary memory devices are considered peripherals. Another category of peripherals consists of **data transfer devices**, which allow information to be sent and received between computers. The computer specified in **Figure 1.8** includes a network card that uses the 802.11 standard for wireless radio communication (or WiFi) to a computer network.

In some ways, secondary memory devices and data transfer devices can be thought of as I/O devices because they represent a source of information (input) and a place to send information (output). For our discussion, however, we define I/O devices as those devices that allow the user to interact with the computer.

Input/Output Devices

Let’s examine some I/O devices in more detail. The most common input devices are the keyboard and the mouse or trackpad. Others include

- **bar code readers**, such as the one used at a retail store checkout
- **microphones**, used by voice recognition systems that interpret voice commands
- **virtual reality devices**, such as handheld devices that interpret the movement of the user’s hand

- **scanners**, which convert text, photographs, and graphics into machine-readable form
- **cameras**, which capture still pictures and video, and can also be used to process special input such as QR codes

Monitors and printers are the most common output devices. Others include

- **plotters**, which move pens across large sheets of paper (or vice versa)
- **speakers**, for audio output
- **goggles**, for virtual reality display

Some devices can provide both input and output capabilities. A *touch screen* system can detect the user touching the screen at a particular place. Software can then use the screen to display text and graphics in response to the user's touch. Touch screens have become commonplace for handheld devices.

The computer described in **Figure 1.8** includes a monitor with a 15.6-inch display area (measured diagonally). A picture is represented in a computer by breaking it up into separate picture elements, or *pixels*. This monitor can display a grid of 1366 by 768 pixels.

Main Memory and Secondary Memory

Main memory is made up of a series of small, consecutive **memory locations**, as shown in **Figure 1.10**. Associated with each memory location is a unique number called an **address**.

Key Concept
An address is a unique number associated with a memory location.

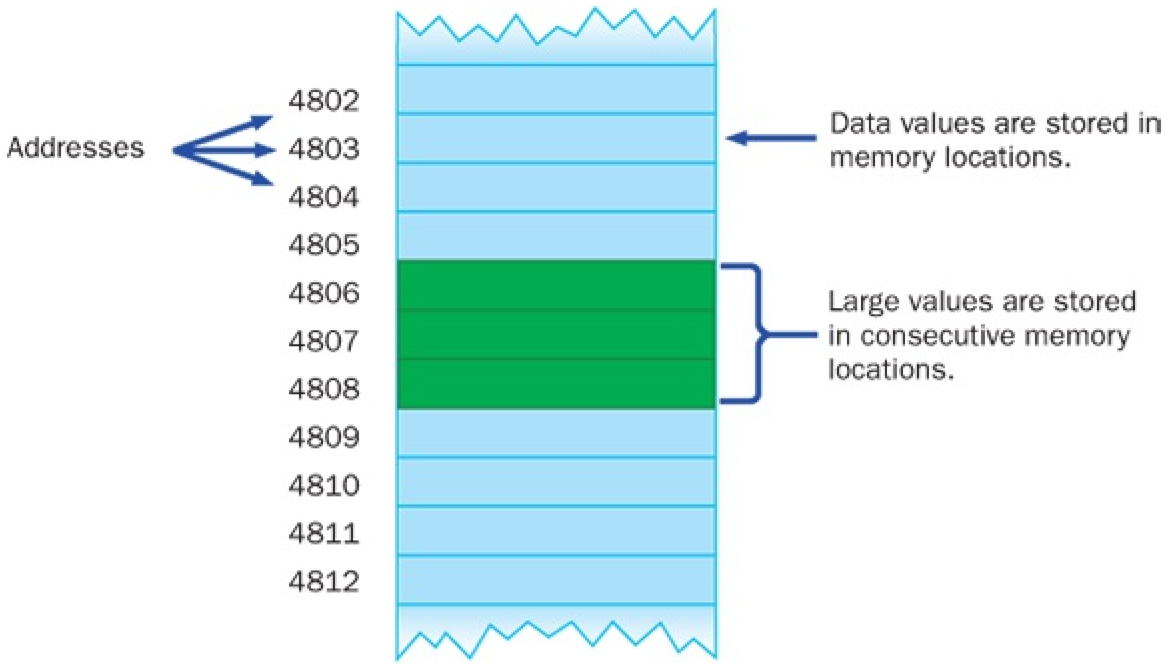
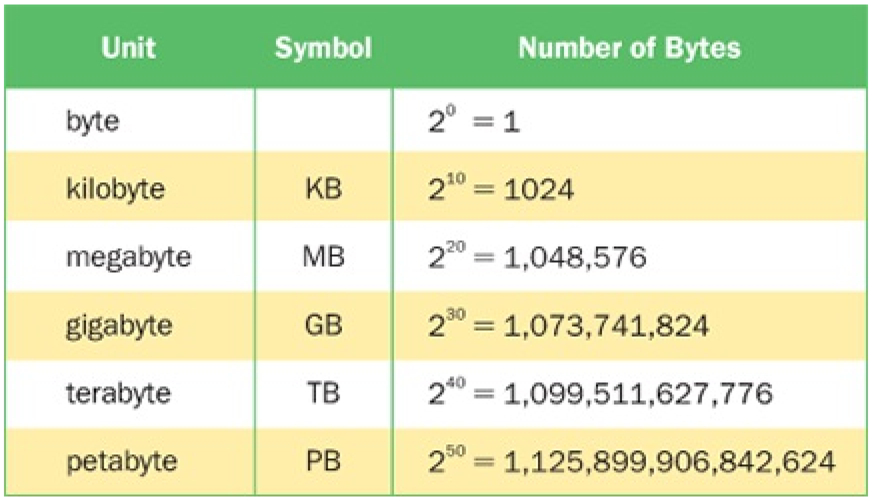

Addresses
4802
4803
4804
4805
4806
4807
4808
4809
4810
4811
4812
Data values are stored in memory locations.
Large values are stored in consecutive memory locations.

Figure 1.10 Memory locations

When data is stored in a memory location, it overwrites and destroys any information that was previously stored at that location. However, the process of reading data from a memory location does not affect it.

On many computers, each memory location consists of eight bits, or one *byte*, of information. If we need to store a value that cannot be represented in a single byte, such as a large number, then multiple, consecutive bytes are used to store the data.

The **storage capacity** of a device such as main memory is the total number of bytes it can hold. Devices can store thousands or millions of bytes, so you should become familiar with larger units of measure. Because computer memory is based on the binary number system, all units of storage are powers of two. A **kilobyte** (KB) is 1024, or 2^{10}, bytes. Some larger units of storage are a *megabyte* (MB), a ***gigabyte*** (GB), a **terabyte** (TB), and a *petabyte* (PB) as listed in **Figure 1.11**. It's usually easier to think about these capacities by rounding them off. For example, most computer users think of a kilobyte as approximately one thousand bytes, a megabyte as approximately one million bytes, and so forth.



Unit	Symbol	Number of Bytes
byte		$2^{0} = 1$
kilobyte	KB	$2^{10} = 1024$
megabyte	MB	$2^{20} = 1,048,576$
gigabyte	GB	$2^{30} = 1,073,741,824$
terabyte	TB	$2^{40} = 1,099,511,627,776$
petabyte	PB	$2^{50} = 1,125,899,906,842,624$

Figure 1.11 Units of binary storage

Key Concept
Main memory is volatile, meaning the stored information is maintained only as long as electric power is supplied.

Many personal computers have 4 GB of main memory, such as the system described in **Figure 1.8**. A large main memory allows large programs or multiple programs to run efficiently, because they don’t have to retrieve information from secondary memory as often.

Main memory is usually *volatile*, meaning that the information stored in it will be lost if its electric power supply is turned off. When you are working on a computer, you should often save your work onto a secondary memory device such as a USB flash drive in case the power goes out. Secondary memory devices are usually *nonvolatile*; the information is retained even if the power supply is turned off.

The *cache* is used by the CPU to reduce the average access time to instructions and data. The cache is a small, fast memory that stores the contents of the most frequently used main memory locations. Contemporary CPUs include an instruction cache to speed up the fetching of executable instructions and a data cache to speed up the fetching and storing of data.

The most common secondary storage devices are *hard disks* and USB *flash drives*. A typical USB flash drive stores between 1 and 256 GB of information. The storage capacities of hard drives vary, but on personal computers, capacities typically range between 500 and 750 GB, such as in the system described in **Figure 1.8**. Some hard disks can store much more.

A USB flash drive consists of a small printed circuit board carrying the circuit elements and a USB connector, insulated electrically and protected inside a plastic, metal, or rubberized case, which can be carried in a pocket or on a key chain, for example.

A disk is a magnetic medium on which bits are represented as magnetized particles. A read/write head passes over the spinning disk, reading or writing information as appropriate. A hard disk drive might actually contain several disks in a vertical column with several read/write heads, such as the one shown in **Figure 1.12**.

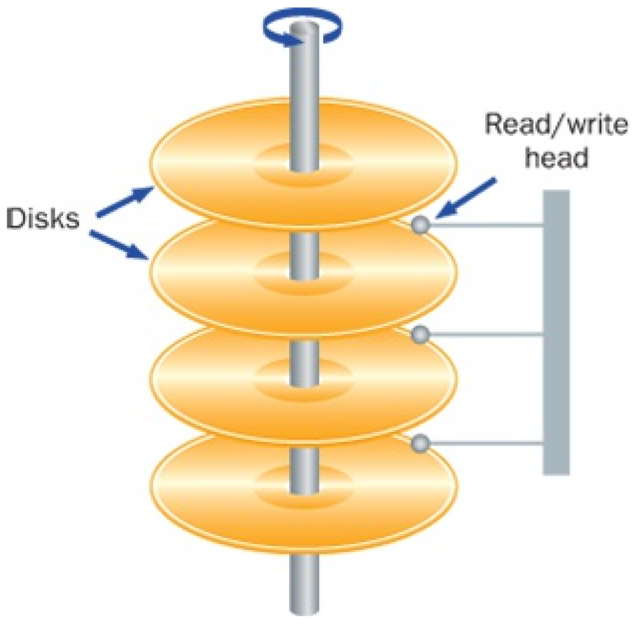


Figure 1.12 A hard disk drive with multiple disks and read/write heads

Before disk drives were common, magnetic tapes were used as secondary storage. Tapes are considerably slower than hard disk and USB flash drives because of the way information is accessed. A hard disk is a *direct access device* since the read/write head can move, in general, directly to the information needed. A USB flash drive is also a direct access device, but nothing moves mechanically. The terms *direct access* and *random access* are often used interchangeably. However, information on a tape can be accessed only after first getting past the intervening data. A tape must be rewound or fast-forwarded to get to the appropriate position. A tape is therefore considered a *sequential access device*. For these reasons, tapes are no longer used as a computing storage device, just as audio cassettes were supplanted by compact discs.

Two other terms are used to describe memory devices: **random access memory (RAM)** and **read-only memory (ROM)**. It's important to understand these terms because they are used often and their names can be misleading. The terms RAM and main memory are basically interchangeable, describing the memory where active programs and data are stored. **ROM** can refer to chips on the computer motherboard or to portable storage such as a compact disc. ROM chips typically store software called BIOS (basic input/output system) that provide the preliminary instructions needed when the computer is turned on initially. After information is stored on ROM, generally it is not altered (as the term *read-only* implies) during typical computer use. Both RAM and ROM are direct (or random) access devices.

Key Concept
The surface of a CD has both smooth areas and small pits. A pit represents a binary 1 and a smooth area represents a binary 0.

A **CD-ROM** is a portable secondary memory device. CD stands for compact disc. It is called ROM because information is stored permanently when the CD is created and cannot be changed. Like its musical CD counterpart, a CD-ROM stores information in binary format. When the CD is initially created, a microscopic pit is pressed into the disc to represent a binary 1, and the disc is left smooth to represent a binary 0. The bits are read by shining a low-intensity laser beam onto the spinning disc. The laser beam reflects strongly from a smooth area on the disc but weakly from a pitted area. A sensor receiving the reflection determines whether each bit is a 1 or a 0 accordingly. A typical CD-ROM's storage capacity ranges between 650 and 900 MB.

Variations on basic CD technology emerged quickly. A CD-Recordable (CD-R) disk can be used to create a CD for music or for general computer storage. Once created, you can use a CD-R disc in a standard CD player, but you can’t change the information on a CD-R disc once it has been “burned.” Music CDs that you buy are pressed from a mold, whereas CD-Rs are burned with a laser.

CDs were initially a popular format for music; they later evolved to be used as a general computer storage device. Similarly, the *DVD* format was originally created for video and is now making headway as a general format for computer data. DVD once stood for digital video disc or digital versatile disc, but now the acronym generally stands on its own. A DVD has a tighter format (more bits per square inch) than a CD and can therefore store more information.

The use of CDs and DVDs as secondary storage devices for computers has declined greatly as advances in networks and external “cloud” storage provided better options.

The capacity of storage devices changes continually as technology improves. A general rule in the computer industry suggests that storage capacity approximately doubles every 18 months. However, this progress eventually will slow down as capacities approach absolute physical limits.

The Central Processing Unit

The CPU interacts with main memory to perform all fundamental processing in a computer. The CPU interprets and executes instructions, one after another, in a continuous cycle. It is made up of three important components, as shown in **Figure 1.13**. The *control unit* coordinates the processing steps, the *registers* provide a small amount of storage space in the CPU itself, and the *arithmetic/logic unit* performs calculations and makes decisions. The registers are the smallest, fastest cache in the system.

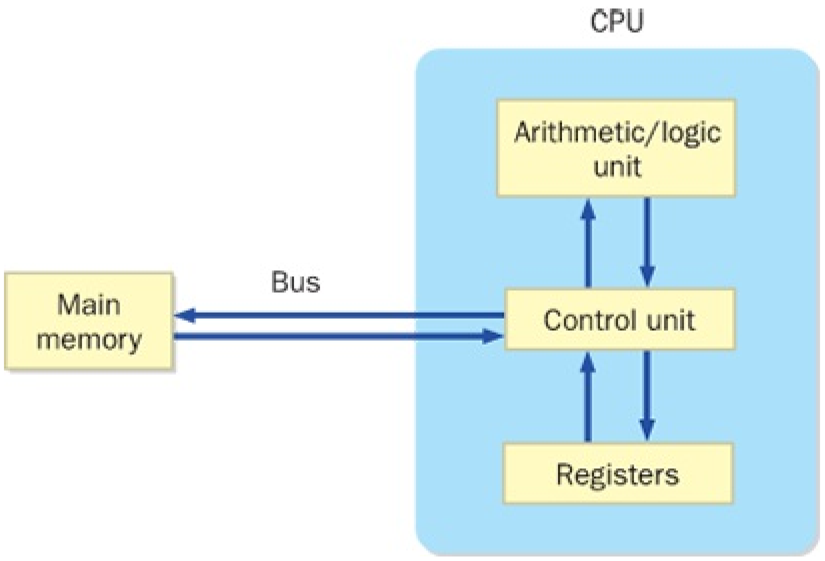


Figure 1.13 CPU components and main memory

The control unit coordinates the transfer of data and instructions between main memory and the registers in the CPU. It also coordinates the execution of the circuitry in the arithmetic/logic unit to perform operations on data stored in particular registers.

In most CPUs, some registers are reserved for special purposes. For example, the *instruction register* holds the current instruction being executed. The *program counter* is a register that holds the address of the next instruction to be executed. In addition to these and other special-purpose registers, the CPU also contains a set of general-purpose registers that are used for temporary storage of values as needed.

The concept of storing both program instructions and data together in main memory is the underlying principle of the *von Neumann architecture* of computer design, named after John von Neumann, a Hungarian-American mathematician who first advanced this programming concept in 1945. These computers continually follow the *fetch–decode–execute* cycle depicted in **Figure 1.14**. An instruction is fetched from main memory at the address stored in the program counter and is put into the instruction register. The program counter is incremented at this point to prepare for the next cycle. Then, the instruction is decoded electronically to determine which operation to carry out. Finally, the control unit activates the correct circuitry to carry out the instruction, which may load a data value into a register or add two values together, for example.

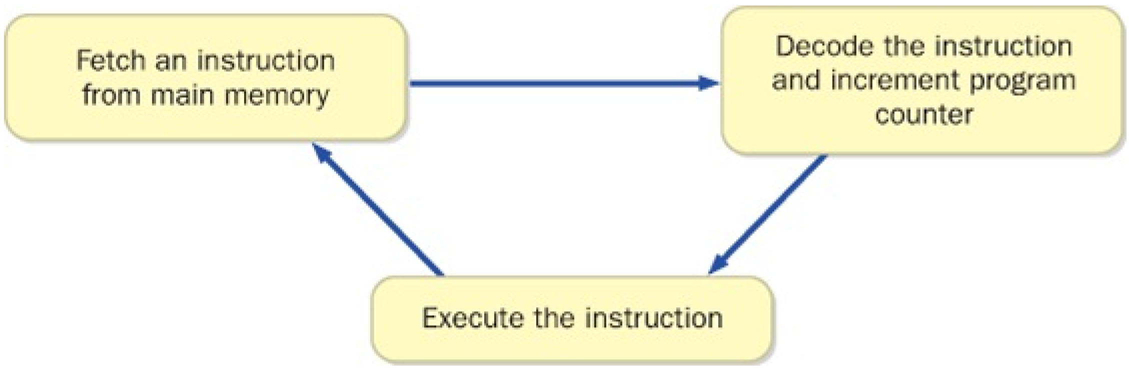

Fetch an instruction from main memory
Decode the instruction and increment program counter
Execute the instruction

Figure 1.14 The continuous fetch–decode–execute cycle

Key Concept

The fetch-decode-execute cycle forms the foundation of computer processing.

The CPU is constructed on a chip called a *microprocessor*, a device that is part of the main circuit board of the computer. This board also contains ROM chips and communication sockets to which device controllers, such as the controller that manages the video display, can be connected.

Another crucial component of the main circuit board is the *system clock*. The clock generates an electronic pulse at regular intervals, which synchronizes the events of the CPU. The rate at which the pulses occur is called the *clock speed*, and it varies depending on the processor. The computer described in **Figure 1.8** includes an Intel Dual Core i7 processor that runs at a clock speed of 3.1 GHz, or approximately 3.1 billion pulses per second. The speed of the system clock provides a rough measure of how fast the CPU executes instructions.

A processor described as duel-core actually has two processors built onto one chip, and can do two things at once if the program it is executing is designed for that. However, it is difficult to write programs that can take advantage of that second processor. This will become increasingly more important for software developers, because the future of processor design is likely to be more cores, not single cores that run at faster speeds.

Self-Review Questions

(see answers in **Appendix L**)

SR 1.7 How many bytes are there in each of the following?

a. 3 KB
b. 2 MB
c. 4 GB

SR 1.8 How many bits are there in each of the following?

a. 8 bytes
b. 2 KB
c. 4 MB

SR 1.9 The music on a CD is created using a sampling rate of 44,000 measurements per second. Each measurement is stored as a number that represents a specific voltage level. Suppose each of these numbers requires two bytes of storage space. How many MB does it take to represent one hour of music?
SR 1.10 What are the two primary hardware components in a computer? How do they interact?
SR 1.11 What is a memory address?
SR 1.12 What does volatile mean? Which memory devices are volatile and which are nonvolatile?
SR 1.13 Select the word from the following list that best matches each of the following phrases:

controller, CPU, main, network card, peripheral, RAM, register, ROM, secondary

a. Almost all devices in a computer system, other than the CPU and the main memory, are categorized as this.
b. A device that coordinates the activities of a peripheral device.
c. Allows information to be sent and received.
d. This type of memory is usually volatile.
e. This type of memory is usually nonvolatile.
f. This term basically is interchangeable with the term “main memory.”
g. Where the fundamental processing of a computer takes place.

1.3 Networks

A **network** consists of two or more computers connected together so they can exchange information. Using networks has become the normal mode of commercial computer operation. New technologies are emerging every day to capitalize on the connected environments of modern computer systems.

Key Concept
A network consists of two or more computers connected together so that they can exchange information.

Figure 1.15 shows a simple computer network. One of the devices on the network is a printer, which allows any computer connected to the network to print a document on that printer. One of the computers on the network is designated as a *file server*, which is dedicated to storing programs and data that are needed by many network users. A file server usually has a large amount of secondary memory. When a network has a file server, each individual computer doesn’t need its own copy of a program.

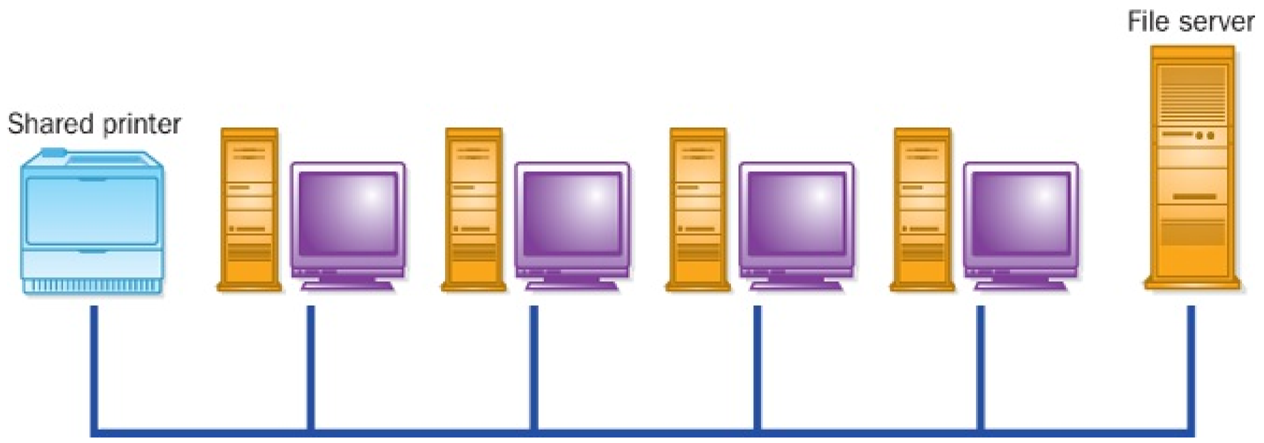


Figure 1.15 A simple computer network

Network Connections

If two computers are directly connected, they can communicate in basically the same way that information moves across wires inside a single machine. When connecting two geographically close computers, this solution works well and is called a **point-to-point connection**. However, consider the task of connecting many computers together across large distances. If point-to-point connections are used, every computer is directly connected by a wire to every other computer in the network. A separate wire for each connection is not a workable solution because every time a new computer is added to the network, a new communication line will have to be installed for each computer already in the network. Furthermore, a single computer can handle only a small number of direct connections.

Figure 1.16 shows multiple point-to-point connections. Consider the number of communication lines that would be needed if two or three additional computers were added to the network. These days, local networks, such as those within a single building, often use wireless connections, minimizing the need for running wires.

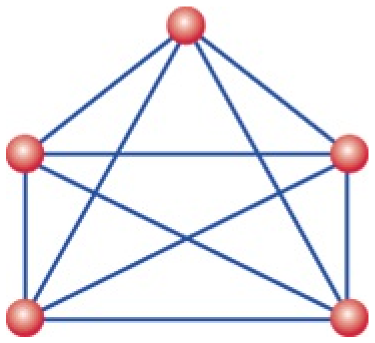


Figure 1.16 Point-to-point connections

Compare the diagrams in Figures 1.15 and 1.16. All of the computers shown in Figure 1.15 share a single communication line. Each computer on the network has its own *network address*, which uniquely identifies it. These addresses are similar in concept to the addresses in main memory except that they identify individual computers on a network instead of individual memory locations inside a single computer. A message is sent across the line from one computer to another by specifying the network address of the computer for which it is intended.

Sharing a communication line is cost effective and makes adding new computers to the network relatively easy. However, a shared line introduces delays. The computers on the network cannot use the communication line at the same time. They have to take turns sending information, which means they have to wait when the line is busy.

Key Concept
Sharing a communication line creates delays, but it is cost effective and simplifies adding new computers to the network.

One technique to improve network delays is to divide large messages into segments, called *packets*, and then send the individual packets across the network intermixed with pieces of other messages sent by other users. The packets are collected at the destination and reassembled into the original message. This situation is similar to a group of people using a conveyor belt to move a set of boxes from one place to another. If only one person were allowed to use the conveyor belt at a time, and that person had a large number of boxes to move, the others would be waiting a long time before they could use it. By taking turns, each person can put one box on at a time, and they all can get their work done. It's not as fast as having a conveyor belt of your own, but it's not as slow as having to wait until everyone else is finished.

Local-Area Networks and Wide-

Area Networks

A **local-area network** (LAN) is designed to span short distances and connect a relatively small number of computers. Usually, a LAN connects the machines in only one building or in a single room. LANs are convenient to install and manage and are highly reliable. As computers became increasingly small and versatile, LANs provided an inexpensive way to share information throughout an organization. However, having a LAN is like having a telephone system that allows you to call only the people in your own town. We need to be able to share information across longer distances.

Key Concept
A local-area network (LAN) is an effective way to share information and resources throughout an organization.

A **wide-area network** (WAN) connects two or more LANs, often across long distances. Usually one computer on each LAN is dedicated to handling the communication across a WAN. This technique relieves the other computers in a LAN from having to perform the details of long-distance communication. **Figure 1.17** shows several LANs connected into a WAN. The LANs connected by a WAN are often owned by different companies or organizations and might even be located in different countries.

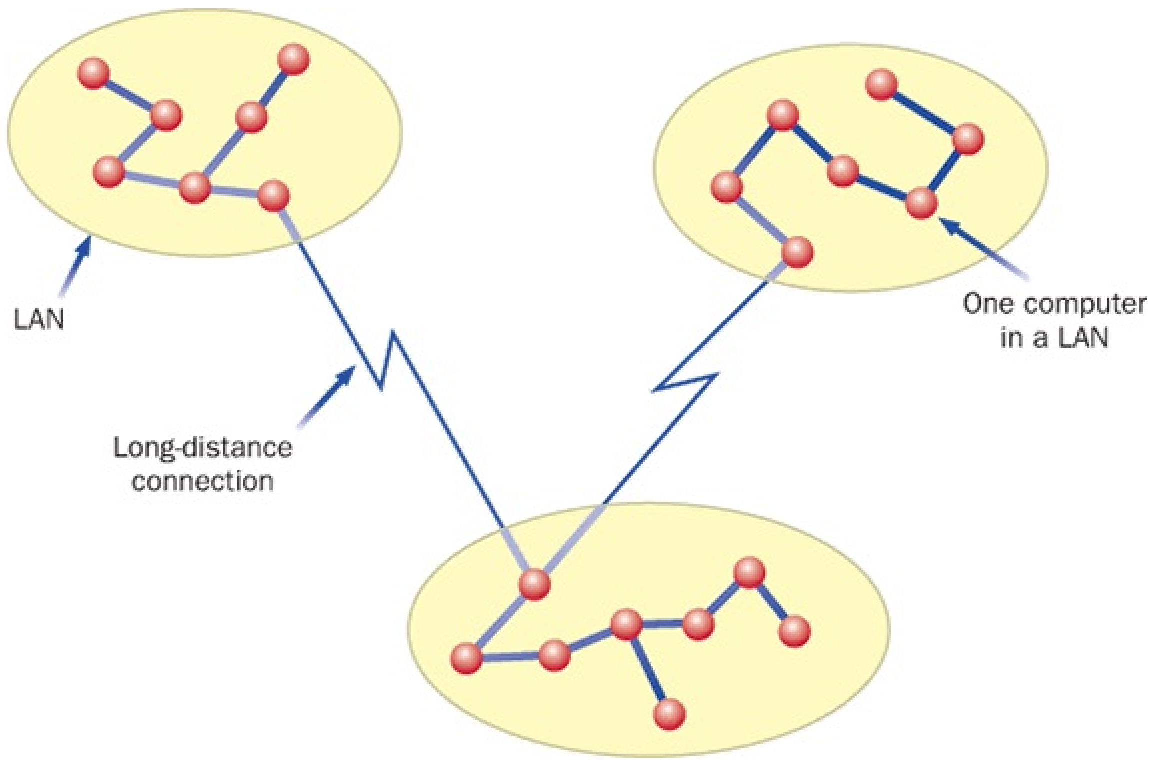


Figure 1.17 LANs connected into a WAN

The Internet

Throughout the 1970s, an agency in the Department of Defense known as the Advanced Research Projects Agency (ARPA) funded several projects to explore network technology. One result of these efforts was the ARPANET, a wide-area network that eventually became known as the Internet. The **Internet** is a network of networks. The term Internet comes from the WAN concept of *internetworking*— connecting many smaller networks together.

In 1983, there were fewer than 600 computers connected to the Internet. At the present time, the Internet serves billions of users worldwide. As more and more computers connected to the Internet, the task of keeping up with the larger number of users and heavier traffic became difficult. New technologies have replaced the ARPANET several times since the initial development, each time providing more capacity and faster processing.

Key Concept
The Internet is a wide-area network (WAN) that spans the globe.

A *protocol* is a set of rules that governs how two things communicate. The software that controls the movement of messages across the Internet must conform to a set of protocols called **TCP/IP** (pronounced by spelling out the letters, T-C-P-I-P). TCP stands for *Transmission Control Protocol*, and IP stands for *Internet Protocol*. The IP software defines how information is formatted and transferred from the source to the destination. The TCP software handles problems such as pieces of information arriving out of their original order or information getting lost, which can happen if too much information converges at one location at the same time.

Every computer connected to the Internet has an **IP address** that uniquely identifies it among all other computers on the Internet. An example of an IP address is 204.192.116.2. Fortunately, the users of the Internet rarely have to deal with IP addresses. The Internet allows each computer to be given a name. Like IP addresses, the names must be unique. The Internet name of a computer is often referred to as its **Internet address**. An example of Internet address is `hector.vt.edu.`

Key Concept
Every computer connected to the Internet has an IP address that uniquely identifies it.

The first part of an Internet address is the local name of a specific computer. The rest of the address is the **domain name**, which indicates the organization to which the computer belongs. For example, `vt.edu` is the domain name for the network of computers at Virginia Tech, and `hector` is the name of a particular computer on that campus. Because the domain names are unique, many organizations can have a computer named `hector` without confusion. Individual departments might be assigned *subdomains* that are added to the basic domain name to uniquely distinguish their set of computers within the larger organization. For example, the cs.vt.edu subdomain is devoted to the Department of Computer Science at Virginia Tech.

The last part of each domain name, called a **top-level domain** (TLD), usually indicates the type of organization to which the computer belongs. The TLD `edu` indicates an educational institution. The TLD `com` usually refers to a commercial business. Another common TLD is `org`, used by nonprofit organizations. Other top-level domains include `biz`, `info`, `jobs`, and `name`. Many computers, especially those outside of the United States, use a country-code top-level domain that denotes the country of origin, such as `uk` for the United Kingdom or `au` for Australia.

When an Internet address is referenced, it gets translated to its corresponding IP address, which is used from that point on. The software that does this translation is called the **Domain Name System** (DNS). Each organization connected to the Internet operates a **domain server** that maintains a list of all computers at that organization and their IP addresses. You provide the name, and the domain server gives back a number. If the local domain server does not have the IP address for the name, it contacts another domain server that does.

Initially, the primary use of interconnected computers was to send electronic mail, but Internet capabilities grew quickly. One of the most significant uses of the Internet today is the World Wide Web.

The World Wide Web

The Internet gives us the capability to exchange information. The **World Wide Web** (also known as WWW or simply the Web) makes the exchange of information easy for humans. Web software provides a common user interface through which many different types of information can be accessed with the click of a mouse.

Key Concept
The World Wide Web is software that makes sharing information across a network easy for humans.

The Web is based on the concepts of hypertext and hypermedia. The term **hypertext** was coined in 1965 by Ted Nelson. It describes a way to organize information so that the flow of ideas was not constrained to a linear progression. Paul Otlet (1868–1944), considered by some to be the father of information science, envisioned that concept as a way to manage large amounts of information. The underlying idea is that documents can be linked at various points according to natural relationships so that the reader can jump from one document to another, following the appropriate path for that reader's needs. When other media components are incorporated, such as graphics, sound, animations, and video, the resulting organization is called **hypermedia**.

The terms Internet and World Wide Web are sometimes used interchangeably, but there are important differences between the two. The Internet makes it possible to communicate via computers around the world. The Web makes that communication a straightforward and enjoyable activity. The Web is essentially a distributed information service based on a set of software applications.

A **browser** is a software tool that loads and formats Web documents for viewing. *Mosaic*, the first graphical interface browser for the Web, was released in 1993. The designer of a Web document refers to other Web information that might be anywhere on the Internet. Popular browsers include Google Chrome, Apple Safari, Mozilla Firefox, and Opera. The use of Internet Explorer, from Microsoft, has recently declined with the rise of alternatives. The Microsoft Edge browser has recently replaced Internet Explorer.

A computer dedicated to providing access to Web documents is called a *Web server*. Browsers load and interpret documents provided by a Web server. Many such documents are formatted using the *HyperText Markup Language* (HTML). The Java programming language has an intimate relationship with Web processing, because links to Java programs can be embedded in HTML documents and executed through Web browsers.

Uniform Resource Locators

Information on the Web is found by identifying a *Uniform Resource Locator* (URL, pronounced by spelling out the letters U-R-L). A **URL** uniquely specifies documents and other information for a browser to obtain and display. The following is an example URL:

```
http://www.google.com
```

The Web site at this particular URL is the home of the well-known Google *search engine*, which enables you to search the Web for information using particular words or phrases.

A URL contains several pieces of information. The first piece is a protocol, which determines the way the browser transmits and processes information. The second piece is the Internet address of the machine on which the document is stored. The third piece of information is the file name or resource of interest. If no file name is given, as is the case with the Google URL, the Web server usually provides a default page.

Key Concept
A URL uniquely specifies documents and other information found on the Web for a browser to

obtain and display.

Let’s look at another example URL:

```
https://www.whitehouse.gov/issues/education
```

In this URL, the protocol is https, which stands for a secure version of the *HyperText Transfer Protocol*. The machine referenced is `www` (a typical reference to a Web server), found at domain **`whitehouse.gov`**. Finally, `issues/education` represents a file (or a reference that generates a file) to be transferred to the browser for viewing. Many other forms for URLs exist, but this form is the most common.

Self-Review Questions

(see answers in **Appendix L**)

SR 1.14 What is a file server?
SR 1.15 What is the total number of communication lines needed for a fully connected point-to-point network of five computers? Six computers?
SR 1.16 Describe a benefit of having computers on a network share a communication line. Describe a cost/drawback of sharing a communication line.

SR 1.17 What is the etymology of the word Internet?

SR 1.18 The TCP/IP set of protocols describes communication rules for software that uses the Internet. What does TCP stand for? What does IP stand for?

SR 1.19 Explain the parts of the following URLs:

a. duke.csc.villanova.edu/jss/examples.html
b. **java.sun.com/products/index.html**

1.4 The Java Programming Language

Let’s now turn our attention to the software that makes a computer system useful. A program is written in a particular *programming language* that uses specific words and symbols to express the problem solution. A programming language defines a set of rules that determines exactly how a programmer can combine the words and symbols of the language into *programming statements*, which are the instructions that are carried out when the program is executed.

Since the inception of computers, many programming languages have been created. We use the Java language in this book to demonstrate various programming concepts and techniques. Although our main goal is to learn these underlying software development principles, an important side effect will be to become proficient in the development of Java programs specifically.

The development of the Java programming language began in 1991 by James Gosling at Sun Microsystems. The language initially was called Oak, then Green, and ultimately Java. Java was introduced to the public in 1995 and has gained tremendous popularity since. In 2010, Sun Microsystems was purchased by Oracle.

There are variations of the Java Platform, including the Standard Edition, which is the mainstream version of the language; the Enterprise Edition, which includes extra libraries to support large-scale system development; and the Micro Edition, which is specifically for developing software for portable devices such as cell phones. This book focuses on the Standard Edition.

Furthermore, the Java Standard Edition has gone through several revisions over the years, extending its capabilities and making changes to certain aspects of it. The table below lists the versions and some of the new features. Note that they changed the numbering technique with version 5. Readers of this book should be using version 6 or later.

Version	Year	New Features
1.0	1996	Initial deployment
1.1	1997	Inner classes
1.2	1998	Collections framework, Swing graphics
1.3	2000	Sound framework
1.4	2002	Assertions, XML support, regular expressions
5	2004	Generics, for-each loop, autoboxing, enumerations, annotations, variable-length parameter lists
6	2006	GUI improvements, various library updates

7	2011	Use of strings in a switch, other language changes
8	2014	JavaFX, lambda expressions, date and time API

Some parts of early Java technologies have been *deprecated*, which means they are considered old-fashioned and should not be used. When it is important, we point out deprecated elements and discuss their preferred alternatives.

One reason Java attracted some initial attention was because it was the first programming language to deliberately embrace the concept of writing programs (called applets) that can be executed using the Web. Since then, the techniques for creating a Web page that has dynamic, functional capabilities have expanded dramatically.

Java is an *object-oriented programming language*. Objects are the fundamental elements that make up a program. The principles of object-oriented software development are the cornerstone of this book. We explore object-oriented programming concepts later in this chapter and throughout the rest of the book.

Key Concept
This book focuses on the principles of object-oriented programming.

The Java language is accompanied by a library of extra software that we can use when developing programs. This software is referred to as the *Java API*, which stands for Application Programmer Interface, or simply the *standard class library*. The Java API provides the ability to create graphics, communicate over networks, and interact with databases, among many other features. The Java API is huge and quite versatile. We won't be able to cover all aspects of the library, though we will explore several of them.

Java is used in commercial environments all over the world. It is one of the fastest growing programming technologies of all time. So not only is it a good language in which to learn programming concepts, it is also a practical language that will serve you well in the future.

A Java Program

Let's look at a simple but complete Java program. The program in **Listing 1.1** prints two sentences to the screen. This particular program prints a quote by Abraham Lincoln. The output is shown below the program listing.

Listing 1.1

```
//***************************************************************

//  Lincoln.java       Author: Lewis/Loftus
```

```
//
//  Demonstrates the basic structure of a Java application.
//************************************************************


public class Lincoln
{
   //---------------------------------------------------------
--------
   //  Prints a presidential quote.
   //---------------------------------------------------------
--------
   public static void main(String[] args)
   {
      System.out.println("A quote by Abraham Lincoln:");

      System.out.println("Whatever you are, be a good one.");
   }
}
```

Output

```
A quote by Abraham Lincoln:
Whatever you are, be a good one.
```

All Java applications have a similar basic structure. Despite its small size and simple purpose, this program contains several important features. Let’s carefully dissect it and examine its pieces.

The first few lines of the program are comments, which start with the `//` symbols and continue to the end of the line. Comments don’t affect what the program does but are included to make the program easier to understand by humans. Programmers can and should include comments as needed throughout a program to clearly identify the purpose of the program and describe any special processing. Any written comments or documents, including a user’s guide and technical references, are called **documentation**. Comments included in a program are called ***inline documentation***.

Key Concept
Comments do not affect a program’s processing; instead, they serve to facilitate human comprehension.

The rest of the program is a *class definition*. This class is called `Lincoln`, though we could have named it just about anything we wished. The class definition runs from the first opening brace (`{`) to the final closing brace (`}`) on the last line of the program. All Java programs are defined using class definitions.

VideoNote Overview of program elements.

Inside the class definition are some more comments describing the purpose of the `main` method, which is defined directly below the comments. A **method** is a group of programming statements that is given a name. In this case, the name of the method is `main` and it contains only two programming statements. Like a class definition, a method is also delimited by braces.

All Java applications have a `main` method, which is where processing begins. Each programming statement in the `main` method is executed, one at a time in order, until the end of the method is reached. Then the program ends, or *terminates*. The `main` method definition in a Java program is always preceded by the words `public`, `static`, and `void`, which we examine later in the text. The use of `String` and `args` does not come into play in this particular program. We describe these later also.

The two lines of code in the `main` method invoke another method called `println` (pronounced print line). We *invoke*, or *call*, a method when we want it to execute. The `println` method prints the specified characters to the screen. The characters to be printed are represented as a *character string*, enclosed in double quote characters (`"`). When the program is executed, it calls the `println` method to print the first statement, calls it again to print the second statement, and then, because that is the last line in the `main` method, the program terminates.

The code executed when the `println` method is invoked is not defined in this program. The `println` method is part of the `System.out` object, which is part of the Java standard class library. It's not technically part of the Java language, but is always available for use in any Java program. We explore the `println` method in more detail in **Chapter 2**.

Comments

Let's examine comments in more detail. **Comments** are the only language feature that allows programmers to compose and communicate their thoughts independent of the code. Comments should provide insight into the programmer's original intent. A program is often used for many years, and often many modifications are made to it over time. The original programmer often will not remember the details of a particular program when, at some point in the future, modifications are required. Furthermore, the original programmer is not always available to make the changes; thus, someone completely unfamiliar with the program will need to understand it. Good documentation is therefore essential.

As far as the Java programming language is concerned, the content of comments can be any text whatsoever. Comments are ignored by the computer; they do not affect how the program executes.

The comments in the `Lincoln` program represent one of two types of comments allowed in Java. The comments in `Lincoln` take the following form:

```
// This is a comment.
```

This type of comment begins with a double slash (`//`) and continues to the end of the line. You cannot have any characters between the two slashes. The computer ignores any text after the double slash to the end of the line. A comment can follow code on the same line to document that particular line, as in the following example:

```
System.out.println("Monthly Report");  // always use this title
```

The second form a Java comment may have is the following:

```
/* This is another comment. */
```

This comment type does not use the end of a line to indicate the end of the comment. Anything between the initiating slash-asterisk (`/*`) and the terminating asterisk-slash (`*/`) is part of the comment, including the invisible *newline* character that represents the end of a line. Therefore, this type of comment can extend over multiple lines. No space can be between the slash and the asterisk.

If there is a second asterisk following the `/*` at the beginning of a comment, the content of the comment can be used to automatically generate external documentation about your program by using a tool called *javadoc*. More information about javadoc is given in **Appendix I**.

The two basic comment types can be used to create various documentation styles, such as the following:

```
// This is a comment on a single line.

//------------------------------------------------------------
// Some comments such as those above methods or classes
// deserve to be blocked off to focus special attention
// on a particular aspect of your code. Note that each of
// these lines is technically a separate comment.
//------------------------------------------------------------

/*
   This is one comment
   that spans several lines.
*/
```

Programmers often concentrate so much on writing code that they focus too little on documentation. You should develop good commenting practices and follow them habitually. Comments should be well written, often in complete sentences. They should not belabor the obvious but should provide appropriate insight into the intent of the code. The following examples are *not* good comments:

```
System.out.println("hello");   // prints hello
System.out.println("test");    // change this later
```

The first comment paraphrases the obvious purpose of the line and does not add any value to the statement. It is better to have no comment than a useless one. The second comment is ambiguous. What should be changed later? When is later? Why should it be changed?

Key Concept
Inline documentation should provide insight into your code. It should not be ambiguous or belabor the obvious.

Identifiers and Reserved Words

The various words used when writing programs are called **identifiers**. The identifiers in the `Lincoln` program are `class`, `Lincoln`, `public`, `static`, `void`, `main`, `String`, `args`, `System`, `out`, and `println`. These fall into three categories:

- words that we make up when writing a program (`Lincoln` and `args`)
- words that another programmer chose (`String`, `System`, `out`, `println`, and `main`)
- words that are reserved for special purposes in the language (`class`, `public`, `static`, and `void`)

While writing the program, we simply chose to name the class `Lincoln`, but we could have used one of many other possibilities. For example, we could have called it `Quote`, or `Abe`, or `GoodOne`. The identifier `args` (which is short for arguments) is often used in the way we use it in `Lincoln`, but we could have used just about any other identifier in its place.

The identifiers `String`, `System`, `out`, and `println` were chosen by other programmers. These words are not part of the Java language. They are part of the Java standard library of predefined code, a set of classes and methods that someone has already written for us. The authors of that code chose the identifiers in that code—we're just making use of them.

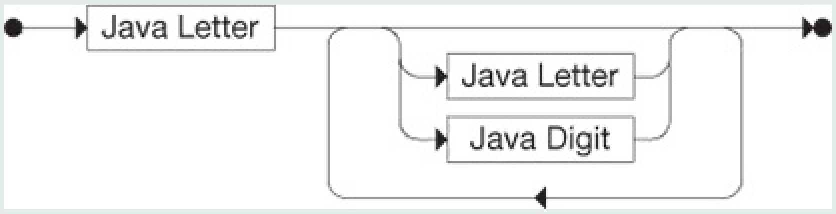
Reserved words are identifiers that have a special meaning in a programming language and can only be used in predefined ways. A reserved word cannot be used for any other purpose, such as naming a class or method. In the `Lincoln` program, the reserved words used are `class`, `public`, `static`, and `void.` Throughout the book, we show Java reserved words in blue type. **Figure 1.18** lists all of the Java reserved words in alphabetical order. The words marked with an asterisk have been reserved, but currently have no meaning in Java.

abstract	default	goto*	package	this
assert	do	if	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	true
case	extends	int	short	try
catch	false	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const*	float	new	switch	
continue	for	null	synchronized	

Figure 1.18 Java reserved words

An identifier that we make up for use in a program can be composed of any combination of letters, digits, the underscore character (`_`), and the dollar sign (`$`), but it cannot begin with a digit. Identifiers may be of any length. Therefore, `total`, `label7`, `nextStockItem`, `NUM_BOXES`, and `$amount` are all valid identifiers, but `4th_word` and `coin#value` are not valid.

Identifier



An identifier is a letter followed by zero or more letters and digits. A Java Letter includes the 26 English alphabetic characters in both uppercase and lowercase, the `$` and `_` (underscore) characters, as well as alphabetic characters from other languages. A Java Digit includes the digits `0` through `9`.

Examples:

```
total
MAX_HEIGHT
num1
Keyboard
System
```

Both uppercase and lowercase letters can be used in an identifier, and the difference is important. Java is **case sensitive**, which means that two identifier names that differ only in the case of their letters are considered to be different identifiers. Therefore, `total`, `Total`, `ToTaL`, and `TOTAL` are all different identifiers. As you can imagine, it is not a good idea to use multiple identifiers that differ only in their case, because they can be easily confused.

Although the Java language doesn't require it, using a consistent case format for each kind of identifier makes your identifiers easier to understand. There are various Java conventions regarding identifiers that should be followed, though technically they don't have to be. For example, we use *title case* (uppercase for the first letter of each word) for class names. Throughout the text, we describe the preferred case style for each type of identifier when it is first encountered.

Key Concept
Java is case sensitive. The uppercase and lowercase versions of a letter are distinct.

While an identifier can be of any length, you should choose your names carefully. They should be descriptive but not verbose. You should avoid meaningless names such as `a` or `x`. An exception to this rule can be made if the short name is actually descriptive, such as using `x` and `y` to represent (*x*, *y*) coordinates on a two-dimensional grid. Likewise, you should not use unnecessarily long names, such as the identifier `theCurrentItemBeingProcessed`. The name `currentItem` would serve just as well. As you might imagine, the use of identifiers that are verbose is a much less prevalent problem than the use of names that are not descriptive.

You should always strive to make your programs as readable as possible. Therefore, you should always be careful when abbreviating words. You might think `curStVal` is a good name to represent the current stock value, but another person trying to understand the code may have trouble figuring out what you meant. It might not even be clear to you two months after writing it.

Key Concept
Identifier names should be descriptive and readable.

White Space

All Java programs use *white space* to separate the words and symbols used in a program. **White space** consists of blanks, tabs, and newline characters. The phrase “white space” refers to the fact that, on a white sheet of paper with black printing, the space between the words and symbols is white. The way a programmer uses white space is important because it can be used to emphasize parts of the code and can make a program easier to read.

Key Concept
Appropriate use of white space makes a program easier to read and understand.

Except when it’s used to separate words, the computer ignores white space. It does not affect the execution of a program. This fact gives programmers a great deal of flexibility in how they format a program. The lines of a program should be divided in logical places, and certain lines should be indented and aligned so that the program’s underlying structure is clear.

Because white space is ignored, we can write a program in many different ways. For example, taking white space to one extreme, we could put as many words as possible on each line. The code in **Listing 1.2**, the `Lincoln2` program, is formatted quite differently from `Lincoln` but prints the same message.

Listing 1.2

```
//************************************************************

//  Lincoln2.java        Author: Lewis/Loftus
//
//  Demonstrates a poorly formatted, though valid, program.
//************************************************************

public class Lincoln2{public static void main(String[]args){
System.out.println("A quote by Abraham Lincoln:");
System.out.println("Whatever you are, be a good one.");}}
```

Output

```
 A quote by Abraham Lincoln:
Whatever you are, be a good one.
```

Taking white space to the other extreme, we could write almost every word and symbol on a different line with varying amounts of spaces, such as `Lincoln3`, shown in **Listing 1.3**.

Listing 1.3

```
//************************************************************
```

```
//  Lincoln3.java       Author: Lewis/Loftus
//
//  Demonstrates another valid program that is poorly
formatted.
//************************************************************

          public       class
     Lincoln3
   {
                 public
   static
      void
  main
        (
String
          []
    args                       )
  {
  System.out.println    (
"A quote by Abraham Lincoln:"           )
  ;        System.out.println
            (
       "Whatever you are, be a good one."
      )
  ;
}
          }
```

Output

```
 A quote by Abraham Lincoln:
Whatever you are, be a good one.
```

Key Concept
You should adhere to a set of guidelines that establish the way you format and document your programs.

All three versions of `Lincoln` are technically valid and will execute in the same way, but they are radically different from a reader's point of view. Both the latter examples show poor style and make the program difficult to understand. You may be asked to adhere to particular guidelines when you write your programs. A software development company often has a programming style policy that it requires its programmers to follow. In any case, you should adopt and consistently use a set of style guidelines that increase the readability of your code.

Self-Review Questions

(see answers in **Appendix L**)

SR 1.20 When was the Java programming language developed? By whom? When was it introduced to the public?

SR 1.21 Where does processing begin in a Java application?

SR 1.22 What do you predict would be the result of the following line in a Java program?

```
System.out.println("Hello");   // prints hello
```

SR 1.23 What do you predict would be the result of the following line in a Java program?

```
// prints hello  System.out.println("Hello");
```

SR 1.24 Which of the following are not valid Java identifiers? Why?

a. `RESULT`
b. `result`
c. `12345`
d. `x12345y`
e. `black&white`
f. `answer_7`

SR 1.25 Suppose a program requires an identifier to represent the sum of the test scores of a class of students. For each of the following names, state whether or not each is a good name to use for the identifier. Explain your answers.

a. `x`

b. `scoreSum`
c. `sumOfTheTestScoresOfTheStudents`
d. `smTstScr`

SR 1.26 What is white space? How does it affect program execution? How does it affect program readability?

1.5 Program Development

The process of getting a program running involves various activities. The program has to be written in the appropriate programming language, such as Java. That program has to be translated into a form that the computer can execute. Errors can occur at various stages of this process and must be fixed. Various software tools can be used to help with all parts of the development process as well. Let’s explore these issues in more detail.

Programming Language Levels

Suppose a particular person is giving travel directions to a friend. That person might explain those directions in any one of several languages, such as English, Russian, or Italian. The directions are the same no matter which language is used to explain them, but the manner in which the directions are expressed is different. The friend must be able to understand the language being used in order to follow the directions.

Similarly, a problem can be solved by writing a program in one of many programming languages, such as Java, Ada, C, C++, C#, Pascal, and Smalltalk. The purpose of the program is essentially the same no matter which language is used, but the particular statements used to express the instructions, and the overall organization of those instructions, vary with each language. A computer must be able to understand the instructions in order to carry them out.

Programming languages can be categorized into the following four groups. These groups basically reflect the historical development of computer languages.

- machine language
- assembly language
- high-level languages
- fourth-generation languages

In order for a program to run on a computer, it must be expressed in that computer's *machine language*. Each type of CPU has its own language.

Each machine language instruction can accomplish only a simple task. For example, a single machine language instruction might copy a value into a register or compare a value to zero. It might take four separate machine language instructions to add two numbers together and to store the result. However, a computer can do millions of these instructions in a second, and therefore many simple commands can be executed quickly to accomplish complex tasks.

Key Concept

All programs must be translated to a particular CPU's machine language in order to be executed.

Machine language code is expressed as a series of binary digits and is extremely difficult for humans to read and write. Originally, programs were entered into the computer by using switches or some similarly tedious method. Early programmers found these techniques to be time consuming and error prone.

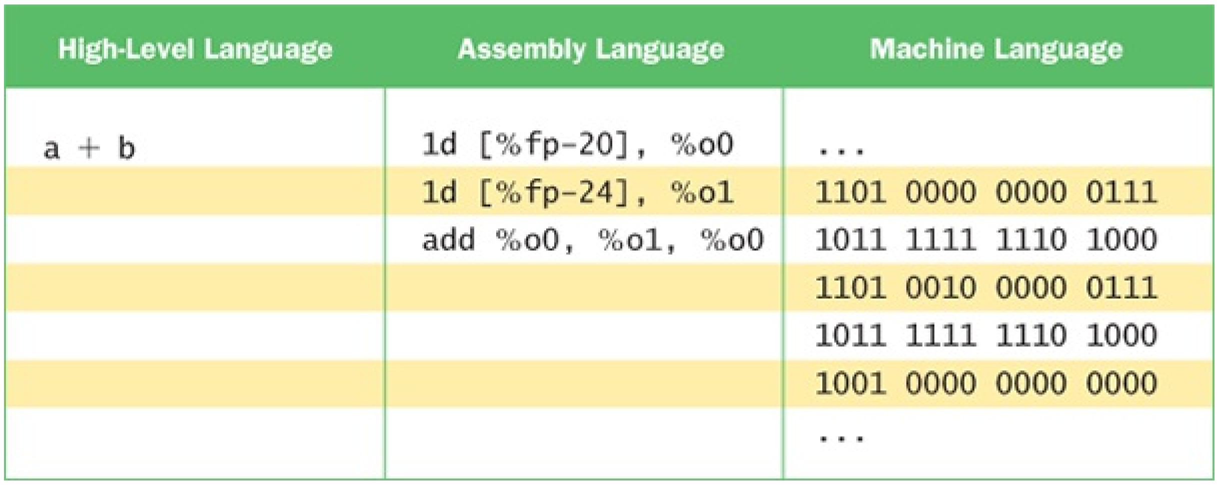
These problems gave rise to the use of *assembly language*, which replaced binary digits with **mnemonics**, short English-like words that represent commands or data. It is much easier for programmers to deal with words than with binary digits. However, an assembly language program cannot be executed directly on a computer. It must first be translated into machine language.

Generally, each assembly language instruction corresponds to an equivalent machine language instruction. Therefore, similar to machine language, each assembly language instruction accomplishes only a simple operation. Although assembly language is an improvement over machine code from a programmer's perspective, it is still tedious to use. Both assembly language and machine language are considered *low-level languages*.

Key Concept
High-level languages allow a programmer to ignore the underlying details of machine language.

Today, most programmers use a *high-level language* to write software. A high-level language is expressed in English-like phrases, and thus is easier for programmers to read and write. A single high-level language programming statement can accomplish the equivalent of many—perhaps hundreds—of machine language instructions. The term high-level refers to the fact that the programming statements are expressed in a way that is far removed from the machine language that is ultimately executed. Java is a high-level language, as are Ada, C+ +, Smalltalk, and many others.

Figure 1.19 shows equivalent expressions in a high-level language, assembly language, and machine language. The expressions add two numbers together. The assembly language and machine language in this example are specific to a Sparc processor.



High-Level Language	Assembly Language	Machine Language
a + b	ld [%fp-20], %o0	...
	ld [%fp-24], %o1	1101 0000 0000 0111
	add %o0, %o1, %o0	1011 1111 1110 1000
		1101 0010 0000 0111
		1011 1111 1110 1000
		1001 0000 0000 0000
		...

Figure 1.19 A high-level expression and its assembly language and machine language equivalent

The high-level language expression in **Figure 1.19** is readable and intuitive for programmers. It is similar to an algebraic expression. The equivalent assembly language code is somewhat readable, but it is more verbose and less intuitive. The machine language is basically unreadable and much longer. In fact, only a small portion of the binary machine code to add two numbers together is shown in **Figure 1.19**. The complete machine language code for this particular expression is over 400 bits long.

A high-level language insulates programmers from needing to know the underlying machine language for the processor on which they are working. But high-level language code must be translated into machine language in order to be executed.

Some programming languages are considered to operate at an even higher level than high-level languages. They might include special facilities for automatic report generation or interaction with a database.

These languages are called *fourth-generation languages*, or simply 4GLs, because they followed the first three generations of computer programming: machine, assembly, and high-level.

Editors, Compilers, and Interpreters

Several special-purpose programs are needed to help with the process of developing new programs. They are sometimes called software tools because they are used to build programs. Examples of basic software tools include an editor, a compiler, and an interpreter.

Initially, you use an *editor* as you type a program into a computer and store it in a file. There are many different editors with many different features. You should become familiar with the editor you will use regularly because it can dramatically affect the speed at which you enter and modify your programs.

Figure 1.20 shows a very basic view of the program development process. After editing and saving your program, you attempt to translate it from high-level code into a form that can be executed. That translation may result in errors, in which case you return to the editor to make changes to the code to fix the problems. Once the translation occurs successfully, you can execute the program and evaluate the results. If the results are not what you want, or if you want to enhance your existing program, you again return to the editor to make changes.

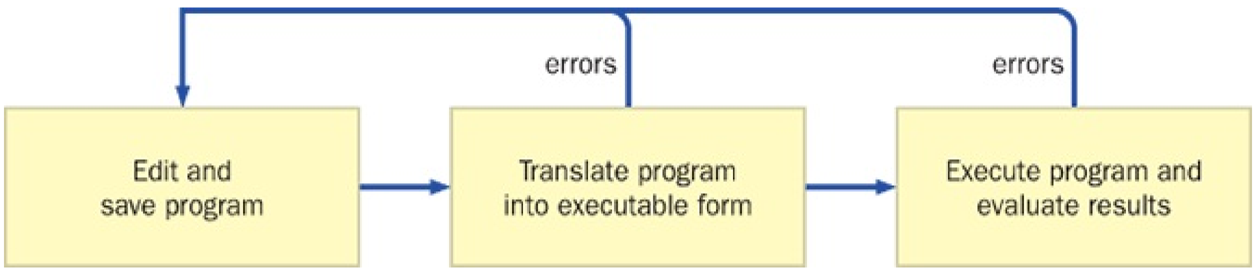

errors
errors
Edit and save program
Translate program into executable form
Execute program and evaluate results

Figure 1.20 Editing and running a program

The translation of source code into (ultimately) machine language for a particular type of CPU can occur in a variety of ways. A **compiler** is a program that translates code in one language to equivalent code in another language. The original code is called *source code*, and the language into which it is translated is called the *target language*. For many traditional compilers, the source code is translated directly into a particular machine language. In that case, the translation process occurs once and the resulting executable program can be run whenever needed on any computer using that type of CPU.

An **interpreter** is similar to a compiler but has an important difference. An interpreter interweaves the translation and execution activities. A small part of the source code, such as one statement, is translated and executed. Then another statement is translated and executed, and so on. One advantage of this technique is that it eliminates the need for a separate compilation phase. However, the program generally runs more slowly because the translation process occurs during each execution.

Key Concept
A Java compiler translates Java source code into Java bytecode, a low-level, architecture-neutral representation of the program.

The process generally used to translate and execute Java programs combines the use of a compiler and an interpreter. This process is pictured in **Figure 1.21**. The Java compiler translates Java source code into Java **bytecode**, which is a representation of the program in a low-level form similar to machine language code. A Java interpreter called the *Java Virtual Machine* (JVM) executes the Java bytecode.

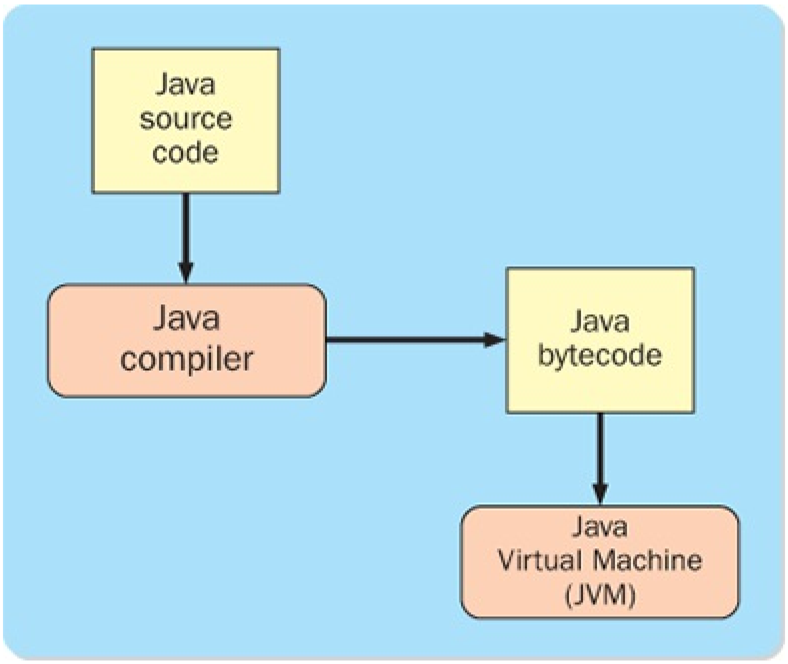


Figure 1.21 The Java translation and execution process

The difference between Java bytecode and true machine language code is that Java bytecode is not tied to any particular processor type. This approach has the distinct advantage of making Java *architecture neutral*, and therefore easily portable from one machine type to another. The only restriction is that there must be a JVM on each machine.

Since the compilation process translates the high-level Java source code into a low-level representation, the interpretation process by the JVM is far more efficient than interpreting high-level code directly. Executing a program by interpreting its bytecode is still slower than executing machine code directly, but it is fast enough for most applications.

Development Environments

A software *development environment* is the set of tools used to create, test, and modify a program. Some development environments are available for free while others, which may have advanced features, must be purchased. Some environments are referred to as *integrated development environments* (IDEs) because they integrate various tools into one software program and provide a convenient graphical user interface.

Any development environment will contain certain key tools, such as a Java compiler and interpreter. Some will include a **debugger**, which helps you find errors in a program. Other tools that may be included are documentation generators, archiving tools, and tools that help you visualize your program structure.

VideoNote Comparison of Java IDEs.

Included in the download of the Java Standard Edition is the Java *Software Development Kit* (SDK), which is sometimes referred to simply as the *Java Development Kit* (JDK). The Java SDK contains the core development tools needed to get a Java program up and running, but it is not an integrated environment. The commands for compilation and interpretation are executed on the command line. That is, the SDK does not have a GUI. It also does not include an editor, although any editor that can save a document as simple text can be used.

One of the most popular Java IDEs is called Eclipse (see **www.eclipse.org**). Eclipse is an *open-source* project, meaning that it is developed by a wide collection of programmers and is available for free. Other popular Java IDEs include jEdit (**www.jedit.org**), DrJava (**drjava.sourceforge.net**), jGRASP (**www.jgrasp.com**), and BlueJ (**www.bluej.org**).

Various other Java development environments are available. A Web search will unveil dozens of them. The choice of which development environment to use is important. The more you know about the capabilities of your environment, the more productive you can be during program development.

Key Concept
Many different development environments exist to help you create and modify Java programs.

Syntax and Semantics

Each programming language has its own unique *syntax*. The **syntax rules** of a language dictate exactly how the vocabulary elements of the language can be combined to form statements. These rules must be followed in order to create a program. We've already discussed several Java syntax rules. For instance, the fact that an identifier cannot begin with a digit is a syntax rule. The fact that braces are used to begin and end classes and methods is also a syntax rule. **Appendix L** formally defines the basic syntax rules for the Java programming language, and specific rules are highlighted throughout the text.

During compilation, all syntax rules are checked. If a program is not syntactically correct, the compiler will issue error messages and will not produce bytecode. Java has a similar syntax to the programming languages C and C++, and therefore the look and feel of the code is familiar to people with a background in those languages.

The **semantics** of a statement in a programming language define what will happen when that statement is executed. Programming languages are generally unambiguous, which means the semantics of a program are well defined. That is, there is one and only one interpretation for each statement. On the other hand, the **natural languages** that humans use to communicate, such as English and Italian, are full of ambiguities. A sentence can often have two or more different meanings. For example, consider the following sentence:

```
Time flies like an arrow.
```

The average human is likely to interpret this sentence as a general observation: that time moves quickly in the same way that an arrow moves quickly. However, if we interpret the word *time* as a verb (as in “run the 50-yard dash and I’ll time you”) and the word *flies* as a noun (the plural of fly), the interpretation changes completely. We know that arrows don’t time things, so we wouldn’t normally interpret the sentence that way, but it is a valid interpretation of the words in the sentence. A computer would have a difficult time trying to determine which meaning is intended. Moreover, this sentence could describe the preferences of an unusual insect known as a “time fly,” which might be found near an archery range. After all, fruit flies like a banana.

Key Concept
Syntax rules dictate the form of a program. Semantics dictate the meaning of the program statements.

The point is that one specific English sentence can have multiple valid meanings. A computer language cannot allow such ambiguities to exist. If a programming language instruction could have two different meanings, a computer would not be able to determine which one should be carried out.

Errors

Several different kinds of problems can occur in software, particularly during program development. The term *computer error* is often misused and varies in meaning depending on the situation. From a user’s point of view, anything that goes awry when interacting with a machine can be called a computer error. For example, suppose you charged a $23 item to your credit card, but when you received the bill, the item was listed at $230. After you have the problem fixed, the credit card company apologizes for the “computer error.” Did the computer arbitrarily add a zero to the end of the number, or did it perhaps multiply the value by 10? Of course not. A computer follows the commands we give it and operates on the data we provide. If our programs are wrong or our data inaccurate, then we cannot expect the results to be correct. A common phrase used to describe this situation is “garbage in, garbage out.”

Key Concept
The programmer is responsible for the accuracy and reliability of a program.

You will encounter three kinds of errors as you develop programs:

- compile-time error
- run-time error
- logical error

The compiler checks to make sure you are using the correct syntax. If you have any statements that do not conform to the syntactic rules of the language, the compiler will produce a *syntax error*. The compiler also tries to find other problems, such as the use of incompatible types of data. The syntax might be technically correct, but you may be attempting to do something that the language doesn’t semantically allow. Any error identified by the compiler is called a *compile-time error*. If a compile-time error occurs, an executable version of the program is not created.

Key Concept
A Java program must be syntactically correct or the compiler will not produce bytecode.

The second kind of problem occurs during program execution. It is called a *run-time error* and causes the program to terminate abnormally. For example, if we attempt to divide by zero, the program will "crash" and halt execution at that point. Because the requested operation is undefined, the system simply abandons its attempt to continue processing your program. The best programs are *robust*; that is, they avoid as many run-time errors as possible. For example, the program code could guard against the possibility of dividing by zero and handle the situation appropriately if it arises. In Java, many run-time problems are called *exceptions* that can be caught and dealt with accordingly.

VideoNote Examples of various error types.

The third kind of software problem is a **logical error**. In this case, the software compiles and executes without complaint, but it produces incorrect results. For example, a logical error occurs when a value is calculated incorrectly or when a graphical button does not appear in the correct place. A programmer must test the program thoroughly, comparing the expected results to those that actually occur. When defects are found, they must be traced back to the source of the problem in the code and corrected. The process of finding and correcting defects in a program is called **debugging**. Logical errors can manifest themselves in many ways, and the actual root cause might be difficult to discover.

Self-Review Questions

(see answers in **Appendix L**)

SR 1.27 We all know that computers are used to perform complex jobs. In this section, you learned that a computer's instructions can do only simple tasks. Explain this apparent contradiction.
SR 1.28 What is the relationship between a high-level language and a machine language?
SR 1.29 What is Java bytecode?
SR 1.30 Select the word from the following list that best matches each of the following phrases:

assembly, compiler, high-level, IDE, interpreter, Java, low-level, machine

a. A program written in this type of language can run directly on a computer.
b. Generally, each language instruction in this type of language corresponds to an equivalent machine language instruction.
c. Most programmers write their programs using this type of language.
d. Java is an example of this type of language.
e. This type of program translates code in one language to code in another language.
f. This type of program interweaves the translation of code and the execution of the code.

SR 1.31 What do we mean by the syntax and semantics of a programming language?

SR 1.32 Categorize each of the following situations as a compile-time error, run-time error, or logical error.

a. Misspelling a Java reserved word.
b. Calculating the average of an empty list of numbers by dividing the sum of the numbers on the list (which is zero) by the size of the list (which is also zero).
c. Printing a student’s high test grade when the student’s average test grade should have been output.

1.6 Object-Oriented Programming

As we stated earlier in this chapter, Java is an object-oriented (OO) language. As the name implies, an **object** is a fundamental entity in a Java program. This book is focused on the idea of developing software by defining objects that interact with each other.

The principles of object-oriented software development have been around for many years, essentially as long as high-level programming languages have been used. The programming language Simula, developed in the 1960s, had many characteristics that define the modern OO approach to software development. In the 1980s and 1990s, object-oriented programming became wildly popular, due in large part to the development of programming languages such as C++ and Java. It is now the dominant approach used in commercial software development.

One of the most attractive characteristics of the object-oriented approach is the fact that objects can be used quite effectively to represent real-world entities. We can use a software object to represent an employee in a company, for instance. We'd create one object per employee, each with behaviors and characteristics that we need to represent. In this way, object-oriented programming allows us to map our programs to the real situations that the programs represent. That is, the object-oriented approach makes it easier to solve problems, which is the point of writing a program in the first place.

Key Concept
Object-oriented programming helps us solve problems, which is the purpose of writing a program.

Let’s discuss the general issues related to problem solving, and then explore the specific characteristics of the object-oriented approach that helps us solve those problems.

Problem Solving

In general, problem solving consists of multiple steps:

1. Understanding the problem.
2. Designing a solution.
3. Considering alternatives to the solution and refining the solution.
4. Implementing the solution.
5. Testing the solution and fixing any problems that exist.

Although this approach applies to any kind of problem solving, it works particularly well when developing software. These steps aren’t purely linear. That is, some of the activities will overlap others. But at some point, all of these steps should be carefully addressed.

The first step, understanding the problem, may sound obvious, but a lack of attention to this step has been the cause of many misguided software development efforts. If we attempt to solve a problem we don’t completely understand, we often end up solving the wrong problem or at least going off on improper tangents. Each problem has a *problem domain*, the real-world issues that are key to our solution. For example, if we are going to write a program to score a bowling match, then the problem domain includes the rules of bowling. To develop a good solution, we must thoroughly understand the problem domain.

The key to designing a problem solution is breaking it down into manageable pieces. A solution to any problem can rarely be expressed as one big task. Instead, it is a series of small cooperating tasks that interact to perform a larger task. When developing software, we don’t write one big program. We design separate pieces that are responsible for certain parts of the solution, and then integrate them with the other parts.

Key Concept

Program design involves breaking a solution down into manageable pieces.

Our first inclination toward a solution may not be the best one. We must always consider alternatives and refine the solution as necessary. The earlier we consider alternatives, the easier it is to modify our approach.

Implementing the solution is the act of taking the design and putting it in a usable form. When developing a software solution to a problem, the implementation stage is the process of actually writing the program. Too often programming is thought of as writing code. But in most cases, the act of designing the program should be far more interesting and creative than the process of implementing the design in a particular programming language.

At many points in the development process, we should test our solution to find any errors that exist so that we can fix them. Testing cannot guarantee that there aren’t still problems yet to be discovered, but it can raise our confidence that we have a viable solution.

Throughout this text, we explore techniques that allow us to design and implement elegant programs. Although we will often get immersed in these details, we should never forget that our primary goal is to solve problems.

Object-Oriented Software Principles

Object-oriented programming ultimately requires a solid understanding of the following terms:

- object
- attribute
- method
- class
- encapsulation
- inheritance
- polymorphism

In addition to these terms, there are many associated concepts that allow us to tailor our solutions in innumerable ways. This book is designed to help you evolve your understanding of these concepts gradually and naturally. This section provides an overview of these ideas at a high level to establish some terminology and provide the big picture.

We mentioned earlier that an **object** is a fundamental element in a program. A software object often represents a real object in our problem domain, such as a bank account. Every object has a **state** and a set of **behaviors**. By “state” we mean state of being—fundamental characteristics that currently define the object. For example, part of a bank account's state is its current balance. The behaviors of an object are the activities associated with the object. Behaviors associated with a bank account probably include the ability to make deposits and withdrawals.

In addition to objects, a Java program also manages primitive data. *Primitive data* includes fundamental values such as numbers and characters. Objects usually represent more interesting or complex entities.

An object's *attributes* are the values it stores internally, which may be represented as primitive data or as other objects. For example, a bank account object may store a floating point number (a primitive value) that represents the balance of the account. It may contain other attributes, such as the name of the account owner. Collectively, the values of an object's attributes define its current state.

Key Concept
Each object has a state, defined by its attributes, and a set of behaviors, defined by its methods.

As mentioned earlier in this chapter, a *method* is a group of programming statements that is given a name. When a method is invoked, its statements are executed. A set of methods is associated with an object. The methods of an object define its potential behaviors. To define the ability to make a deposit into a bank account, we define a method containing programming statements that will update the account balance accordingly.

An object is defined by a *class*. A class is the model or blueprint from which an object is created. Consider the blueprint created by an architect when designing a house. The blueprint defines the important characteristics of the house—its walls, windows, doors, electrical outlets, and so on. Once the blueprint is created, several houses can be built using it, as depicted in **Figure 1.22**.

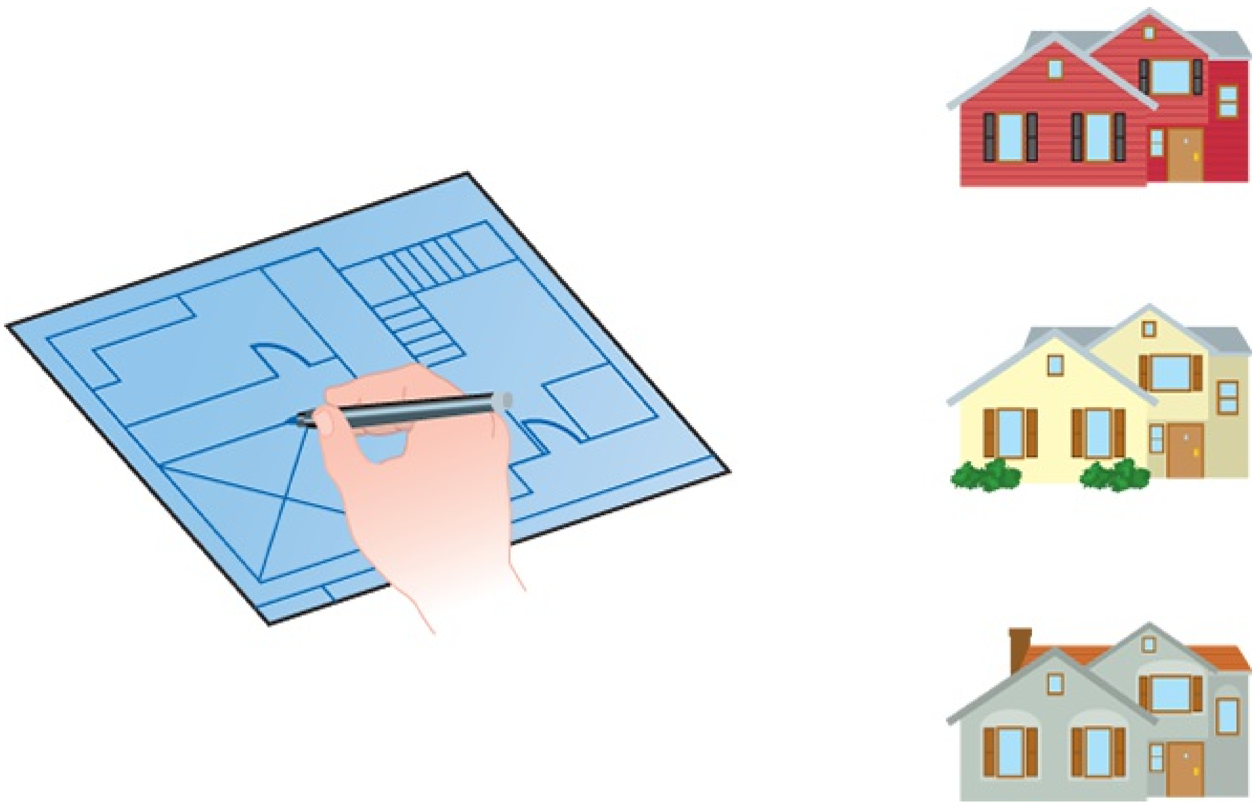


Figure 1.22 A class is used to create objects just as a house blueprint is used to create different, but similar, houses

In one sense, the houses built from the blueprint are different. They are in different locations, have different addresses, contain different furniture, and are inhabited by different people. Yet in many ways they are the “same” house. The layout of the rooms and other crucial characteristics are the same in each. To create a different house, we would need a different blueprint.

A class is a blueprint of an object. It establishes the kind of data an object of that type will hold and defines the methods that represent the behavior of such objects. However, a class is not an object any more than a blueprint is a house. In general, a class contains no space to store data. Each object has space for its own data, which is why each object can have its own state.

Once a class has been defined, multiple objects can be created from that class. For example, once we define a class to represent the concept of a bank account, we can create multiple objects that represent specific, individual bank accounts. Each bank account object would keep track of its own balance.

Key Concept
A class is a blueprint of an object. Multiple objects can be created from one class definition.

An object should be *encapsulated*, which means it protects and manages its own information. That is, an object should be self-governing. The only changes made to the state of the object should be accomplished by that object’s methods. We should design objects so that other objects cannot “reach in” and change their states.

Classes can be created from other classes by using *inheritance*. That is, the definition of one class can be based on another class that already exists. Inheritance is a form of *software reuse*, capitalizing on the similarities between various kinds of classes that we may want to create. One class can be used to derive several new classes. Derived classes can then be used to derive even more classes. This creates a hierarchy of classes, where the attributes and methods defined in one class are inherited by its children, which in turn pass them on to their children, and so on. For example, we might create a hierarchy of classes that represent various types of accounts. Common characteristics are defined in high-level classes, and specific differences are defined in derived classes.

Polymorphism is the idea that we can refer to multiple types of related objects over time in consistent ways. It gives us the ability to design powerful and elegant solutions to problems that deal with multiple objects.

Some of the core object-oriented concepts are depicted in **Figure 1.23**. We don’t expect you to understand these ideas fully at this point. Most of this book is designed to flesh out these ideas. This overview is intended only to set the stage.

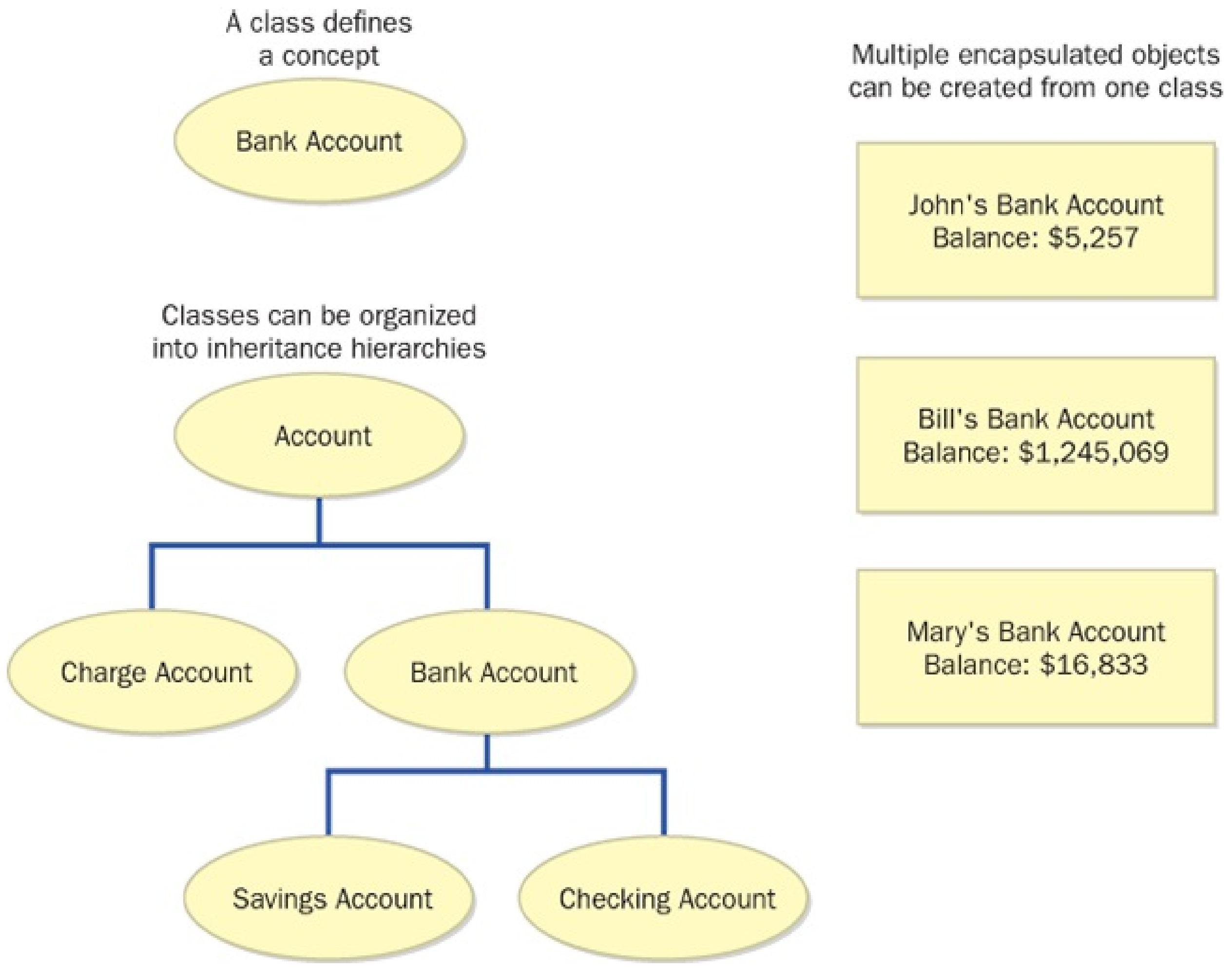

A class defines a concept
Bank Account
Multiple encapsulated objects can be created from one class
John's Bank Account Balance: $5,257
Bill's Bank Account Balance: $1,245,069
Mary's Bank Account Balance: $16,833
Classes can be organized into inheritance hierarchies
Account
Charge Account
Bank Account
Savings Account
Checking Account

Figure 1.23 Various aspects of object-oriented software

Self-Review Questions

(see answers in **Appendix L**)

SR 1.33 List the five general steps required to solve a problem.
SR 1.34 Why is it important to consider more than one approach to solving a problem? Why is it important to consider alternatives early in the process of solving a problem?

SR 1.35 What are the primary concepts that support object-oriented programming?

Summary of Key Concepts

- A computer system consists of hardware and software that work in concert to help us solve problems.
- The CPU reads the program instructions from main memory, executing them one at a time until the program ends.
- The operating system provides a user interface and manages computer resources.
- As far as the user is concerned, the interface *is* the program.
- Digital computers store information by breaking it into pieces and representing each piece as a number.
- Binary is used to store and move information in a computer because the devices that store and manipulate binary data are inexpensive and reliable.
- There are exactly 2^N permutations of N bits. Therefore, N bits can represent up to 2^N unique items.
- The core of a computer is made up of main memory, which stores - programs and data, and the CPU, which executes program instructions one at a time.
- An address is a unique number associated with a memory location.
- Main memory is volatile, meaning the stored information is maintained only as long as electric power is supplied.
- The surface of a CD has both smooth areas and small pits. A pit represents a binary 1 and a smooth area represents a binary 0.
- The fetch–decode–execute cycle forms the foundation of computer processing.

- A network consists of two or more computers connected together so that they can exchange information.
- Sharing a communication line creates delays, but it is cost effective and simplifies adding new computers to the network.
- A local-area network (LAN) is an effective way to share information and resources throughout an organization.
- The Internet is a wide-area network (WAN) that spans the globe.
- Every computer connected to the Internet has an IP address that uniquely identifies it.
- The World Wide Web is software that makes sharing information across a network easy for humans.
- A URL uniquely specifies documents and other information found on the Web for a browser to obtain and display.
- This book focuses on the principles of object-oriented programming.
- Comments do not affect a program’s processing; instead, they serve to facilitate human comprehension.
- Inline documentation should provide insight into your code. It should not be ambiguous or belabor the obvious.
- Java is case sensitive. The uppercase and lowercase versions of a letter are distinct.
- Identifier names should be descriptive and readable.
- Appropriate use of white space makes a program easier to read and understand.
- You should adhere to a set of guidelines that establish the way you format and document your programs.
- All programs must be translated to a particular CPU’s machine language in order to be executed.

- High-level languages allow a programmer to ignore the underlying details of machine language.
- A Java compiler translates Java source code into Java bytecode, a low-level, architecture-neutral representation of the program.
- Many different development environments exist to help you create and modify Java programs.
- Syntax rules dictate the form of a program. Semantics dictate the meaning of the program statements.
- The programmer is responsible for the accuracy and reliability of a program.
- A Java program must be syntactically correct or the compiler will not produce bytecode.
- Object-oriented programming helps us solve problems, which is the purpose of writing a program.
- Program design involves breaking a solution down into manageable pieces.
- Each object has a state, defined by its attributes, and a set of behaviors, defined by its methods.
- A class is a blueprint of an object. Multiple objects can be created from one class definition.

Exercises

EX 1.1 Describe the hardware components of your personal computer or of a computer in a lab to which you have access. Include the processor type and speed, storage capacities of main and secondary memory, and types of I/O devices. Explain how you determined your answers.

EX 1.2 Why do we use the binary number system to store information on a computer?

EX 1.3 How many unique items can be represented with each of the following?

a. 1 bit
b. 3 bits
c. 6 bits
d. 8 bits
e. 10 bits
f. 16 bits

EX 1.4 If a picture is made up of 128 possible colors, how many bits would be needed to store each pixel of the picture? Why?

EX 1.5 If a language uses 240 unique letters and symbols, how many bits would be needed to store each character of a document? Why?

EX 1.6 How many bits are there in each of the following? How many bytes are there in each?

a. 12 KB

b. 5 MB
c. 3 GB
d. 2 TB

EX 1.7 Explain the difference between random access memory (RAM) and read-only memory (ROM).
EX 1.8 A disk is a random access device but it is not RAM (random access memory). Explain.
EX 1.9 Determine how your computer, or a computer in a lab to which you have access, is connected to others across a network. Is it linked to the Internet? Draw a diagram to show the basic connections in your environment.
EX 1.10 Explain the differences between a local-area network (LAN) and a wide-area network (WAN). What is the relationship between them?
EX 1.11 What is the total number of communication lines needed for a fully connected point-to-point network of eight computers? Nine computers? Ten computers? What is a general formula for determining this result?
EX 1.12 Explain the difference between the Internet and the World Wide Web.
EX 1.13 List and explain the parts of the URLs for

a. your school
b. the Computer Science department of your school
c. your instructor’s Web page

EX 1.14 Give examples of the two types of Java comments and explain the differences between them.

EX 1.15 Which of the following are not valid Java identifiers? Why?

a. `Factorial`
b. `anExtremelyLongIdentifierIfYouAskMe`
c. `2ndLevel`
d. `level2`
e. `MAX_SIZE`
f. `highest$`
g. `hook&ladder`

EX 1.16 Why are the following valid Java identifiers not considered good identifiers?

a. `q`
b. `totVal`
c. `theNextValueInTheList`

EX 1.17 Java is case sensitive. What does that mean?

EX 1.18 What is a Java Virtual Machine? Explain its role.

EX 1.19 What do we mean when we say that the English language is ambiguous? Give two examples of English ambiguity (other than the example used in this chapter) and explain the ambiguity. Why is ambiguity a problem for programming languages?

EX 1.20 Categorize each of the following situations as a compile-time error, run-time error, or logical error.

a. multiplying two numbers when you meant to add them
b. dividing by zero

c. forgetting a semicolon at the end of a programming statement
d. spelling a word incorrectly in the output
e. producing inaccurate results
f. typing a `{` when you should have typed a `(`

Programming Projects

PP 1.1 Enter, compile, and run the following program:

```
public class Test
{
   public static void main(String[] args)
   {
      System.out.println("An Emergency Broadcast");
   }
}
```

VideoNote Developing a solution for **PP 1.2**.

PP 1.2 Introduce the following errors, one at a time, to the program from **PP 1.1**. Record any error messages that the compiler produces. Fix the previous error each time before you introduce a new one. If no error messages are produced, explain why. Try to predict what will happen before you make each change.

a. change `Test` to `test`
b. change `Emergency` to `emergency`
c. remove the first quotation mark in the string
d. remove the last quotation mark in the string
e. change `main` to `man`
f. change `println` to `bogus`
g. remove the semicolon at the end of the `println` statement
h. remove the last brace in the program

PP 1.3 Write a program that prints, on separate lines, your name, your birthday, your hobbies, your favorite book, and your favorite movie. Label each piece of information in the output.

PP 1.4 Write a program that prints a list of four or five Web sites that you enjoy. Print both the site name and the URL.

PP 1.5 Write a program that prints the first few verses of a song (your choice). Label the chorus.

PP 1.6 Write a program that prints the outline of a tree using asterisk (*) characters.

PP 1.7 Write a program that prints a paragraph from a novel of your choice.

PP 1.8 Write a program that prints the phrase `Knowledge is Power`:

a. on one line
b. on three lines, one word per line, with the words centered relative to each other

c. inside a box made up of the characters = and |

PP 1.9 Write a program that prints the following diamond shape. Don’t print any unneeded characters. (That is, don’t make any character string longer than it has to be.)

```
     *
    ***
   *****
  *******
 *********
  *******
   *****
    ***
     *
```

PP 1.10 Write a program that displays your initials in large block letters. Make each large letter out of the corresponding regular character. For example:

```
JJJJJJJJJJJJJJJ    AAAAAAAAA    LLLL
JJJJJJJJJJJJJJJ  AAAAAAAAAAA    LLLL
        JJJJ     AAA     AAA    LLLL
        JJJJ     AAA     AAA    LLLL
        JJJJ     AAAAAAAAAAA    LLLL
J       JJJJ     AAAAAAAAAAA    LLLL
JJ      JJJJ     AAA     AAA    LLLL
 JJJJJJJJJJJ     AAA     AAA    LLLLLLLLLLLLLL
```

JJJJJJJJJ   AAA   AAA   LLLLLLLLLLLLLL