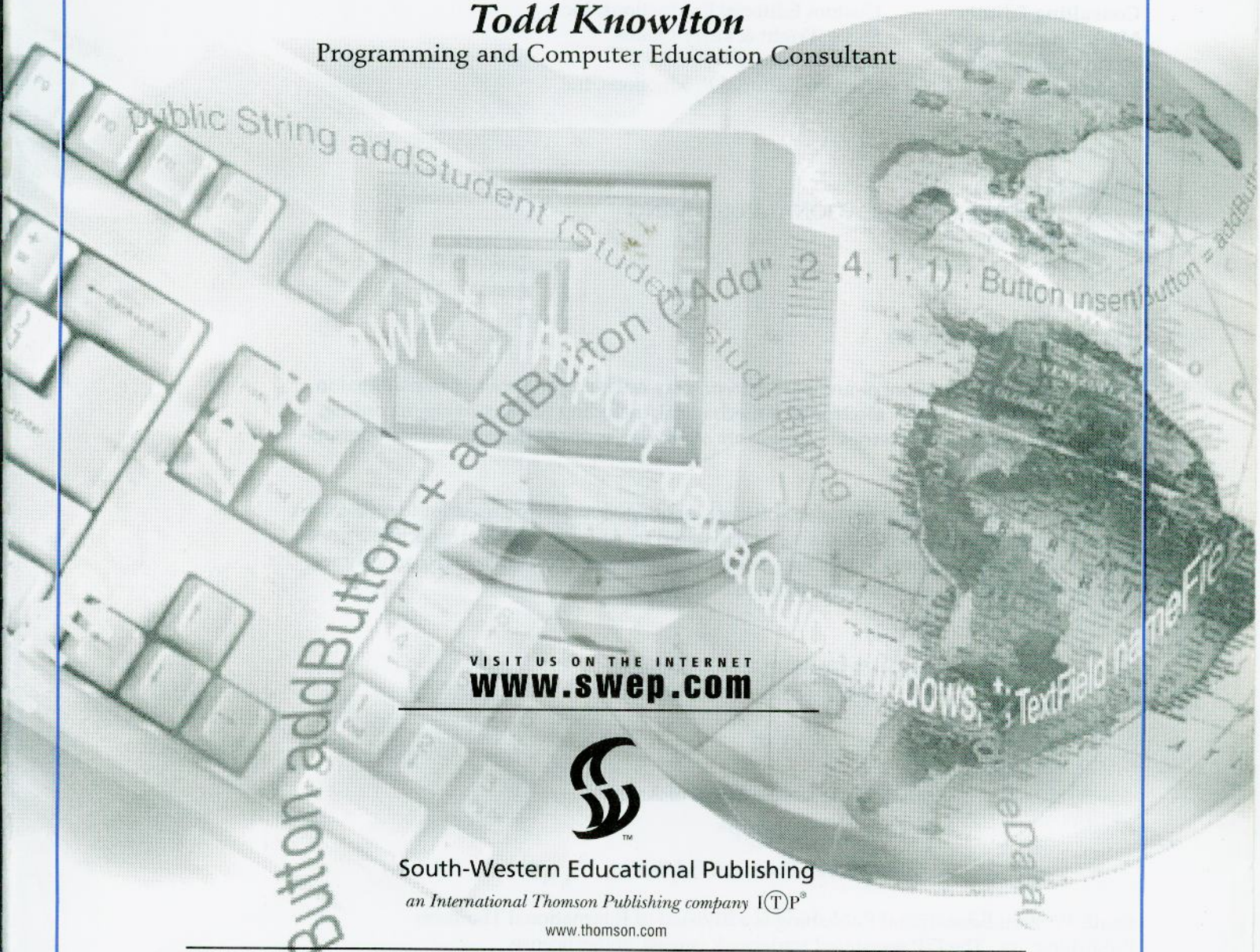
JAVA™ Introduction to Programming

Todd Knowlton
Programming and Computer Education Consultant



VISIT US ON THE INTERNET
www.swep.com

South-Western Educational Publishing
an International Thomson Publishing company ITP®
www.thomson.com

Cincinnati • Albany, NY • Belmont, CA • Bonn • Boston • Detroit • Johannesburg • London • Madrid
Melbourne • Mexico City • New York • Paris • Singapore • Tokyo • Toronto • Washington

1 Computers and Programming

OBJECTIVES

- *List the differences between specific-purpose and general-purpose computers.*
- *Recite the four tasks of a computer.*
- *Illustrate how data and instructions are represented inside the computer.*
- *Explain the binary number system.*
- *Tell the differences between high- and low-level computer languages and the advantages of each.*
- *Define the roles of assemblers, interpreters, and compilers.*
- *Explain the process of compiling Java source code.*

Overview

No one has to tell you what a computer is. You see and use them every day. Computers have revolutionized the way we work and play. And the revolution isn't over yet. Both large and small businesses use computers for everything from word processing and spreadsheets to payroll and inventory. Computers are used at home for personal finance, correspondence, education, entertainment, and more.

One of the most exciting uses for computers is to communicate and share information over the worldwide network known as the Internet. Businesses and individuals alike are finding ways that the Internet can help them, enlighten them, and entertain them.

Although computers were once very expensive to make, advances in manufacturing have made computers affordable for almost anyone today. And these advances have also helped to make computers a part of many products we buy: cars, televisions, and even toasters. Some computers are made to carry out a specific task, whereas other computers are made to be programmed for a variety of tasks.

In this chapter, you will learn about computers that perform specific tasks and computers that perform more general tasks. You will also learn how computers are programmed, and you will be introduced to programming languages, including Java.

Note

This chapter assumes that you have experience using the Internet and can use a Web browser such as Internet Explorer or Netscape Navigator.

CHAPTER 1, SECTION 1

A Machine with a Purpose

Even though computers are fascinating, they are just machines—machines designed by people. There are thousands of different computers, and each is designed with a purpose in mind. Some computers, like the one in Figure 1-1, are powerful, number-crunching, music-playing, graphic-displaying, data-storing workhorses. These computers can be programmed to do bookkeeping, produce documents, play games, assist in music composition, track important data, and more. Other computers are created for much more specific purposes, such as monitoring a security system.

FIGURE 1-1
Computers like this are capable of many different tasks. *(Courtesy of IBM Corporation)*

COMPUTERS FOR SPECIFIC PURPOSES

Computers are found in our wristwatches, cameras, televisions, and VCRs. In automobiles, computers control fuel injection and the spark plugs. Computers monitor everything from fuel efficiency to the comfort of the passengers. Computers tell automatic transmissions when to shift gears. Computers can also help you quickly change from one radio station to another. These are known as specific-purpose computers.

Specific-purpose computers are used solely for what they are designed to accomplish. For example, the computer in a camera calculates the settings and exposure time required for a perfect photograph, but it can't help you with your math homework. The computer in your VCR will remember to record your favorite show Tuesday night at 8 P.M., but it can't remind you that you have a project due tomorrow.

GENERAL-PURPOSE COMPUTERS

The kind of machine most people think of when they hear the term *computer* looks something like Figure 1-2. A computer like this one is a general-purpose computer and can be programmed to perform many different tasks. It can do word processing, Internet access, and spreadsheets—even at the same time. What makes this computer system so popular is that it can perform a wide variety of tasks.

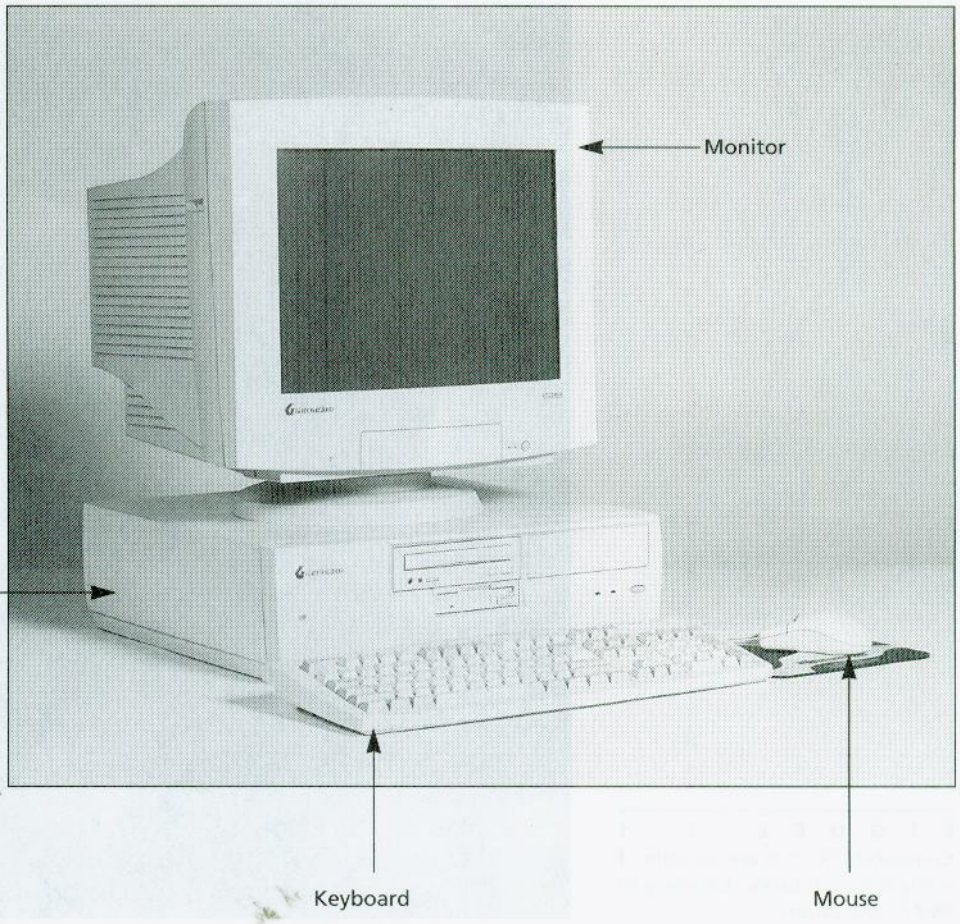


FIGURE 1-2
A typical microcomputer system includes a monitor, keyboard, mouse, and system unit. *(Courtesy of Gateway 2000)*

SYSTEM COMPONENTS

The physical equipment that makes up a computer is called ***hardware.*** Each piece of hardware performs one of four tasks: input, output, processing, or storage.

INPUT AND OUTPUT

All computers, whether on your wrist or on your desk, *interact* with someone or something. Interaction involves getting information and then giving a response. In a computer, this interaction is called ***input*** and ***output.*** For example, input could be a user entering customer names and addresses into a database program. An example of output would be printing mailing labels from a database.

A desktop computer interacts primarily with people. It interacts using a keyboard, a mouse, speakers, a monitor, and a printer. Some desktop computers may also have other input and output devices, such as a microphone or modem. A ***modem,*** which is a device that allows interaction to occur between computers using a telephone line, is capable of both input and output.

The keyboard, mouse, and microphone are input devices. The computer uses these devices to get information and instructions from the person using the computer (the user). The computer gives information back to the user via output devices, such as the monitor, printer, and speakers.

When you think of computer input and output, you may think of data going in and answers coming out. For example, a program might receive the temperature in Fahrenheit as input and then convert it to provide the temperature in Celsius as output. But even a game has input and output. The keys you strike or the movement of a mouse or joystick is the input. The image on the monitor and sound through the speakers is the output.

You may not have thought about speakers as an output device, but they have become an important way for the computer to give information to the user. Early computers were incapable of sophisticated sound output. They could only beep. Now multimedia computers speak, play music, and offer sounds that add realism to games.

PROCESSING AND STORAGE

A computer processes and stores input, and it generates output. This is accomplished using a variety of devices such as RAM, ROM, a microprocessor, a bus, and disk drives.

RAM, which is an acronym for *random access memory*, is your computer's primary storage. RAM is where currently running programs and active data are stored. Some RAM is also used to store items that support your program and its input and output. For example, when you are using your word processor, the word processor program is stored in RAM along with the document upon which you are working and the programs that allow it to print on your specific printer.

RAM is sometimes referred to as ***primary storage.*** RAM is electronic and requires a constant supply of electricity to store the data. Because data stored in RAM is lost if power is interrupted, RAM is called ***volatile storage.***

ROM is an acronym for *read-only memory*. Your computer's ROM is a set of memory chips that have data permanently stored upon them. Typically, ROM chips store data and programs necessary to get the computer started and to handle the basic functions of the computer.

The microprocessor does the actual processing and also controls everything else in the computer. All of the other parts of the computer in turn support the microprocessor. As soon as you turn on your computer, the microprocessor begins performing millions of commands every second.

The microprocessor is connected to the RAM, ROM, and other devices by a system of wires called a *bus*. In most computers, the wires that make up the bus are not actually individual wires, but lines etched on a circuit board.

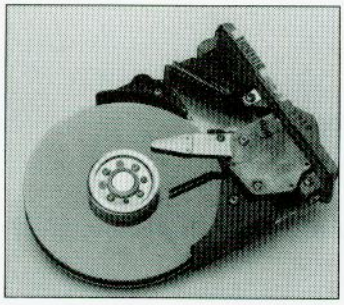


FIGURE 1-3
A hard disk drive is an example of secondary storage.

Data and programs that are not currently in RAM need to be stored in more permanent storage, which is not affected when the power is turned off. In the computer, this more permanent "filing cabinet" storage is called ***secondary storage.*** Secondary storage usually comes in the form of disks. Programs and data stored on disks remain stored when the power is off. The ***hard disk*** installed in your computer is an example of secondary storage. Figure 1-3 shows a hard disk drive with its top removed. A hard disk is sealed to prevent dust from affecting its sensitive surface.

A ***floppy disk*** is another example of secondary storage. A floppy disk uses a thin, flexible disk on which to store the data. Both hard disks and floppy disks store data by placing magnetic fields on the surface of the disk—the same principle used by audio tapes and videotapes. The data stored on hard or floppy disks, by means of magnetic fields, remains when the power is turned off.

SECTION 1.1 QUESTIONS

TRUE/FALSE

T F 1. A computer in a VCR is an example of a general-purpose computer.

T F 2. The equipment that makes up a computer is called hardware.

T F 3. A speaker is an input device.

T F 4. Output is the response a computer gives to input.

T F 5. A modem is both an input and an output device.

T F 6. RAM is an acronym for *read anywhere memory*.

T F 7. ROM is an acronym for *read-only memory*.

T F 8. The bus connects the microprocessor to the RAM, ROM, and other devices.

T F 9. A hard disk is an example of primary storage.

T F 10. Hard disks and floppy disks store data magnetically.

SHORT ANSWER

1. Name an item other than one mentioned in this chapter that includes a specific-purpose computer.
2. What are the four tasks that hardware performs?
3. List two devices used for input in a general-purpose computer.
4. List three devices involved in output from a general-purpose computer.
5. List two devices involved in either processing or storage in a general-purpose computer.
6. What is typically stored in RAM?
7. What type of chip has data permanently stored on it?
8. Why is RAM called volatile?
9. What happens to the data on a hard disk when electric power is discontinued?
10. Why are hard disks sealed in a case?

CHAPTER 1, SECTION 2

The Computer's Language

Computers are complex machines. But unless they are given instructions, all that great hardware is wasted. In order to be useful, computers must be programmed.

Compared to the computers of today, the first computers were simple devices. They were programmed by flipping switches or by inserting cards with holes punched in them. Early computers were used only to do simple math or tabulation. Today's computers, however, are used to display pictures, play sounds, and perform very complex tasks. To make the programming of modern computers possible, more sophisticated languages had to be developed. However, regardless of how the computer is programmed, the data and the instructions directing how the data is to be processed are represented by simple electronic circuits.

In this section, you will see how the circuits in a computer are used to represent data and give instructions.

REPRESENTING DATA

Data is a computer representation of something that exists in the real world. For example, data can be values, such as money, measurements, quantities, or a high score. Data can also be alphabetic, such as names and addresses or a business letter.

In a computer, all data is represented by numbers, and the numbers are represented electronically in the computer. To understand how electrical signals become numbers, let's begin by looking at a simple electric circuit that everyone is familiar with: a switch controlling a light bulb.

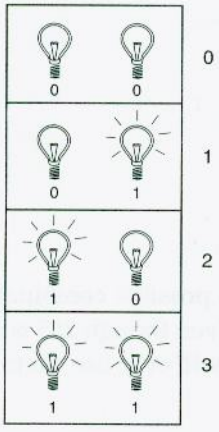


FIGURE 1-4
There are four light combinations possible with two light bulbs.

FROM CIRCUITS TO NUMBERS

When you think of an electric circuit, you probably think of it being either on or off; for example, a light bulb is turned on and off by a switch. The light bulb can exist in two conditions: on or off. In technical terms, the one light bulb has two *states*.

Imagine you had two light bulbs on two switches. With two light bulbs, there are four possible states, as shown in Figure 1-4. Now suppose you assigned a number to each of those states. If both light bulbs were off, it would represent the number 0. If the first light bulb was off and the second was on, it would represent the number 1. You could assign a number to each of the four states and represent the four numbers 0, 1, 2, and 3.

You can't do much using only the numbers 0 through 3, but if more light bulbs are added, the number of states increases. For example, Figure 1-5 shows how three light bulbs can represent the numbers 0 through 7 because there are eight possible states.

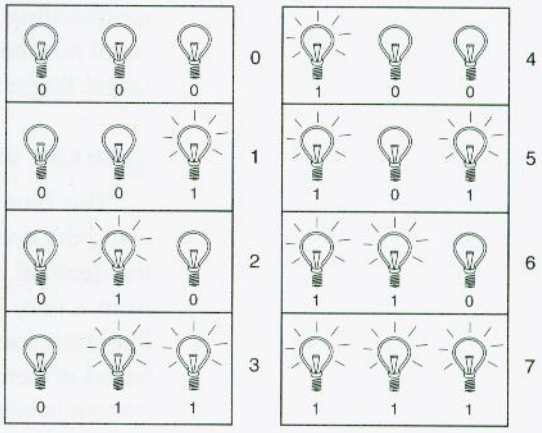


FIGURE 1-5
There are eight light combinations possible with three light bulbs.

If you are the mathematical type, you may have noticed that the number of states is determined by the formula 2^n where n is the number of light bulbs (see Table 1-1).

NUMBER OF CIRCUITS	NUMBER OF STATES	NUMBERS THAT CAN BE REPRESENTED
1	$2^1 = 2$	0,1
2	$2^2 = 4$	0..3
3	$2^3 = 8$	0..7
4	$2^4 = 16$	0..15
5	$2^5 = 32$	0..31
6	$2^6 = 64$	0..63
7	$2^7 = 128$	0..127
8	$2^8 = 256$	0..255

TABLE 1-1

Now instead of light bulbs, think about circuits in the computer. A single circuit in a computer is like a single light bulb; it can be on or off. A special number system, called the ***binary number system,*** is used to represent numbers with groups of these circuits. In the binary number system, each binary digit (or bit) is either a 0 or a 1. As shown in Figure 1-6, circuits that are off are defined as 0, and circuits that are on are defined as 1. Binary digits (***bits***) are combined into groups of eight bits called ***bytes.***

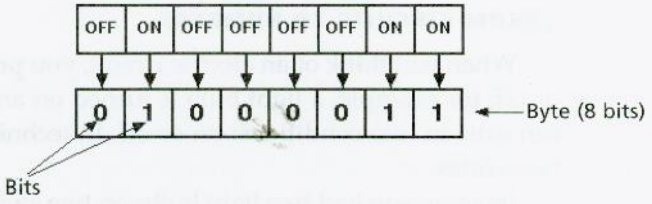


FIGURE 1-6
In the computer, signals that are off are defined as 0 and signals that are on are defined as 1.

If a byte is made up of eight bits, then there are 256 possible combinations representing the numbers 0 through 255 (see Table 1-1). Even though 255 is not a small number, it is definitely not the largest number you will ever use. So to represent larger numbers, computers group bytes together.

BINARY VS. DECIMAL

The binary number system may seem strange to you because you count using the ***decimal number system,*** which uses the digits 0 through 9. Counting in the decimal number system comes very naturally to you because you learned it from a very young age. But someone invented the decimal number system just like someone invented the binary number system. The decimal number system is based on tens because you have ten fingers on your hands. The binary number system is based on twos because of the circuits in a computer. *Both systems, however, can be used to represent the same values.*

In the decimal number system, each digit of a number represents a power of 10. That is why the decimal number system is also called the base 10 number system. Consider the number 3208, for example. When you read that number, you automatically understand it to mean three thousands, two hundreds, no tens, and eight ones. Represented mathematically, you could say $(3 \times 1000) + (2 \times 100) + (0 \times 10) + (8 \times 1) = 3208$, as shown in Figure 1-7.

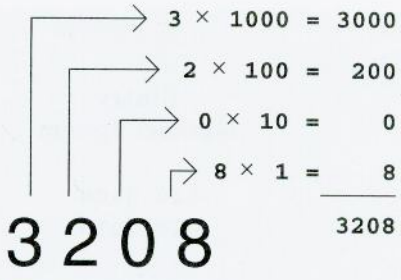


FIGURE 1-7
Each digit of the decimal number 3208 represents a power of 10.

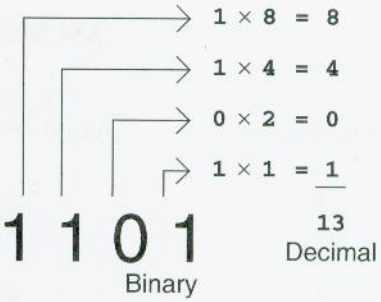


FIGURE 1-8
Each digit of the binary number 1101 represents a power of 2, so conversion to the decimal system is easy.

In the binary number system, each digit represents a power of 2, as you saw in Table 1-1. Working with powers of 2 is not as natural to you as working with powers of 10. But with a little practice you will see that base 2 numbers are not so mysterious. Consider the binary number 1101. Even though the number is four digits long, its value is nowhere near a thousand. Remember that the powers of 2 are 1, 2, 4, 8, 16, 32, and so on. So for this number, its decimal equivalent is $(1 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = 13$, as shown in Figure 1-8. So the binary number 1101 is equivalent to 13 in the decimal number system.

Extra for Experts

Decimal Points and Binary Points

You have used decimal points for a long time. A decimal point divides the ones place and the tenths place, or 10^0 from 10^{-1}. Did you know there is a binary point? The equivalent to the decimal point in the binary number system is called the binary point. It divides the 2^0 place from the 2^{-1} place.

With a binary point, it is possible to have binary numbers such as 100.1, which in decimal is 4.5. Can you convert the binary number 10.01 to decimal? If you got 2.25 as the answer, you are correct. Try converting the binary number 11.001001 to decimal.

NUMBERS REPRESENTING LETTERS AND SYMBOLS

If all data in a computer is represented by numbers, how are letters and symbols stored? Letters and symbols, called ***characters***, are assigned a number that the computer uses to represent them. Most computers assign numbers to characters according to the ***American Standard Code for Information Interchange (ASCII).*** Figure 1-9 shows some of the ASCII (pronounced *ask-e*) codes. For a complete ASCII table, see Appendix A.

The basic ASCII code is based on 7 bits, which gives 128 characters. About 95 of these are upper- and lowercase letters, numbers, and symbols. Some of the characters are used as codes for controlling communication hardware and other devices. Others are invisible characters such as Tab and Return. Most computers extend the ASCII code to 8 bits (a whole byte) to represent 256 characters. The additional 128 characters are used for graphical characters and characters used with foreign languages.

ASCII CODES

Character	Decimal Number System	Binary Number System
$	36	010 0100
*	42	010 1010
A	65	100 0001
B	66	100 0010
C	67	100 0011
D	68	100 0100
a	97	110 0001
b	98	110 0010
c	99	110 0011
d	100	110 0100

FIGURE 1-9
In the computer, each character is stored as a number.

Some languages, including Java, use an extension of the ASCII code called *Unicode*. Unicode is a 16-bit standard that defines over 65,000 special characters and international symbols. The first 128 characters of Unicode, however, are the same as ASCII.

REPRESENTING INSTRUCTIONS

You have seen how computers use bits to represent data. We use computers to do much more than represent data for storage. Computers are useful because they follow instructions to do work. And just like data, instructions are represented by combinations of bits.

Recall that the microprocessor is the device in which instructions are executed. Each instruction, which consists of ones and zeros, is called ***machine language.***

Writing a computer program in machine language would be very difficult because even a simple program requires hundreds or even thousands of microprocessor instructions. Another problem is that the numbers used to represent microprocessor instructions are difficult for people to understand. Figure 1-10 shows a short machine language program. Each line is one instruction for the microprocessor. To find out what the program does, you would have to look up the machine language instructions in the microprocessor's reference manual.

Analog vs. Digital

The early computers used gears, wheels, or other mechanical devices to represent numbers. These are called analog devices. An ***analog*** *device uses quantities that are variable or exist in a range. For example, a second hand on a clock is an analog device because it represents a value with a continuously variable quantity.*

Electronics have made digital devices the basis for computers. A ***digital*** *device uses switches (or digits) in combination to represent something in the real world. For example, rather than have a second hand rotate at a fixed speed to represent a value, a digital clock counts the seconds electronically.*

```
01010101
10001011 11101100
01001100
01001100
01010110
01010111
10111111 00000011 00000000
10111110 00000010 00000000
10001011 11000111
00000011 11000110
10001001 01000110 11111110
01011111
01011110
10001011 11100101
01011110
11000011
```

FIGURE 1-10
Machine language is the language of the microprocessor. This machine language program adds 3 + 2 and stores the result.

SECTION 1.2 QUESTIONS

TRUE/FALSE

T F 1. Names, addresses, and dollar amounts are all examples of data.

T F 2. The decimal number system is also called the base 2 number system.

T F 3. Each digit of a binary number represents a power of 10.

T F 4. Letters and symbols are represented in the computer by numbers.

T F 5. The basic ASCII code uses 8 bits.

SHORT ANSWER

1. Define data.
2. How many bits are in a byte?
3. What is the language called that is "understood" by the microprocessor?
4. How many combinations of bits are possible with three bits?
5. How many combinations of bits are possible with five bits?

PROBLEM 1.2.1

Convert the following numbers from binary to decimal:

Binary	Decimal	Binary	Decimal
0000	_____	1000	_____
0001	_____	1110	_____
0111	_____	1111	_____

PROBLEM 1.2.2

Complete the table below by converting the decimal numbers on the left to binary. Hint: Remember, each digit in a binary number represents a power of 2 (8, 4, 2, 1).

Decimal	Binary
0	0000
1	_____
2	_____
3	_____
4	_____
5	_____
6	_____
7	_____
8	_____
9	_____
10	_____

PROBLEM 1.2.3

Fill in the missing powers of 2 in the series below.

1, 2, 4, 8, _____, _____, _____, _____, _____, _____, _____,

_____, 4096.

PROBLEM 1.2.4

Looking at the chart in Appendix B, add the decimal values of the ASCII characters that spell your first name. Remember to use a capital letter where necessary. Write each character, its decimal equivalent, and the sum of all the ASCII values.

CHAPTER 1, SECTION 3

Programming Languages

Supplying computers with instructions would be extremely difficult if machine language were the only option available to programmers. Fortunately, special languages have been developed that are more easily understood. These languages, called ***programming languages,*** provide a way to program computers using instructions that can be understood by both computers and people.

Like human languages, programming languages have their own vocabulary and rules of usage. Some programming languages are very technical, and others are made to be as similar to English as possible. The programming languages available today allow programming at many levels of complexity.

ASSEMBLY LANGUAGE

The programming language most like machine language is assembly language. ***Assembly language*** uses letters and numbers to represent machine language instructions (see Figure 1-11). However, assembly language is still difficult to read.

Machine Language	Assembly Language
01010101	PUSH BP
10001011 11101100	MOV BP,SP
01001100	DEC SP
01001100	DEC SP
01010110	PUSH SI
01010111	PUSH DI
10111111 00000011 00000000	MOV DI, 0003
10111110 00000010 00000000	MOV SI, 0002
10001011 11000111	MOV AX, DI
00000011 11000110	ADD AX, SI
10001001 01000110 11111110	MOV [BP-02], AX
01011111	POP DI
01011110	POP SI
10001011 11100101	MOV SP, BP
01011110	POP BP
11000011	RET

FIGURE 1-11
In assembly language, each microprocessor instruction is assigned a code that makes the program more meaningful to programmers.

Assembly language programming requires using an assembler. An ***assembler*** is a separate program that reads the codes the programmer has written and creates or "assembles" a machine language program based on those codes.

LOW-LEVEL VS. HIGH-LEVEL LANGUAGES

Machine language and assembly language are called ***low-level languages.*** In a low-level language, it is necessary for the programmer to know the instruction set of the microprocessor in order to program the computer. Each instruction in a

low-level language corresponds to one or only a few microprocessor instructions. In the program in Figure 1-11, each assembly language instruction corresponds to one machine language instruction.

Today almost all programming is done in ***high-level languages.*** In a high-level language, instructions do not necessarily correspond one-to-one with the instruction set of the microprocessor. One command in a high-level language may represent many microprocessor instructions. Therefore, high-level languages reduce the number of instructions that must be written by a programmer. A program that might take hours to write in a low-level language can be done in minutes in a high-level language. Programming in a high-level language also reduces the number of errors because the programmer doesn't have to write as many instructions and the instructions are easier to read. Figure 1-12 shows a program written in four popular high-level languages. Like the machine language and assembly language programs you saw earlier, these high-level programs add the numbers 3 and 2 together.

BASIC

```
10 I = 3
20 J = 2
30 K = I + J
```

Pascal

```
program AddIt;

var
  i, j, k : integer;

begin
  i := 3;
  j := 2;
  k := i + j;
end.
```

C++

```
main()
 {
   int i,j,k;
   i = 3;
   j = 2;
   k = i + j;
   return 0;
 }
```

Java

```
public class AddIt
{
    public static void
    main(String args[])
   {
  int i, j, k;
  i = 3;
  j = 2;
  k = i + j;
   }
}
```

FIGURE 1-12
The same program can be written in more than one high-level language.

Programs written in a high-level language are also easier to move among computers with different microprocessors. For example, the microprocessors in Apple Macintosh computers use a different instruction set than the microprocessors in PC-compatible computers. An assembly language program written for a PC computer will not work on a Macintosh. However, a simple program written in a high-level language can work on both computers with little or no modification.

So why use a low-level language? It depends on what you need to do. The drawback of high-level languages is that they do not always provide a command for everything the programmer needs a program to do. Using assembly language, the programmer can write instructions that enable the computer to do anything the hardware will allow.

Another advantage of low-level languages is that a program written in a low-level language will generally require less memory and run more quickly than the same program written in a high-level language. This is because high-level languages must be translated into machine language before the microprocessor can execute the instructions. The translation is done by another program and is usually less efficient than the work of a skilled assembly-language programmer. Table 1-2 summarizes the advantages of low- and high-level languages.

ADVANTAGES OF LOW-LEVEL LANGUAGES	ADVANTAGES OF HIGH-LEVEL LANGUAGES
Better use of hardware's capabilities	Requires less programming
Requires less memory	Fewer programming errors
Runs more quickly	Easier to move between computers with different microprocessors
	More easily read

TABLE 1-2

INTRODUCING JAVA

In this book you will learn to program in a high-level language called *Java*. Java is similar to some other programming languages, but it is also unique in many ways.

Java was originally developed to program specific-purpose computers like those mentioned earlier in this chapter. Although it may have been developed for programming blenders and toaster ovens, Java has found a home in general-purpose computers as well—especially on the Internet.

The same features that make Java ideal for programming a pager make Java ideal for the Internet. Java can be used to create tiny and efficient programs. When programming a consumer electronic product or appliance, having a program that requires very little space is important. It is also important on the Internet because Java programs, called ***applets,*** can travel quickly over the Internet and execute on the user's computer.

Web browsers such as later versions of Internet Explorer and Netscape Navigator are able to execute Java applets included as part of Web pages. For example, a Web page might use a Java applet to scroll text across the screen.

EXERCISE 1-1 WATCH JAVA GO

1. Open your Web browser. In the address line, key **http://www.studio-jplus.com/programjava**, and press **Enter**. If you have a Java-enabled browser, you should see text scrolling across your screen once the page loads.
2. Click the link **Other Cool Java Samples** and watch the demonstration.
3. Use the **Back** button to return to the previous page.
4. Click the **Testing Center** link.
5. Choose the link for the **Sample Test**. The test that appears on your screen is actually a Java applet.
6. When finished taking the sample test, close your browser.

INTERPRETERS AND COMPILERS

Programmers writing in a high-level language enter the program's instructions into a text editor. A ***text editor*** is similar to a word processor, except the files are saved in ASCII format without the font and formatting codes word processors use. The files saved by text editors are called ***text files.*** A program in the form of a high-level language is called ***source code.***

Programmers must have their high-level programs translated into the machine language the microprocessor understands. The translation may be done by interpreters or compilers. The resulting machine language code is known as ***object code.***

INTERPRETERS

An ***interpreter*** is a program that translates the source code of a high-level language into machine language. An interpreter translates a computer language in a way similar to the way a person might interpret between languages such as English and Spanish. Each instruction is interpreted from the programming language into machine language as the instructions are needed. Interpreters are normally used only with very high-level languages. Many versions of BASIC as well as some database programming languages are interpreted.

To run a program written in an interpreted language, you must first load the interpreter into the computer's RAM memory. Then you load the program to be interpreted. The interpreter steps through the program one instruction at a time and translates each instruction into machine language, which is then sent to the microprocessor. Every time the program is run, the interpreter must once again translate each instruction.

Because of the need to have the interpreter in memory before the program can be interpreted, interpreted languages are not widely used to write programs that are sold. The buyer of the program would have to have the correct interpreter in order to use the program.

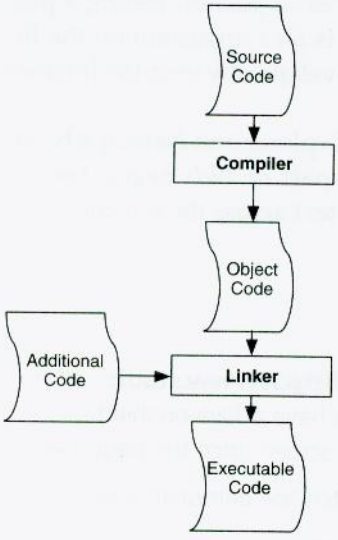


FIGURE 1-13 Compiling a program involves a compiler and a linker.

COMPILERS

A ***compiler*** is another program that translates a high-level language into machine language. A compiler, however, makes the translation once, then saves the machine language so that the instructions do not have to be translated each time the program is run. Programming languages such as Pascal and C++ use compilers rather than interpreters.

Figure 1-13 shows the steps involved in using a compiler. First, the source code is translated using the compiler to a file called an ***object file.*** An object file, however, is incomplete. A program called a ***linker*** must also be used. The linker combines the object file with other machine language instructions necessary to create a program that can run without an interpreter. The linker produces an ***executable file*** that can be run as many times as desired without further need for translating.

Although using a compiler involves more steps than using an interpreter, most compilers automate the task and make it easy for the programmer to use. For example, most compilers allow you to compile and link in a single operation.

Programs you use regularly, such as word processors and games, are examples of programs written with a compiler. Compiled programs require less memory than interpreted programs because a compiled program does not require that an interpreter be loaded into memory. Compiled programs also run faster than interpreted programs because the translation has already been done. When a compiled program is run, the program is loaded into memory in the machine language the microprocessor needs.

COMPILING JAVA

Java adds a twist to traditional compiling. If someone ever asks you whether Java is a compiled or interpreted language, the answer is both.

In most languages, it is the job of the compiler to create machine code that will execute on a specific machine. For example, a C++ compiler running on a computer with the Windows 95 operating system works with the linker to create a program that will execute on a computer running Windows 95. If you take that program to a Macintosh or some other type of computer or operating system, the program will not run. Even though the same C++ source code may compile on a Macintosh compiler and on a Windows compiler with no modification, the executable programs produced by the two compilers are very different and not interchangeable.

Java compilers do not create machine code that is specific to a certain computer or operating system. Instead, a Java compiler creates a small program (or applet) that is made up of ***byte codes.*** Byte codes are instructions that are created to be understood by an imaginary computer called the ***Java virtual machine*** (JVM). That's where the interpreter comes in.

The Java virtual machine doesn't actually exist. However, any computer can be programmed to act like the Java virtual machine. Any computer can execute the Java byte codes if the computer can emulate the Java virtual machine. In other words, the computer running the Java program uses an interpreter to execute the byte codes. This Java byte code interpreter is usually part of a Web browser.

When you load a Web page that includes a Java applet, the applet is loaded into your computer's memory and the Web browser interprets the Java byte codes and runs the Java applet. Because the Java byte codes are not created for any specific computer, any computer with a Web browser that can interpret the Java byte codes can run the applet.

Figure 1-14 shows the steps that code goes through in a Java compiler and how the byte codes are interpreted by the Web browser.

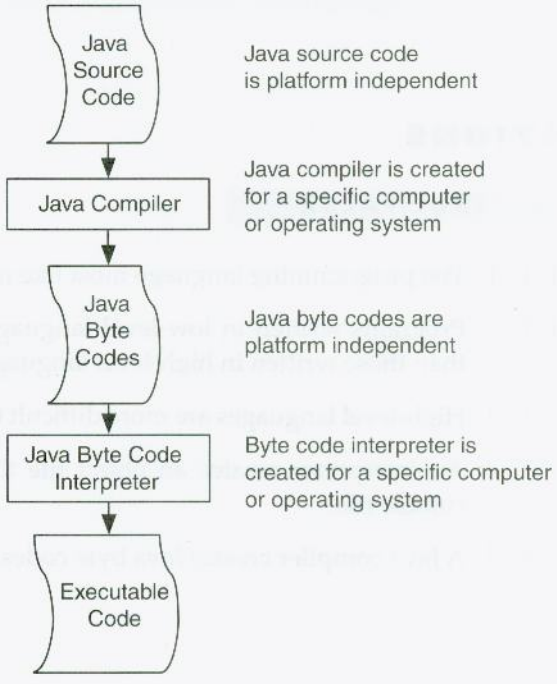


FIGURE 1-14 Java source code must go through these steps to become executable.

Categories of Software

There are two basic categories of software: application software and system software. Application software is the software that performs the tasks you want the computer to perform. For example, application software is what you use to produce a document with a word processor or balance your checkbook. System software coordinates the interaction of the hardware devices, controls the input and output, and loads application software into memory so that you can run the programs you want to run. The system software required to run your computer is most often packaged together and called an ***operating system.***

You may be asking, "Why would you want to compile a Java program if the results of the compilation still have to be interpreted? Why not just have the Web browser interpret the Java source code?" The answer is that Java byte codes can be interpreted more efficiently than pure source code. By compiling the Java source code into Java byte codes, you get some of the benefit of a compiled program while maintaining the ability to interpret the byte codes on any computer that supports the Java virtual machine. It is a trade-off of speed for compatibility.

Extra for Experts

Just-in-Time Compilation

Interpreting Java byte codes in a browser normally causes a slight decrease in performance. A byte code program does not execute as fast as a normal compiled program. To make Java applets run faster, some browsers support a feature called Just-in-Time Compilation, or JIT. A browser that is running with JIT enabled converts the byte codes for a Java program into native code and saves it as a file on disk. The browser then executes the native program code instead of the Java byte codes, resulting in increased performance of the program.

SECTION 1.3 QUESTIONS

TRUE/FALSE

T F 1. The programming language most like machine language is C++.

T F 2. Programs written in low-level languages usually require less memory than those written in high-level languages.

T F 3. High-level languages are more difficult to read than low-level languages.

T F 4. An interpreter creates an object file that a linker makes into an executable file.

T F 5. A Java compiler creates Java byte codes.

SHORT ANSWER

1. Give an example of a low-level programming language.
2. List three examples of high-level programming languages.
3. Describe one advantage that compiled programs have over interpreted programs.
4. Explain why Java can run on any computer that supports the Java virtual machine.
5. Java programs are often run from what kind of application?

KEY TERMS

American Standard Code for Information Interchange (ASCII)
analog
applet
assembler
assembly language
binary number system
bit
bus
byte
byte codes
characters
compiler
data
decimal number system
digital
executable file
floppy disk
hard disk
hardware
high-level language
input
interact
interpreter
Java
Java virtual machine
linker
low-level language
machine language
modem
object code
object file
operating system
output
primary storage
programming language
RAM
ROM
secondary storage
source code
states
text editor
text file
Unicode
volatile storage

SUMMARY

- Today, computers are everywhere. Some computers are designed to perform specific tasks and some are designed to be programmable general-purpose computers.
- The equipment that makes up a computer is called hardware. Each piece of hardware is involved in input, output, processing, or storage.
- RAM is the computer's primary storage for currently running programs and current data. ROM is memory that has data permanently stored on it. Hard disks and floppy disks are secondary storage.
- The microprocessor does the computing and controls everything else that is going on in the computer.
- Inside a computer, signals called bits represent data and give instructions. Bits are commonly arranged in groups of eight, called bytes. The microprocessor responds to commands called machine language.
- High-level programming languages allow programmers to work in a language that people can more easily read. Machine language and assembly language are low-level languages because each instruction in the language corresponds to one or only a few microprocessor instructions. In high-level languages, instructions may represent many microprocessor instructions.
- An interpreter or compiler must translate high-level languages into machine language. An interpreter translates each program step into machine language as the program runs. A compiler translates the program before it is run and saves the machine language as an object file. A linker then creates an executable file from the object file.
- Java is both compiled and interpreted. A compiler translates Java source code into byte codes. Java byte codes are not compiled for a specific computer or operating system. Instead they are compiled to be understood by an imaginary computer called the Java virtual machine. The byte codes are then interpreted by the Java virtual machine.

PROJECTS

PROJECT 1-1

Choose some large value, such as the salary of your favorite professional athlete, and convert it to the binary number system.

PROJECT 1-2

Make a chart of at least six high-level languages. Include a brief description of each language that tells the primary use of the language or its historical significance. If you can find the date the language was created, include that on your chart. Some languages to consider are Ada, ALGOL, BASIC, C, C++, COBOL, FORTRAN, Java, LISP, Logo, Oberon, Pascal, PL/I, and Smalltalk.

2 Integrating Applets into Web Pages

OBJECTIVES

- *Recite the history of the Java language.*
- *Explain how Java applets are delivered over the Internet from Web servers to client computers.*
- *Discuss HTML and the common tags.*
- *Use the <APPLET> tag to embed Java applets into Web pages.*
- *Control Java applets by passing parameters.*

Overview

Java is a programming language with an interesting history. In this chapter, you will learn about the history of the development of the Java language and why it is so widely used on the World Wide Web. You will also learn how Java applets are integrated into Web pages and learn how to control the behavior of a Java applet.

CHAPTER 2, SECTION 1

More about Java

Java was not created with the World Wide Web in mind. The language was originally developed for consumer electronic devices. It wasn't until later that Java found its most comfortable home on the Internet. In this section, you will get a glimpse at the history of Java and learn how Java applets are delivered over the Internet and run on a Web browser.

JAVA HISTORY

In the early 1990s, a company called Sun Microsystems created a team of top software developers to study consumer electronic devices to see if they could be programmed in ways similar to general-purpose computers. The team, code-named Green, discussed and studied current technologies. They took apart several different electronic devices, such as remote controls, cable boxes, gaming devices, TVs, and stereos. The team observed that the devices were not compatible with each other. To address this issue, the team began development of a new programming language they called ***Oak***.

Oak was based on a language called C++ but was stripped down to be more compatible with the small electronic devices. However, Oak failed to generate excitement among electronics manufacturers, and it fell on hard times. The Green team pursued several different avenues to develop a niche for this language. Unfortunately, none of their attempts to establish the language in the electronics market met with much success.

It also turned out that another company was using the name *Oak* for a different technology. So a new name had to be found. The story is that after a trip to a coffee shop, the team decided to change the name of the language to Java.

Just about the time the team was giving up on Java, the idea was born to use Java in conjunction with the World Wide Web. Java's ability to produce small yet functional programs was ideal for the Internet. Most World Wide Web users connect over a telephone line using a modem, which is much slower than the processing speed of the computer itself. Thus, short programs that transmit rapidly are very important. And Java programs by nature are very short and concise. Also working in Java's favor was the fact that Java programs do not compile for a specific type of computer. By enabling a Web browser to interpret the Java byte codes, any computer on the Internet can run the same Java programs. Web browsers written for Windows, OS/2, Unix, or the MacOS can run the same Java byte codes.

Although Java has been mostly used as a way to add more interesting and dynamic content to WWW pages, it has evolved into a computing platform on which programmers can develop full-scale applications.

SERVING UP JAVA

Let's take a closer look at how a Java applet is served up over the Internet. In Figure 2-1, a computer called a Web server stores a Java applet on its hard disk. A *Web server* is a computer that delivers Web pages to computers running Web browsers. The computers running the Web browsers are called ***client computers.*** The Web server typically has a full-time connection to the Internet so that it can be available 24 hours a day. For example, the Web pages you saw at the address *www.studio-jplus.com* are on a Web server in Texas. In the previous chapter, when you browsed those pages, your computer was acting as a client computer.

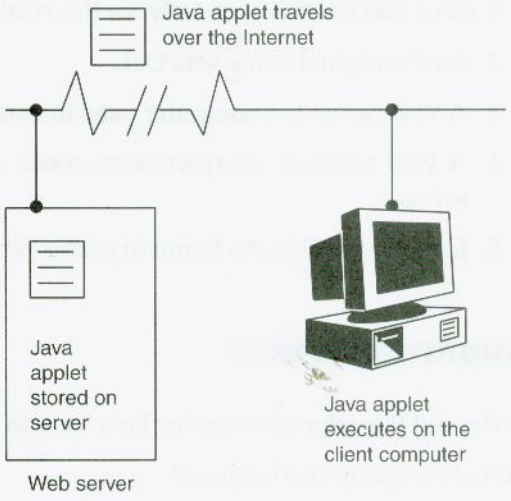


FIGURE 2-1 Java applets are stored on a Web server, which delivers the applet to the Web browsers running on the client computers.

The Java applet on the Web server is sent to a client computer as part of a Web page. The program does not run on the server. Once the entire applet is delivered, the Web browser begins interpreting the byte codes. What you see on your screen is actually a program running on your computer.

Note

Because the Java applet runs on the client computer, Java has built-in security to prevent the Java applet from doing harm to the client computer. For example, the built-in security prevents an applet from erasing files or implanting a virus on your computer.

EXERCISE 2-1

BROWSING A PAGE WITH AN APPLET

1. Start your Web browser. In the address line, key **http://www.studio-jplus.com/programjava/chapter02** and press **Enter**. Watch the status bar at the bottom of your browser as the page loads. You might catch a glimpse of the Java applet loading and beginning to execute.

2. Move your mouse pointer over the word "Java" at the top of the page and watch how the letters change color. The Java applet senses the position of your mouse pointer and changes the color of the letter to which you are pointing.
3. After you have demonstrated the applet on your screen, close your browser.

Java applets can be run in a Web browser or from any other Java byte code interpreter. Java compilers typically include an interpreter to allow you to run Java programs from within your Java programming environment. In the next chapter, you will use your Java compiler's interpreter.

SECTION 2.1 QUESTIONS

TRUE/FALSE

T F 1. Java was created especially for the World Wide Web.

T F 2. Java's original name was Oak.

T F 3. A Web server is a computer used to surf the Web with a Web browser.

T F 4. A Web browser interprets byte codes as they are transferred from the server.

T F 5. Java interpreters are found in places other than Web browsers.

SHORT ANSWER

1. Why did the original name for Java have to be changed?
2. What company created Java?
3. Identify two features of Java that make it ideal for use on the World Wide Web.
4. What is the function of a Web server?
5. What is a client computer?

PROBLEM 2.1.1

Search the Internet for information about Java and the history of Java. Use a search engine or investigate the **www.studio-jplus.com** site for links. Write a short report based on what you find.

PROBLEM 2.1.2

Search the Internet for galleries of sample Java applets. Search for interesting uses for Java, such as chat rooms that use Java applets. After seeing some of what Java is being used for, write a brief report about how Java is being used and how it might be used in the future.

CHAPTER 2, SECTION 2

Applets and Web Pages

In this and the previous chapter, you used a World Wide Web browser to view an HTML document that contained a Java applet. You may be wondering how that Java applet got into that Web page. In this section, you will learn how to include Java applets in Web pages using the <APPLET> tag.

Note

This section assumes you have some experience creating Web pages and are familiar with the basics of HTML. Only a quick review of HTML is provided in this section.

HTML AND HTTP

HTML (Hypertext Markup Language) is the programming language used to create World Wide Web documents. HTML uses codes called *tags* to specify the structure of a document. Some tags specify titles and formatting for paragraphs. Other tags allow links to other HTML documents. ***Hypertext Transfer Protocol (HTTP)*** is the protocol that Web servers use to transfer HTML documents and other kinds of files to your Web browser.

You may be wondering how HTML compares to what is actually being displayed on the screen. Figure 2-2 shows a World Wide Web page and Figure 2-3 shows the HTML that created that page.

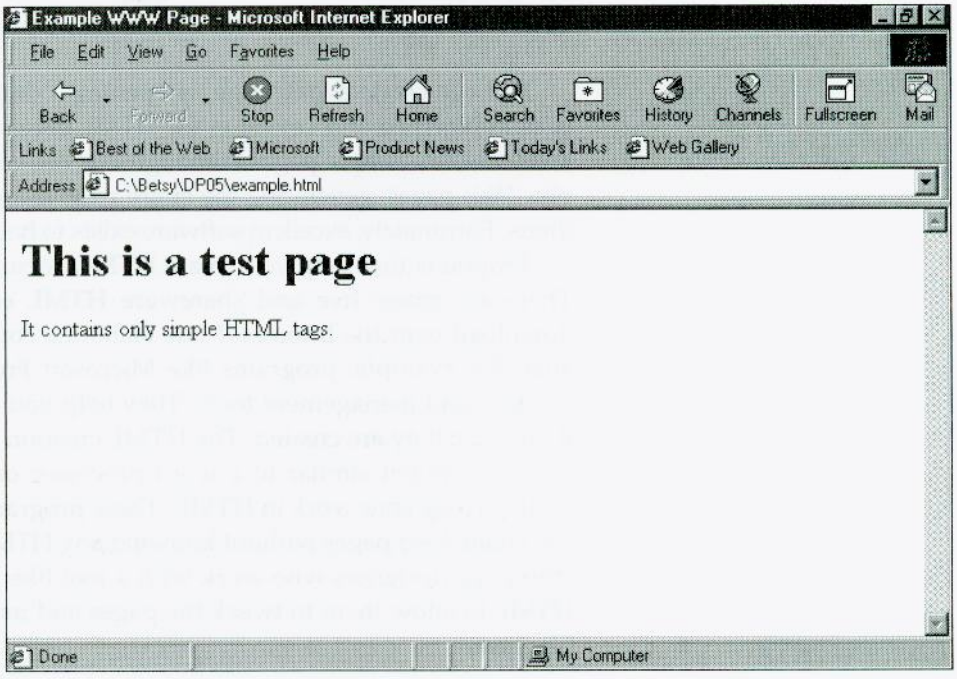


FIGURE 2-2
A Web browser translates the HTML document into the image you see on the screen.

```
<HTML>
<HEAD>
<TITLE>Example WWW Page</TITLE>
</HEAD>
<BODY>
</BODY>
<H1>This is a test page</H1>
<P>It contains only simple HTML tags.
</HTML>
```

FIGURE 2-3
The HTML source for the page shown in Figure 2-2 is filled with special codes called tags.

Table 2-1 is a summary of the common HTML tags. If you are not familiar with all of the tags, take a moment to learn what tags are available.

HTML QUICK REFERENCE

Structure Tags	
`<! ... >`	creates a comment
`<HTML> ... </HTML>`	begins and ends an HTML document
`<HEAD> ... </HEAD>`	begins and ends header
`<BODY> ... </BODY>`	begins and ends body
Titles	
`<TITLE> ... </TITLE>`	page title, must be in header, appears at top of screen
Headings	
`<H1> ... </H1>`	headings 1 through 6; 1 uses the largest font size, and
`<H6> ... </H6>`	6 uses the smallest font size; and
Paragraphs	
`<P> ... </P>`	
Character Formatting	
`<B> ... </B>`	bold
`<I> ... </I>`	italic
`<CODE> ... </CODE>`	code sample (normally Courier font)

TABLE 2-1

You can create a Web page by opening a simple text document in a program such as Notepad and keying in the HTML necessary to generate the page. This is acceptable for simple pages (although it is not very convenient). With more complex Web pages, generating the entire page from scratch by keying HTML is tedious. Fortunately, excellent software exists to help you create Web pages quickly.

Programs that help you create HTML documents are called HTML editors. There are many free and shareware HTML editors available that you may download over the Internet. More elaborate commercial software is also available. For example, programs like Microsoft FrontPage are complete Web site creation and management tools. They help you create Web pages and maintain them once they are created. The HTML creation programs allow you to work in an environment similar to a word processor or desktop publishing program, while saving your work in HTML. These programs are so easy that some people can create Web pages without knowing any HTML. However, most professional Web page designers who work with a tool like FrontPage know enough about HTML to allow them to tweak the pages and make the best use of technologies like Java.

EXERCISE 2-2 CREATING AN HTML DOCUMENT

1. Open a text editor such as Notepad.
2. Enter the HTML code below into the document.

```
<HTML>
<HEAD>
<TITLE>Example WWW Page</TITLE>
</HEAD>
<BODY>
</BODY>
<H1>This is a test page</H1>
<P>It contains only simple HTML tags.
</HTML>
```

3. Save the document as **mine.htm** or **mine.html** to your student disk or folder. Depending on your operating system, you may use the html filename extension or the htm extension. In many cases, either one is acceptable.
4. Close the text editor.
5. Open your Web browser.
6. In the address (or URL) field, enter **file://** followed by the path and filename of the document you just saved. For example, if you saved the file on a floppy disk in the A: drive of your computer, enter **file://a:\mine.htm.**
7. If you made any errors in the HTML, open the document with the text editor, make any necessary corrections, and view it again.
8. When the document displays correctly, close your Web browser.

THE <APPLET> TAG

Table 2-1 includes many popular HTML tags that are used to format Web pages. Java applets are put into Web pages using a special tag called the *<APPLET> tag.* The **<APPLET>** tag tells the Web browser that an applet is embedded in the page. In order to tell the applet where to appear and how to act, the **<APPLET>** tag includes information called *attributes.* Three of these attributes are required and others are optional.

Figure 2-4 shows the HTML source code for a simple page that includes an applet. The **<APPLET>** tag in the figure contains only the required attributes **CODE**, **WIDTH**, and **HEIGHT**. Note that, like many other tags, the **<APPLET>** tag is paired with a closing tag (**</APPLET>**).

The **CODE** attribute specifies what Java applet is to be executed. The name of the file is used to identify the applet. Java applets carry the .class extension.

Note

The applet file is not Java source code. A Java compiler has already translated the Java source code into byte codes. The file with the .class extension is a file of byte codes.

```
<HTML>
<HEAD>
<TITLE>A Simple Web Page with an Applet</TITLE>
</HEAD>
<BODY>
</BODY>
<H1>A WWW Page with a Java applet</H1>
<HR>
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
</APPLET>
</HTML>
```

FIGURE 2-4
An **<APPLET>** tag is included in the HTML source.

The **WIDTH** and **HEIGHT** attributes tell the Web browser how much horizontal and vertical space, in pixels, to reserve in the page for the applet. Depending on the applet, these settings may not be necessary. However, in cases where the Java applet occupies a portion of the screen, these settings may be used to size the area devoted to the applet.

The **<APPLET>** tag can also contain several other attributes: **ALIGN**, **HSPACE**, **VSPACE**, and **CODEBASE**. Let's examine these one at a time.

ALIGN is used to align the applet with other elements of the page. Table 2-2 describes the possible values for the **ALIGN** attribute.

LEFT	The applet is placed at the left margin.
RIGHT	The applet is placed at the right margin.
BASELINE	The bottom of the applet is aligned with the baseline of the text in the line in which the applet appears. The baseline of text is the line on which the text rests. Only descenders such as the bottom of a lowercase *y* extend below the baseline.
ABSMIDDLE	The middle of the applet is aligned with the middle of the largest item on the line on which the applet appears.
MIDDLE	The middle of the applet is aligned with the middle of the baseline of the text in the line in which the applet appears.
ABSBOTTOM	The bottom of the applet is aligned with the lowest item in the line.
BOTTOM	Same as BASELINE alignment.
TEXTTOP	The top of the applet is aligned with the top of the tallest text in the line.
TOP	The top of the applet is aligned with the tallest item in the line. The tallest item could be text or another applet or image.

TABLE 2-2

The **HSPACE** and **VSPACE** attributes indicate the amount of horizontal and vertical space, in pixels, that a Web browser should put between the applet and any surrounding items. This is convenient if there is text or graphics near the space the applet will occupy. Using **HSPACE** and **VSPACE** ensures that the surrounding items do not crowd your applet.

The **CODEBASE** attribute is used when the executable applet is stored on the server in a directory different from the current Web page. You might use an applet

in a variety of Web pages. Rather than store a copy of the applet in the directory of each Web page, you can specify the location of the applet.

EXERCISE 2-3 VIEWING THE HTML SOURCE OF A PAGE WITH APPLETS

1. Start your Web browser. In the address line, key **http://www.studio-jplus.com/programjava/chapter02/applets.html** and press **Enter.**
2. Look at the page and observe the different applets.
3. Use your Web browser's command for viewing the HTML source code. If you are using Internet Explorer, choose **Source** from the **View** menu. If you are using Netscape Navigator, choose **Document Source** or **Page Source** from the **View** menu. Notice how the first **<APPLET>** tag contains only the required attributes, and the others contain additional attribute tags.
4. Close the window displaying the HTML source code.
5. Close the Web browser.

Now that you have learned how the **<APPLET>** tag is used to embed an applet into a Web page, and you have seen examples of HTML source that include applets, let's create a simple Web page with an applet embedded in it.

EXERCISE 2-4 CREATING WEB PAGES WITH APPLETS

1. Open a text editor such as Notepad.
2. Enter the HTML code below into the document.

```
<HTML>
<HEAD>
<TITLE>A Simple Web Page with an Applet</TITLE>
</HEAD>
<BODY>
</BODY>
<H1>A WWW Page with a Java applet</H1>
<HR>
<APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
</APPLET>
</HTML>
```

3. Save the document to your student disk or folder as **hello.htm** or **hello.html**. In future exercises and problems, HTML files will be assumed to have an html extension. If your system can use only three-character extensions, use .htm instead.
4. Copy the applet named HelloWorld.class from the student data files that accompany this book to the same disk or directory where the HTML document is saved.
5. Use your Web browser to view **hello.html** and verify that the Java applet was properly included.
6. Experiment with some of the other **<APPLET>** attributes.
7. Close the Web browser.

WARNING

You may find that changing the attributes in the HTML source file and then viewing the page again does not produce the changes you were expecting. In some cases, the Web browser will continue to run the applet with the same attributes as the first time it was run, even though the attributes have changed. The Refresh or Reload button may not be enough in some cases to make your changes take effect. You may have to exit the browser and restart it to cause the browser to apply your new attributes.

SECTION 2.2 QUESTIONS

TRUE/FALSE

T F 1. HTTP is the programming language used to create Web documents.

T F 2. Codes called tags are used to specify the structure of a Web document.

T F 3. Programs that help you create Web pages are called HTML editors.

T F 4. The **<APPLET>** tag tells the Web browser that an applet is embedded in the Web page.

T F 5. The **CODEBASE** attribute is used to place space around an applet.

SHORT ANSWER

1. What does the acronym HTML stand for?
2. What tags begin and end an HTML document?
3. What are the three required attributes of the **<APPLET>** tag?
4. What is the function of the **CODE** attribute?
5. What units of measurement do the **WIDTH** and **HEIGHT** attributes use?

PROBLEM 2.2.1

Write an HTML source file that will display your name at the top of the page. Below your name, embed the Java applet named **JAVAMouseOver.class** from the student data files. Use a width of 200 pixels and a height of 150 pixels. Load your page into a Web browser to test it. Save the HTML file as **Problem221.html.**

PROBLEM 2.2.2

Modify the page you created in Problem 2.2.1 to place the applet on the same line with your name and align the middle of the Java applet with the middle of the baseline of the text. Save the HTML file as **Problem222.html.**

CHAPTER 2, SECTION 3

Passing Parameters to Applets

Many Java applets are programmed to allow themselves to be customized by the author of the Web page. This customization occurs by passing information called *parameters* to the applet. In this section, you will learn how to customize an applet by passing parameters.

AN EXAMPLE OF A CUSTOMIZABLE APPLET

Many applets are customizable. Parameters can be used to set colors, modify the behavior of the applet, or provide an applet with information to be displayed. Only the programmer of the applet limits the possibilities. Let's look at an example of a customizable applet. In this case, the applet creates a scrolling marquee on the screen. Among other things, a parameter is used to specify the text to be displayed.

EXERCISE 2-5 A LOOK AT A CUSTOMIZABLE APPLET

1. Copy the **ScrollText.class** and the **param.html** files from the student data files to your student disk or folder. Open your Web browser. Rather than going to the World Wide Web for the document, open the **param.html** file from your student disk or folder. The document includes a Java applet that makes text scroll across the screen.
2. Leave the browser open for the next exercise.

SPECIFYING PARAMETERS

To specify parameters, use the *<PARAM> tag* as shown in Figure 2-5. An opening **<APPLET>** tag begins the applet section, and a closing **</APPLET>** tag

```
<APPLET CODE="ScrollText.class"
        ALIGN="baseline" WIDTH="605" HEIGHT="30"
        NAME="ScrollText">
<PARAM NAME="ScrollingText"
       VALUE="Scrolling Message Goes Here">
<PARAM NAME="InitDir" VALUE="right">
<PARAM NAME="FinalDir" VALUE="left">
<PARAM NAME="TextColor" VALUE="blue">
<PARAM NAME="BgrdColor" VALUE="yellow">
<PARAM NAME="FontSize" VALUE="18">
<PARAM NAME="FontType" VALUE="Times New Roman">
<PARAM NAME="ScrollSpeed" VALUE="5">
<PARAM NAME="FadeSpeed" VALUE="6">
<PARAM NAME="xSize" VALUE="605">
<PARAM NAME="ySize" VALUE="30">
</APPLET>
```

FIGURE 2-5
Parameters are specified with the **<PARAM>** tag.

closes the section. Between the opening and closing of the applet, parameters can be specified.

In order to customize an applet, you must know what customization the applet will accept. The parameter names and values are not necessarily standardized. The programmer of the applet can name the parameters anything he or she desires and can set the acceptable values. Before you pass a parameter to an applet, you have to know the name of the available parameters as well as the valid values.

The programmer of the applet should document the available parameters. If you do not have access to any documentation for the applet you are using, you can sometimes learn what parameters are available by looking at the HTML source of pages that customize the applet.

EXERCISE 2-6 VIEWING THE PARAMETERS

1. View the source of the page from Exercise 2-5 currently on your screen.
2. Locate the **<PARAM>** tags and see if you can determine what each parameter does. Which parameter is responsible for providing the message for the marquee?
3. Close the HTML source and the Web browser.

Notice that the **<PARAM>** tag has the syntax **<PARAM NAME="name" VALUE="*value*">**, where *name* is the parameter name and *value* is the value of that particular parameter. Parameter values are enclosed in double quotes.

Note

In a **<PARAM>** *tag, the parameter name is case sensitive and the value must be placed in quotes. The term* case sensitive *means that the capitalization of the characters in the* **<PARAM>** *tag must match the capitalization used by the Java programmer.*

EXERCISE 2-7 RUNNING AN APPLET WITH MULTIPLE PARAMETERS

1. Open a text editor, and then open **param.html.**
2. Modify the **ScrollingText** parameter to display a message of your choice.
3. Modify the parameters that specify colors. The color values accepted by the applet are white, lightGray, darkGray, black, red, pink, orange, yellow, green, magenta, cyan, and blue.
4. Save the edited HTML source.
5. Open **param.html** with your Web browser and observe the effect of your parameters.
6. Close the Web browser.

ALTERNATE HTML

Unfortunately, some Web browsers do not support Java applets. Web browsers without Java support can still view your page, but the applets will not

display. To avoid leaving people who do not have Java-capable browsers completely in the dark, there is a special area where you can place HTML code that will display only on Web browsers that do not support Java. HTML code placed in this area is known as *alternate HTML.*

The code in Figure 2-6 is an example of HTML that includes alternate HTML for an applet.

```
<HTML>
<HEAD>
<TITLE>Java Sample</TITLE>
</HEAD>
<BODY>
<APPLET CODE="COOL.class" WIDTH=600 HEIGHT=450>
Your web browser does not support Java applets.<BR>
</APPLET>
</BODY>
</HTML>
```

FIGURE 2-6
Alternate HTML is used to display a message or alternate information when viewed on a Web browser not capable of interpreting Java.

Alternate HTML can be more complex than just a message. Any valid HTML code can appear in the alternate HTML area. For example, if your Java applet provides menu choices to the user, the alternate HTML could be used to provide those same choices to the user using a non-Java method.

THE FOUR MAIN PARTS OF AN <APPLET> TAG

To review, the **<APPLET>** tag has four main parts:

1. The opening **<APPLET>** tag, which includes the applet attributes.
2. The optional applet parameters.
3. The optional alternate HTML.
4. The closing **</APPLET>** tag.

SECTION 2.3 QUESTIONS

TRUE/FALSE

T F 1. All Java applets are customizable.

T F 2. Java applets are customized by passing parameters to the applet.

T F 3. The programmer of the applet decides the names of parameters and acceptable values.

T F 4. The **<OPTION>** tag is used to pass parameters.

T F 5. Parameter names are case sensitive.

SHORT ANSWER

1. Identify two common uses for applet parameters.
2. Which tag closes an applet section?

3. What is one way you might learn what parameters an applet accepts?
4. The value of a parameter must be enclosed in what kind of character?
5. Write a line of HTML code that passes a parameter named TextColor with the value green.

PROBLEM 2.3.1

Open the file **param.html** from the student data files. Modify the file to occupy a **WIDTH** of 400 and a **HEIGHT** of 60. Set the FontSize parameter to 36, the xSize to 400, and the ySize to 60. Save the modified source as **param2.html** to your student disk or folder. Then, load the page into your Web browser to observe the changes.

PROBLEM 2.3.2

Modify **param2.html** to include the following message in the alternate HTML section:

This browser does not support Java applets.

Save the modified source as **param3.html** to your student disk or folder.

KEY TERMS

alternate HTML
<APPLET> tag
case sensitive
client computers
HTML (Hypertext Markup Language)
HTTP (Hypertext Transfer Protocol)
Oak
parameters
<PARAM> tag
tags
Web server

SUMMARY

- Java was originally named Oak and was created for use with consumer electronic devices.
- Java became a popular language for use with the Internet because of the small size of programs and the fact that Java programs do not compile for any specific computer.
- Java applets are delivered over the Internet from a Web server to a client computer running a Web browser. The Web browser interprets the Java byte codes and the Java applet executes on the client computer.
- HTML (Hypertext Markup Language) is the programming language used to create World Wide Web documents. Within the HTML document are tags, which specify the structure of the document.

- Web servers use the Hypertext Transfer Protocol (HTTP) to transfer HTML documents and other kinds of files to your Web browser.
- The **<APPLET>** tag is used to embed Java applets into HTML documents. A variety of attributes are available as part of the **<APPLET>** tag.
- Parameters may be passed to an applet to modify its behavior. The parameters are passed using the **<PARAM>** tag.
- For Web browsers that do not support Java, you can include alternate HTML. Alternate HTML is displayed only in cases where the Web browser is not Java-capable.

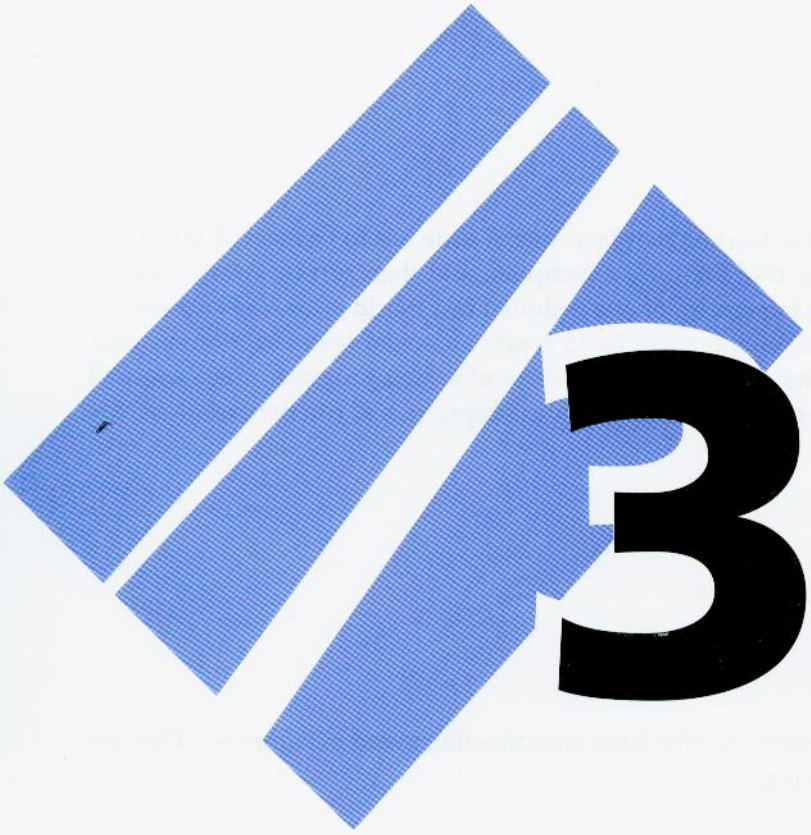
PROJECTS

PROJECT 2-1

1. Start your Web browser. In the address line, key **http://www.studio-jplus.com/ programjava/chapter02/proj2-1.html** and then press **Enter.**
2. Follow the instructions on the Web page for downloading the applet.
3. Print the parameter documentation.
4. Create an HTML document that includes the applet and specifies parameters as instructed on the Web page.
5. Add an alternate HTML message for browsers that do not support Java.
6. Save the program as **project2_1.html** to your student disk or folder. Then, close the document.

PROJECT 2-2

Explore the options available on your Web browser. Look for an option that prevents the Web browser from running Java applets. Set the option to disable Java and view one of the pages you created that includes alternate HTML. Finally, re-enable Java in your browser.



3 Introduction to Java Programming

OBJECTIVES

- *Explain the structure of a Java program.*
- *Access the text editor and enter Java source code.*
- *Compile and run Java programs.*
- *Describe the basic philosophy of object-oriented programming (OOP).*

Overview

You have learned that Java source code has to be entered into a text editor, translated by a compiler, and then run by either a stand-alone interpreter or one included in a World Wide Web browser.

Your task in this chapter will be to create an actual Java program on your system. You will first examine the structure of a Java program. Then you will enter a simple program into the text editor, compile it, and run the executable file that is created.

CHAPTER 3, SECTION 1

Java Program Structure

Java programs have the basic structure illustrated in Figure 3-1. They are the following:

1. **Comments.** Comments are remarks that the compiler ignores.
2. **Class Definitions.** Everything in Java is an object, and a class is the template used to create an object.
3. **Main Method.** The main method is where every Java application begins.
4. **Braces.** Braces are special characters used to mark the beginning and ending of blocks of code.
5. **Statement.** A statement is a line of Java code. Statements end with a semicolon.

Note

As you will learn later in this course, a Java program can have other methods and classes in addition to the main method and the starting class.

```
// Simple Java Program
//
// Purpose: To demonstrate the parts of a
// simple Java program.

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

FIGURE 3-1
A Java program has several parts.

Let's examine each part of a Java program in more detail.

COMMENTS

When writing a program, you may think that you will always remember what you did and why. Most programmers, however, eventually forget. But more important, others may need to make changes in a program you wrote. They probably will be unaware of what you did when you wrote the program. That is why ***comments*** are important.

Use comments to:

- explain the purpose of a program.
- keep notes regarding changes to the source code.
- store the names of programmers for future reference.
- explain the parts of your program.

Comments, which are ignored by the compiler, begin with a double slash (//) and may appear anywhere in the program. An entire line can be a comment or a comment can appear to the right of program statements, as shown in Figure 3-2. Because everything to the right of the // is ignored, do not include any statements to the right of a comment. Be sure to use the forward slash (/) rather than the backslash (\), or the compiler will try to translate your comments and give an error message.

```
int i; // declare i as an integer
```

FIGURE 3-2
Comments can follow a statement.

CLASS DEFINITION

Java is an object-oriented programming language. For now, you don't need to worry too much about object-oriented programming. However, it will help if you understand that everything in Java is an ***object.*** Remember the scrolling text applet you worked with in the previous chapter? The applet is an object—a scrolling marquee object.

An object does not exist until it is created. Thus a Java program is not an object. Rather, a Java program instructs the computer on how to create an object. The set of instructions for creating an object is called a ***class.*** Think of a class as a template or a definition that tells the compiler about an object and how to create it.

Every Java program has at least one class. This class is normally given the same name as the file that it is in. Simple programs can be written entirely within one class. Large programs are divided into several different classes. When a program includes more than one class, execution begins with the class named after the source code's filename.

You will learn much more about classes and objects in future chapters.

MAIN METHOD

A ***method*** consists of one or more statements in a class that performs a specific task within an object. For example, a circle object could have a method

written to calculate the area of a circle. That method could be used (or "called") wherever the calculation is needed in the program.

Every Java application has a ***main method*** (see Figure 3-3). The main method is run first. Just like Java classes, simple programs can be written entirely within the main method, but Java programs are typically divided into multiple methods, which are accessed through the main method.

```
class HelloWorld
{
  public static void main(String args[])       //Main Method
  {
   System.out.println("Hello World!");
  }
}
```

FIGURE 3-3
Every Java application has a main method.

The parentheses that follow the word **main** are required. They tell the compiler that **main** is a method. All methods have parentheses, although most of them have information inside the parentheses. You will learn how to write your own methods in a later chapter.

Note

All Java applications start running at the main method. However, Java applets use several different methods to take the place of the main method. You will learn about these new methods later in the book.

BRACES

Braces are used to mark the beginning and end of blocks of code. Every opening brace must have a closing brace. Notice in Figure 3-4 that the main method is enclosed in a set of braces. Providing comments after each closing brace helps to associate it with the appropriate opening brace. Also, aligning the indention of opening and closing braces improves readability.

```
public static void main(String args[])
{ // This brace marks the start of System.out block of code
  System.out.println("Hello World!");
} // This brace marks the end of System.out block of code
```

FIGURE 3-4
Braces mark the beginning and end of blocks of code.

STATEMENTS

Methods contain ***statements*** that consist of instructions or commands that make the program work. Each statement in Java ends with a semicolon.

SEMICOLONS

You must have a semicolon after every statement. The semicolon terminates the statement. In other words, it tells the compiler that the statement is complete. Notice, however, method declarations such as `main(String args[])` are exempt from being punctuated by semicolons.

JAVA AND BLANK SPACE

Java allows for great flexibility in the spacing and layout of the code. Use this feature to make it easier to read the code by indenting and grouping statements.

UPPERCASE OR LOWERCASE

Remember the ASCII codes? In the computer, an *A* is represented by a different number than an *a*. The capital letters are referred to as ***uppercase,*** and small letters are called ***lowercase.***

Java is known as case sensitive because it interprets uppercase and lowercase letters differently. For example, to a Java compiler, the word *cow* is different from the word *Cow*. Be careful to use the same combination of lettering (either uppercase or lowercase) when you enter source code. Whatever capitalization was used when the command was originally named is what must be used. Most of what you will see in Java will be in lowercase letters. If you key a command in uppercase that is supposed to be lowercase, you will get an error.

SECTION 3.1 QUESTIONS

TRUE/FALSE

T F 1. Comments begin with \\.

T F 2. Java is an object-oriented language.

T F 3. The parentheses after the word *main* indicate to the compiler that `main` is a method.

T F 4. Every opening brace must have a closing brace.

T F 5. Java statements end with a colon.

SHORT ANSWER

1. List four uses for comments.
2. What is a class?
3. What is a method?
4. What purpose do braces serve?
5. What does the term "case sensitive" mean?

From Source Code to a Finished Product

The exact process required to enter source code, compile it, and run it will vary depending on the compiler you are using. Check with your instructor to find out the specifics of compiling and running Java programs on your particular system.

ENTERING SOURCE CODE

The first step in creating a Java program is to enter your Java source code into a text file. Most Java compilers have an integrated programming environment that contains a text editor you can use. An integrated programming environment allows you to enter your source code, compile, and run while your text editor is on the screen. The integrated programming environment runs either the stand-alone interpreter or a Web browser, depending on your software.

EXERCISE 3-1 ENTERING SOURCE CODE

1. Start a new, blank file in your text editor.
2. Enter the Java source code *exactly* as it is shown below.

```
// My first Java program

class MyFirstProg
{
  public static void main(String args[])
  {
  System.out.println("My first Java program.");
  }
}
```

3. Save the file as **MyFirstProg.java** to your student disk or folder, and leave the program on your screen for the next exercise.

COMPILING AND RUNNING THE PROGRAM

Most compilers allow you to compile and run with a single command from the integrated environment.

EXERCISE 3-2 COMPILING AND RUNNING THE PROGRAM

1. Compile and run the program you entered in Exercise 3-1. If your compiler allows all of these operations to be performed with a single command, use that command. If your program fails to compile or run, check to see if you entered the code exactly as shown in Exercise 3-1 and try again.

2. If your program runs successfully, you should see the text *My first Java program* on the screen. Otherwise, ask your instructor for help.
3. Leave the source file open for the next exercise.

MAKING CHANGES AND COMPILING AGAIN

You can add, change, or delete lines from a program's source code and compile it again. The next time the program is run, the changes will be in effect.

EXERCISE 3-3 MAKING CHANGES AND COMPILING AGAIN

1. Add the statement below to the main function, substituting your name in place of Angela Askins.

```
System.out.println("By Angela Askins");
```

Your program should now appear like the one below, except your name should be on the new line.

```
// My first Java program.

class MyFirstProg
{
  public static void main("String args[]")
  {
  System.out.println("My first Java program.");
  System.out.println("By Angela Askins");
  }
}
```

2. Compile and run the program again to see the change.
3. Save and close the source code file.

Note

When executing a Java program in Microsoft's Visual J++ Interactive Development Environment (IDE), the output window may open and close so quickly that the results of executing the program may not be seen. It is possible to keep the window open to view the results of the program execution by changing a Java program to include the following statements.

1. *Modify the main method statement as follows:*

```
public static void main(String args[]) throws java.io.IOException
```

2. *Add the following statement after the last System.out.println statement*

```
System.in.read();
```

*With these two statements added, the output window will remain displayed until the **Enter** key is pressed. When the **Enter** key is pressed, the window will close.*

LOADING AND COMPILING AN EXISTING SOURCE FILE

Often you will load an existing source code file and compile it. Most integrated programming environments have an Open command that can be used to open source files.

EXERCISE 3-4 LOADING AND COMPILING AN EXISTING SOURCE FILE

1. Start your integrated programming environment.
2. Open the source file **Count.java** from the student data files.
3. Compile and run the program. The program quickly displays the numbers 1 through 10.
4. Close the source file.

Responsibilities of the Programmer

As you write more advanced computer programs, you should keep certain responsibilities in mind.

1. ***Privacy.*** *Programmers often have access to databases and other information about individuals. Programmers have a responsibility to protect the privacy of this information.*
2. ***Property Rights.*** *Ideas are not protected by copyright law. Actual program code, however, is. Using software that you do not have legal license to use, or using other programmers' source code without proper permissions, is illegal and irresponsible.*
3. ***Impact of Software.*** *Software can do physical damage to computer equipment (in the form of viruses), or have social ramifications. Programmers should not use computers to cause harm to users and should consider the impact of a program on society before writing a program.*
4. ***Reliability.*** *Individuals, schools, businesses, and the government rely on computers more every year. Programmers have a responsibility to produce software that is as reliable as possible and to report and/or repair problems that may affect the reliability of the system.*

CONGRATULATIONS

Congratulations. You now know the basics of creating and running Java programs. From here you will simply add to your knowledge, which will enable you to write more useful programs. If you feel you need more experience with compiling and running Java programs, repeat the exercises in this chapter or ask your instructor for additional help. Future exercises require that you know how to compile and run Java programs.

SECTION 3.2 QUESTIONS

SHORT ANSWER

1. Which company developed the compiler you are using?
2. Which is the name of the compiler you are using and its version number?
3. Which command or commands are used to run a program with your compiler?
4. Which command opens a source code file from a disk?
5. Which command saves a source code file?

CHAPTER 3, SECTION 3

First Steps: An Introduction to Object-Oriented Programming

Object-oriented programming (OOP) is more than a programming language. OOP is more of a philosophy of how to solve problems, and more specifically, large-scale problems. Languages such as Java, C++, and Smalltalk are examples of languages that are used to write object-oriented programs. OOP, however, has much more to do with how you think about a program than which language is ultimately used to code the program.

A program that prints out "Hello World!" or adds a few numbers together can be programmed in a simple step-by-step process we call ***procedural programming***. However, what do you do when you must develop a program that contains graphical interfaces, such as windows, buttons, or check boxes? What do you do when your company is hired to develop a major software product such as a word processor, a spreadsheet, or a database program? In the software industry, the object-oriented approach to developing software has been the trend for large software projects. In addition, languages such as Java are making OOP a good choice for projects of all sizes.

The OOP County Fair

To understand the difference between procedural programming and object-oriented programming, let's look at two different ways a county fair might be run.

THE PROCEDURAL MODEL

In the first approach to running the county fair, each person or family arriving at the fair is met by an individual (a guide) at the gate. This guide takes the visitors' tickets and stays with the visitors throughout their stay. The same person would strap the visitors into their seats for the roller coaster and then run

it; would fix the visitors hot dogs and take their money; and would clean up the streets and sidewalks after the visitors went home.

In this approach, the fair is run like a procedural program. Think of the person who guides visitors through the fair as a procedural program. As a procedural programmer, you would have to think ahead and be prepared at any time for your program to serve the needs of the fair-goers. Your program might be waiting for the next request, and would have code that looked like:

```
if the visitors need cotton candy then .....
if the visitors want to ride the train then .....
if the visitors are ready to leave then .....
```

As long as the fair does not get too large, this procedural approach will work well and be fairly easy to develop. However, if the roller coaster is updated to a newer and faster one, or another new ride is installed, or a new milkshake stand is added, you have to change the parts of the program that address the changes in operating instructions. Of course, you will also have to insert new "if" statements at every point in the program where the visitors might want to ride the roller coaster or drink a milkshake.

Most important, each time you write or rewrite part of the program, you have to make sure that everything still works together without error and that your new code does not have any errors of its own or introduce any further errors to the rest of your program—a process called ***debugging***. Now, as you can see, as the fair program becomes larger, developing and maintaining the program is going to become more difficult.

THE OBJECT-ORIENTED MODEL

The alternative approach to running our county fair is an object-oriented way. By using an object-oriented approach, you have much more room for improvement and growth, and many fewer problems with the programming and debugging process. What you do from the start is write a set of rules on how to take tickets. You write a set of rules for operating a concession or souvenir stand where food and merchandise are sold. You write another set of rules on how to strap people into the roller coaster and run it. You write still another set of rules on how to clean up the streets, sidewalks, and store windows. In OOP, we would call each of these sets of rules the Ticket Taker *class*, the Retail Outlet *class*, the Roller Coaster *class*, and the Facility Maintenance *class*.

This seems like a nice system, but remember, rules do not get the job done. It takes people following those rules to make the fair really happen. In OOP, we would say that the person following the ticket-taking rules would be a Ticket Taker *object*, the person running the roller coaster, and following Roller Coaster class rules, would be a Roller Coaster *object*, and so on. This is a very different approach from using procedural programming. However, if you have ever attended a county fair, you will note that this is how a fair is really run!

Now let's look at how you have improved the programming task. First, if you look at "running the program," you will note that the visitors (called "users" in programming) are now actively involved in the decision-making process. Instead of writing a whole list of "if" statements and trying to guess what the visitors will do next, your user selects an object, and the object knows what to do. This is quite a bit simpler.

But what happens if you upgrade the roller coaster? The answer is, all you have to do is upgrade the roller coaster! You do not have to make any other changes to the program that would take so much time and possibly introduce more problems. The next time a visitor comes to the park, she just goes to the roller coaster object and rides. The process for selecting the roller coaster is the same, but the visitor gets to enjoy the newer and faster ride.

Building Programs Using Objects

Let's consider how an object-oriented approach could help make a county fair easier to create and manage. Suppose we analyze the attractions at the fair and come to the conclusion that those attractions fall into three categories:

- Rides
- Shows
- Retail outlets

Our fair may consist of 18 rides, 7 shows, and 11 retail outlets (stores). Even so, there are things about each of those rides, shows, and stores that are the same. For example, running each of the 11 retail outlets involves taking money, stocking and delivering a product, opening, closing, and other operations common to retail outlets. Whether the retail outlet is a lemonade stand or a souvenir shop, many of the "rules" of operation are the same.

When a milkshake stand is created, the standard operations of a retail outlet are necessary to run the milkshake stand. Other rules and operations that are specific to a milkshake stand (such as how to make a milkshake) are added to the rules and operations of a retail outlet to make a milkshake stand. In effect, the retail outlet is customized to the needs of a milkshake stand.

In Chapter 2 you wrote HTML code that customized the scrolling text applet. What you were actually doing was causing a customized version of the scrolling text applet to be created, much like the milkshake stand would be a customized version of the retail outlet.

With object-oriented programming, you can create all 11 retail outlets, each customized to serve a unique purpose. Rather than writing 11 different programs for 11 different outlets, you can use the same "class" to create all 11 stores.

But the advantages don't stop there. If the standard rules for a retail outlet change, one change to the basic retail outlet class will change all 11 stores. For example, if the rules for a retail outlet change to refuse any bill over $20, the program need only be changed in one place to apply the change to all 11 outlets.

Object-oriented programming is very powerful and very effective. However, it is not the only way to solve programming problems. The important point is that object-oriented programming can simplify many programming tasks—especially large ones.

There is much more to OOP than can be learned from this simple analogy. You are, however, now on your way to an understanding of the philosophy upon which Java is designed. In the chapters to come, you will learn more about object-oriented programming, and you will see why an understanding of OOP is essential to successful Java programming.

SECTION 3.3 QUESTIONS

SHORT ANSWER

1. What is the name of the method of programming where programs follow a simple, step-by-step process?
2. What is the name of the process where a program is tested for errors and those errors are removed?
3. What approach to programming involves writing a set of rules called a class?
4. Suppose you are designing a class that will include the standard rules and operations of the rides in the fair. One item that would be part of your class would be the length of time the ride runs. List two other items that could be part of your class.
5. Suppose you want to use your ride class to help create a roller coaster ride. What items of information would be specific to a roller coaster that might not be included in your basic ride class?

KEY TERMS

braces	method
class	object
comments	object-oriented programming (OOP)
debugging	procedural programming
lowercase	statements
main method	uppercase

SUMMARY

- A Java program has several parts.
- Comments are remarks that are ignored by the compiler. They allow you to include notes and other information in the program's source code.
- All Java programs have at least one class definition. A class definition tells the compiler how to create an object.
- All Java programs have a main method. The main method is where the program begins running.
- Braces mark the beginning and end of blocks of code.
- Statements are the lines of code the computer executes. Each statement ends with a semicolon.
- Java allows you to indent and insert space in any way that you want. You should take advantage of this flexibility to format source code in a way that makes programs more readable.

- Java is case sensitive, which means that using the wrong capitalization will result in errors.
- Java is an object-oriented programming language
- Object-oriented programming (OOP) is a newer approach to programming that supports large-scale program development and easier, more reliable program maintenance.
- In object-oriented programming, a class is the set of rules that explain what certain things are and how they will be managed or taken care of.
- In OOP, an object is the actual thing that follows the rules and does the job specified by a class.

PROJECTS

PROJECT 3-1

Enter and run the program shown below but substitute your name and the appropriate information for your compiler. Save the source code as **compinfo.java** to your student disk or folder.

```
// COMPINFO
// By Jeremy Wilson

class compinfo
{
public static void main(String args[])
{
System.out.println("This program was compiled using");
System.out.println("Microsoft Visual J++ version 1.1");
}
}
```

PROJECT 3-2

Open the source file **Revcount.java** from the student data files. Compile the source file and then run it. After you have run the program, close the source file and quit.

PROJECT 3-3

Write a program that prints the message of your choice to the screen. Make the message at least four lines long. Save the source code file as **my_msg.java** to your course student disk or folder.

4 Data Types and Strings

OBJECTIVES

- *List different data types used in Java and explain how they are processed in the computer.*
- *Declare, name, and initialize variables.*
- *Output variable values to the console.*
- *Discuss OOP using class wrappers and strings.*

Overview

Computer programs process data to manage and produce information. The job of the programmer, then, is to properly organize data for this storage and use. Most data used by programs is stored in either variables or constants.

A ***variable*** holds data that can change while the program is running. A ***constant*** is used to store data that remains the same throughout the program's execution. In this chapter, you will learn about data types used by most computers, and then you will step into the object-oriented data types used by Java.

CHAPTER 4, SECTION 1

Data Types

There are eight standard variable types in Java. These ***data types***, shown in Table 4-1, are often called the intrinsic data types. The byte, char, short, int, and long data types are for storing integers (whole numbers such as 5 or 27). The float and double data types are for floating-point numbers (numbers having fractional parts such as 7.75 or 92.6).

The char type is used to store single characters such as *A*. However, because Java uses the Unicode character set, char actually stores the numeric value based on its Unicode equivalent numeric value. That is, *A* would be stored as 65 (however, when it is displayed, Java converts the 65 back to *A*). Thus, char is a numeric type. The Boolean type is not numeric. It has a value of either true or false.

DATA TYPE	SIZE (IN BITS)
Boolean	8
byte	8
char	16
short	16
int	32
long	64
float	32
double	64

TABLE 4-1

You may recall from math courses that an integer is a positive or negative whole number, such as –2, 4, or 5133. Real numbers can be whole numbers or decimals and can be either positive or negative, such as 1.99, –2.5, 3.14159, and 4.

Data types are of little concern to the average person. When programming in Java, however, you must select a type of variable, called a data type, that best fits the nature of the data itself. Let's now examine the data types that are available for Java variables.

INTEGER TYPES

When you are working with either positive or negative whole numbers, you should use integer data types for your variables. There are several integer data types available in Java. Selecting which integer data type to use is the next step.

Table 4-2 lists the variable data types used for storing integers. Notice the range of values that each type can hold. For example, any value from –32,768 to 32,767 can be stored in a short data type variable. If, however, you need to store a value outside of that specific range, you must choose a different data type.

TABLE 4-2

MINIMUM INTEGER DATA TYPES	MINIMUM RANGE OF VALUES	NUMBER OF BYTES OCCUPIED
byte	–128 to 127	1
short	–32,768 to 32,767	2
int	–2,147,483,648 to 2,147,483,647	4
long	–9223372036854775808 to 9223372036854775807	8

Why would you want to use the int type when the long type has a bigger range? Notice the third column of Table 4-2. The variables with the larger ranges require more of the computer's memory. In addition, it may take the computer longer to access the data types that require more memory. Having all of these data types gives the programmer the ability to use only what is necessary for each variable, decrease memory usage, and increase speed.

CHARACTERS

Recall from Chapter 1 that in the computer, characters are stored as numbers, and that when programming in Java, the computer assigns numbers to the characters using Unicode. Because the computer considers characters to be numbers, they are stored in an integer data type named char. Again, as stated above, this allows the computer, which can hold only numbers, to be able to hold the character data.

FLOATING-POINT TYPES

Integer variables are inappropriate for certain types of data. For example, tasks as common as working with money call for using floating-point numbers.

There are two types of floating-point variables available to Java. The first, simply called the float, has 32 bits or 4 bytes of memory set aside to hold the data. The second, the double, has 64 bits or 8 bytes of memory set aside for the data. In Java, there is also a third quantity defined as either a float or a double. This quantity is called *NaN* (Not a Number), and may be generated if an attempt is made to divide by zero, or some other mathematical action is taken that does not result in a usable number.

Extra for Experts

Making Floating-Point More Efficient

Earlier you read that floating-point numbers are not as efficient as integers. This is true. But many computer manufacturers include a floating-point unit (FPU) to help with the problem. A ***floating-point unit*** *is a processor that works with floating-point numbers. FPUs are often called* ***math coprocessors****. Microprocessors with an FPU can perform calculations with floating-point numbers much more quickly than a microprocessor without an FPU. Some FPUs are on a chip separate from the microprocessor and some are built into the microprocessor chip.*

Table 4-3 lists the two floating-point data types and their range of values. The range of floating-point data types is more complicated than the range of integers. Selecting an appropriate floating-point type is based upon both the range of values and the required decimal precision.

TABLE 4-3

FLOATING POINT DATA TYPES	APPROXIMATE RANGE OF VALUES	DIGITS OF PRECISION	NUMBER OF BYTES OCCUPIED
float	3.4×10^{-38} to 3.4×10^{38}	7	4
double	1.8×10^{-308} to 1.8×10^{308}	15	8

When you are choosing a floating-point data type, first look to see how many digits of precision are necessary to store the value. For example, storing π as 3.1415926535897 requires 14 digits of precision. Therefore, you should use the double type in this case. You should also verify that your value will fit within the range of values the type supports. But unless you are dealing with very large or very small numbers, the range is not usually as important an issue as the precision.

Let's look at some examples of values and what data types would be appropriate for the values.

Dollar amounts in the range –$99,999.99 to $99,999.99 can be handled with a variable of type float. A variable of type double can store dollar amounts in the range –$9,999,999,999,999.99 to $9,999,999,999,999.99.

The number 5.98×10^{24} kg, which happens to be the mass of the Earth, can be stored in a variable of type float because the number is within the range of values and requires only three digits of precision.

BOOLEAN VARIABLES

A ***Boolean variable*** can have only two possible values. One of the values represents true (or some other form of the affirmative) and the other value represents false (or some other form of the negative). In some languages, certain numbers represent the TRUE or the FALSE condition; however, in Java, the only Boolean values allowed are the actual words *true* and *false*. Boolean variables are very useful in programming to store information such as whether an answer is yes or no, whether a report has been printed or not, or whether a device is currently on or off.

Note

Boolean variables are named in honor of George Boole, an English mathematician who lived in the 1800s. Boole created a system called Boolean algebra, which is a study of operations that take place using variables with the values true and false.

SECTION 4.1 QUESTIONS

TRUE/FALSE

T F 1. An integer is a number with digits after the decimal point.

T F 2. The byte data type has a range of values of 0 to 255.

T F 3. A variable of type long can store the number 3,000,000,000.

T F 4. Each char variable can store one character.

T F 5. A float variable has 10 digits of precision.

SHORT ANSWER

1. What integer data type is necessary to store the value 4,199,999,999?
2. Why is it important to use data types that store your data efficiently?
3. What range of values can be stored in an int variable?
4. What floating-point data type provides the most digits of precision?
5. What is a Boolean variable?

CHAPTER 4, SECTION 2

Using Variables

You are now going to have an opportunity to put your knowledge to work and select the right variable for the job. You must first indicate to the compiler what kind of variable you want and what you want to name it. Then, it is ready to use.

DECLARING AND NAMING VARIABLES

Indicating to the compiler what type of variable you want and what you want to call it is called declaring the variable.

DECLARING VARIABLES

You must ***declare*** a variable before you can use it. The Java statement declaring a variable must include the data type followed by the name you wish to call the variable, and a semicolon. An integer variable named `i` is declared below.

```
int i;
```

Table 4-4 shows that declaring variables for other data types is just as easy as the example above.

Java will allow you to declare a variable anywhere in the program as long as the variable is declared before you use it. However, you should get into the habit of declaring all variables at the top of the method. Declaring variables at the top makes for better organized code, makes the variables easy to locate, and helps you plan for the variables you will need.

DATA TYPE	EXAMPLE JAVA DECLARATION STATEMENT
char	`char Grade;`
int	`int DaysInMonth;`
short	`short temperature;`
long	`long PopulationChange;`
float	`float CostPerUnit;`
double	`double DistanceToDeltaQuadrant;`

TABLE 4-4

NAMING VARIABLES

The names of variables in Java are typically referred to as ***identifiers.*** Notice how the variable names in Table 4-4 are very descriptive and consider how they might help the programmer recall the variable's purpose. You are encouraged to use the same technique. For example, a variable that holds a bank balance could be called **`balance`**, or the circumference of a circle could be stored in a variable named **`circumference`**. The following are rules for creating identifiers.

- Identifiers must start with a letter, an underscore (_), or a dollar sign ($). You should, however, avoid using identifiers that begin with underscores and dollar signs because the compiler's internal identifiers often begin with underscores or contain dollar signs.
- As long as the first character is a letter, you can use any letters or numerals in the rest of the identifier.
- Use a name that makes the purpose of the variable clear, but avoid making it unnecessarily long.
- Identifiers cannot contain spaces. A good way to create a multiword identifier is to use an underscore between the words (for example, **`last_name`**) or to use names with capital letters at the beginning of each word (**`LastName`**).
- Some Java programmers also identify variable types in the name, for example, **`i_numCars`**, which identifies numCars as an integer. In addition, you may see some variables start with an *m*, as in **`m_lastname`**, which indicates that this variable is a member of a certain class (you will be reading about members later).

Note

Any format that helps you or someone reading your code to understand what the variables represent or what they do will improve your code readability, and therefore maintainability. Professional programmers always write their code to be read by others for exactly this reason.

The following words, called ***keywords*** or reserved words, must NOT be used as identifiers because they are part of the Java language. Your compiler may have additional keywords not listed here.

```
abstract    extends     interface   static
case        final       long        super
catch       finally     native      switch
char        float       new         synchronized
class       for         null        thisboolean
const       goto        package     throwbreak
continue    if          private     transientbyte
default     implements  protected   try
do          import      public      void
double      instanceof  return      volatile
else        int         short       while
```

Recall from the previous chapter that Java is case sensitive. The capitalization you use when the variable is declared must be used each time the variable is accessed. For example, `total` is not the same identifier as `Total`.

Table 4-5 gives some examples of illegal identifiers.

TABLE 4-5

IMPROPER JAVA VARIABLE NAMES	WHY ILLEGAL
`Miles per gallon`	Spaces are not allowed
`import`	import is a keyword
`4Sale`	Identifiers cannot begin with numerals

DECLARING MULTIPLE VARIABLES IN A STATEMENT

You can declare more than one variable in a single statement as long as all of the variables are of the same type. For example, if your program requires three variables of type float, all three variables could be declared by placing commas between the variables like this:

```
float x, y, z;
```

or

```
double a, b, c;
```

INITIALIZING VARIABLES

The compiler assigns a location in memory to a variable when it is declared. However, a value already exists in the space reserved for your variable. This is because a random value could have been stored when the computer was turned on or the location could retain data from a program that ran earlier. Regardless, the memory location now belongs to your program, and you must specify the initial value to be stored in the location. This process is known as initializing.

WARNING

One of the most common mistakes made by first-time programmers is not initializing variables. There may be some conditions where a variable is sure to be given a value, such as a variable that receives data input by the user; and there may be conditions where a variable is only initialized under certain conditions. Your job is to make sure that every variable gets some valid information stored in it before it is used or output.

Extra for Experts

Upon turning on your computer, each byte of memory is filled with a value. The compiler sets up memory locations for your variables, and you must initialize them. When your program is through with a variable, that memory location is once again made available to be used by a different variable or for another purpose. The value you last stored in the variable will remain in that memory location until it is assigned another value.

To *initialize* a variable, you simply assign it a value. In Java, the equal sign (=) is used to assign a value to a variable. In Figure 4-1, the variables `i` and `j` are initialized to the values of 3 and 2, respectively. Notice that the variable `k` has yet to be initialized because it is to be assigned the sum of `i` and `j`.

```
int i = 3; // variable declared and initialized in one statement
int j, k;
j = 2;
k = i + j;
```

FIGURE 4-1
Initializing variables and assigning their sum to another variable.

EXERCISE 4-1 DECLARING VARIABLES

In this exercise, you will demonstrate assigning a value to a variable. In addition, you will implement the Java console output method. In a process you will read about later, the java *System* class has a member family of output methods named *out*, and one of the member methods of *out* is *println*. Again, while you will be studying this later in the book, you can begin your application of object-oriented programming now by giving this output process a try.

1. Enter the following program into a blank text editor file:

```
// iDeclare.JAVA
// Example of variable declaration.
class iDeclare
{
  public static void main(String args[])
{
  System.out.println("Begin Program\n");
  int myVal;

  myVal = 0;

  System.out.println("Once initialized, myVal is: " + myVal);
  }
}     // end of class iDeclare
```

2. Save the source code file as **iDeclare.JAVA** to your student disk or folder.
3. Compile and run the program. The program should print *Begin Program* and then the number *0* (zero) on your screen. If no errors are encountered, leave the program on your screen. If errors are found, check the source code for keyboarding errors and compile again.
4. Next, place the following code before the line containing "myVal = 0;"

   (make sure you place all the code on one line):

```
System.out.println("Before initialization,
                    myVal is: " + myVal);
```

What happens? You can recognize that you are trying to output a value before it has been initialized, so in most cases, you would get inconsistent results (called "garbage" in the programming business). However, you may find that some compilers won't even let you do this—attempting to build a program such as this without variable initialization results in an error. What did your compiler do?

5. Change the initialization statement to set the value of `i` to –40 and run the program again. The number –40 should be shown on your screen. Save the source code again and leave the source code file open for the next exercise.

EXPONENTIAL NOTATION

Assigning a floating-point value to a variable works the way you probably expect, except when you need to use ***exponential notation.*** You may have used exponential notation and called it scientific notation. In exponential notation, very large or very small numbers are represented with a fractional part (called the mantissa) and an exponent. Use an *e* to signify exponential notation. Just place an e in the number to separate the mantissa from the exponent. Below are some examples of statements that initialize floating-point variables.

```
x = 2.5;
ElectronGFactor = 1.0011596567;
Radius_of_Earth = 6.378164e6; // radius of Earth at equator
Mass_of_Electron = 9.109e-31; // 9.109 x 10^-31 kilograms
```

EXERCISE 4-2 DECLARING AND INITIALIZING FLOATING-POINT VARIABLES

1. Modify the program on your screen to match the program below.

```
// iDeclare.JAVA
// Example of variable declaration.
class iDeclare
{
  public static void main(String args[])
  {
  float x, MassOfElectron;
  double Radius_of_Earth;
  int myNewVal;

  System.out.println("Begin Program\n");

  myNewVal = 0;
  x = (float) 2.5;
  Radius_of_Earth = 6.378164e6;
  MassOfElectron = (float) 9.109e-31;

  System.out.println("myNewVal = " + myNewVal);
  System.out.println("x = " + x);
  System.out.println("Earth's radius: " + Radius_of_Earth);
  System.out.println("Electron's mass: " + MassOfElectron);
  }
}     // end of iDeclare
```

2. Compile and run the program. The integer and three floating-point values should print to the screen.

Why did you have to put the `(float)` in front of two of the variables? Try removing this from in front of the `MassOfElectron` variable. What happens?

Some compilers will only recognize floating point numbers as double types. By inserting `(float)` in front of the number, you are said to be ***casting*** (or type casting) the variable to a float. In short, this means you are telling the compiler to treat the quantity as a certain type *other than what it looks like* to the compiler. In Java, as you will see later, there are several conditions when casting is used.

3. After trying these experiments and seeing your program run successfully, save and close the source code file.

SECTION 4.2 QUESTIONS

SHORT ANSWER

1. What are the words called that cannot be used as identifiers because they are part of the Java language?
2. Why can't "first name" be used as an identifier?
3. Write a statement that declares four int data type variables `i`, `j`, `k`, and `l` in a single statement.
4. What character is used to assign a value to a variable?
5. If you were to cast a candle into a light bulb, what would the compiler think you were doing?

CHAPTER 4, SECTION 3

Object-Oriented Data Types and Strings

The data types you have worked with up to this point are often referred to as ***primitive data types*** or ***intrinsic data types.*** The primitive data types are not in an object-oriented format. They are based on similar data types in the C++ language. The designers of Java chose to keep the original primitive types for the simplest of calculations and mathematical manipulations. However, they also chose to develop a group of data type classes that would really be object-oriented data types. These data type classes are called ***class wrappers*** or ***object wrappers.***

Why do you need object-oriented data types? The answer can be found in the OOP county fair analogy. Recall that the objects at the fair each know how to take care of themselves. The Roller Coaster object knows how to get power to operate the roller coaster from the outside, how to strap people into the carts, and how to make the roller coaster go. All objects, by way of obeying their class rules, know how to manage themselves, which is why object-oriented programming is popular. The programmer doesn't have to worry about the details of something that knows how to take care of itself.

One of the most frequent conditions to manage in Java is inputting and outputting data. The fact of the matter is, virtually all data comes into and goes out of Java programs as sets of characters called ***strings.*** As you may have seen before, strings are just groups of characters, such as your name or a sentence. Figure 4-2 shows several examples of strings. They can contain spaces, commas, and many other characters besides just letters.

When you enter a number as input to a program, you are actually inputting a string of characters. For example, the string "678" does not mean six-hundred-seventy-eight unless you interpret the string that way. The numerals 0 though 9 are ASCII characters, just like the alphabetic characters. When inputting data into a program, it is easiest to bring in a string and, if necessary, convert the string to a numeric value inside the program.

```
"abcdefghijkl"
"123456789"
"Bill Smith"
"K9"
"How now brown cow."
```

FIGURE 4-2
Examples of strings, or groups of characters.

Many times, processing all input as strings improves reliability of a program because the computer can take in a series of characters, test them to see if they are numbers, and then work on them if they are, in fact, numbers. In other programming languages, entering the character *E* when the user should have entered *3* will cause errors or even unpredictable behavior. In Java, you can simply watch for invalid input and deal with it appropriately.

This leads us to an example of why an object-oriented data type may be more useful than a primitive data type. What if we had a data type that could convert itself from a string to a numeric data type (such as an integer) and back to a string. In Java, this is not a problem. Consider the program in Figure 4-3. The program creates two objects: a Double and a String.

```
class WrapperTest
{
  public static void main(String args[])
  {
  System.out.println("Program Begins");

  // constructing a new Double
  Double myDoubleVal = new Double(4.52);
  // CONSTRUCTING A NEW STRING
  String myStringVal = new String("6.73 abc");

  System.out.println("My String value: " + myStringVal);
  System.out.println("My Double value: " + myDoubleVal);
  System.out.println("My Double value as a string: " +
        myDoubleVal.toString());
  }
}     // end of WrapperTest
```

FIGURE 4-3
Class wrappers make it possible for data types to behave like objects.

Note

Class wrappers are called by the same name as their primitive counterparts. Notice, however, that the names of the data types are capitalized when the data type is a class. Remember, Java is case sensitive. The compiler treats "Double" differently from "double."

EXERCISE 4-3 MANAGING CLASS WRAPPER DATA TYPES AND WORKING WITH STRING OBJECTS

1. Enter the program shown in Figure 4-3, save it to your student disk **WrapperTest.java** to your student disk or folder, compile it, and then run it. Your output should be as follows:

```
My String value: 6.73 abc
My Double value: 4.52
My Double value as a string: 4.52
```

Notice that the string and the double values are interchangeable thanks to being treated the same. Everything in OOP is an object, and it simplifies our programming lives to be able to treat all things with the same actions.

2. Now try to get the integer quantity out of the string value. To do that, add the following line to the end of your code:

```
System.out.println("My String converted back to an int: " +
Integer.parseInt(myStringVal));
```

What happens? You will now experience your very first Java exception (much like an error—you will study exceptions in Chapter 6). The Integer parser (something that breaks things into smaller parts) gets confused trying to turn a space and some letters into a number. So try a different approach.

3. Remember that a String object should know how to take care of itself, so shouldn't it be able to give you a smaller chunk of its larger self? It does. Replace the code from item 2 with the following code:

```
String newStringVal = new String(myStringVal.substring(0, 1));

System.out.println("My String converted to an int: "
        + Integer.parseInt(newStringVal, 10));
```

When you run your program with these two lines added, you will get the number 6 as output after the end of the other numbers. It is a little trickier to get a whole float or double value out of a string, but it can be done. This part of the exercise was to show you how a number (an integer) can be pulled from a string.

This exercise should show you something else as well; if you will start thinking like an object-oriented programmer, you will begin to make intuitive (using your intuition) guesses. If you begin to trust that objects can take care of themselves, you will start to think to yourself, "Geez, if that is a Giraffe object, I expect it can see far distances, or I expect it can eat leaves at the tops of tall trees."

Think like this—ask yourself what a certain kind of object would be *expected* to do; in most cases you will be correct.

Let's look back at what you have written. Look at the following code again:

```
// constructing a new Double
Double myDoubleVal = new Double(4.52);
```

The comment states that you are "constructing" a new **Double** object. You will see more discussion of constructors later, but again, using your intuition you can see that it means we are "creating" a new kind of class. Here is how it is done. First, you have to tell the compiler what the name of the object is going to be, and yours is going to be named **myDoubleVal**. To be an object, it needs to know what rules to follow (what kind of class it will represent), and you can see it will be a **Double** object.

Then you use the *new* keyword to get some memory space set up for your object, and then you use the word Double a second time to get the object started. Again, the meanings and uses of these words will be made clearer as the textbook progresses; the important thing for you at this point is to know what the declaration format should look like.

The number "4.52" is said to be a parameter that is passed to the method **Double** so it knows what value to give to **myDoubleVal**. Now this can be confusing, but don't let it bother you. With each chapter in this book, you will be picking up more parts of the whole so that you can slowly build up your understanding of the big picture.

For example, right now you are asking, "Why is there a Double on the left side, and the very same word on the right side?" The answer is, the words on both sides are supposed to be the same, but they do different jobs. The **Double** on the left side is informing **myDoubleVal** what kind of object it will be. The **Double** on the right side, which as we said is a constructor, is getting **myDoubleVal** started up correctly with the number 4.52.

Now look at the first line you added under step 3 of this exercise. Before you read on, see if you can figure out what each part does—use your intuition. Got it? Check yourself with the following:

1. The first word, **String**, tells the variable what kind of object it will be, and what rules the object will have to follow; String is a class name.

2. The second word, **newDoubleVal**, is going to be the name of the object that will follow the rules of a String class.

3. After the equals sign (the assignment operator) comes the word **new**, which tells the computer to set aside the right amount of memory for a String object.

4. After the word **new** comes the word **String** again, only now this word is acting as a constructor, getting **newDoubleVal** ready to go with new data.

5. The parameter (or necessary information) that is passed to the String constructor is a whole method or operation of its own. You are passing a substring (a smaller portion) of **myDoubleVal** starting with the first character (at position 0 [zero]), and ending with the second character (at position 1). Again, **myDoubleVal**, being a String object, knew how to take out its first character and pass it to another method.

SECTION 4.3 QUESTIONS

SHORT ANSWER

1. What is the difference between a class and an object?
2. Why should Java have class wrappers?
3. If you had a float quantity, but you wanted it to "look" like an integer, how would you do this?
4. The second line of section five of Exercise 4-3 is as follows:

```
System.out.println("My String converted to an int: "
 + Integer.parseInt(newStringVal, 10));
```

What does this line of code do?

KEY TERMS

Boolean variable	initialize
casting	intrinsic data types
class wrappers	keyword
constant	math coprocessor
data type	new
declare	object wrappers
exponential notation	primitive data types
identifier	string
identifier	variable

SUMMARY

- Most data is stored in either variables or constants.
- There are several types of variables. Some are for integer data and some are for floating-point data.
- Integer data types are selected based on the range of values you need to store.
- Characters are stored in the computer as numbers. The char data type can store one character of data.
- Floating-point data types are selected based on the range of values and the required precision.
- Boolean variables can have only two possible values: true or false.
- Variables must be declared before they are used. Variables should also be initialized to clear any random values that may be in the memory location. When a variable is declared, it must be given a legal name called an identifier.

- Output to the Java console is implemented with the member method *System.out.println*.
- Class wrappers allow primitive data types to be treated and used as objects.
- Groups of characters together are called strings in programming languages.
- Java uses String classes to manage character strings in programs.

PROJECTS

PROJECT 4-1

1. Enter, compile, and run the following program. Save the source code file as **DataType.java** to your student disk or folder.

```
// DataType.java
// Examples of variable declaration and
// initialization.

class DataType
  {
  public static void main(String args[])
  {
  // declare a double for the square root of two
  // remember that you cannot declare a constant (final)
  // unless you are in a unique class
  double SQUARE_ROOT_OF_TWO = 1.414214;

  int i; // declare i as an integer
  long j; // j as a long integer
  long k; // k as a long integer
  float n; // n as a floating point number

  i = 3; // initialize i to 3
  j= -2048111; // j to -2,048,111
  k=40000000021; // k to 4,000,000,002
  // Note: the l suffix above is required by some compilers
  n= (float) 1.887; // n to 1.887

  // output constant and variables to screen
  System.out.println(SQUARE_ROOT_OF_TWO);
  System.out.println("i = " + i);
  System.out.println("j = " + j);
  System.out.println("k = " + k);
  System.out.println("n = " + n);
  }

} // end of class DataType
```

2. Add declarations using appropriate identifiers for the values below. Declare *e*, the speed of light, and the speed of sound as constants. Initialize the variables. Use any identifier you want for those values that give you some indication as to their purpose.

100	e (2.7182818)
–1000	Speed of light (3.00×10^8 m/s)
–40,000	Speed of sound (340.292 m/s)
40,000	

3. Print the new values to the screen.
4. Save, compile, and run. Correct any errors you have made.
5. Close the source code file.

PROJECT 4-2

Write a program that declares two variables (A and B). Initialize A = 1 and B = 2.2. Next, declare an int named C and a float named D. Initialize C = A and D = B. Write statements to print C and D to the screen. Save the program to your student disk as **Project 4_2.java.** Compile and run the program. Close the source file.

5 Math Operations

OBJECTIVES

- *Use the arithmetic operators.*
- *Increment and decrement variables.*
- *List the order of operations.*
- *Use mixed data types.*
- *Demonstrate the ability to avoid overflow, underflow, and floating-point errors.*

Overview

In this chapter, you will learn about Java math operations and demonstrate each concept by compiling and running programs.

First you will learn about Java operators, some of which have already been mentioned. Next, you will learn how to build expressions using the operators. And finally, we'll talk about shortcuts and how to avoid pitfalls.

CHAPTER 5, SECTION 1

The Fundamental Operators

There are several types of operators. In this chapter you will be concerned only with the assignment operator, arithmetic operators, and some special operators.

Note

Appendix B contains a more complete list of operators.

ASSIGNMENT OPERATOR

You have already used the assignment operator (=) to initialize variables. The ***assignment operator*** changes the value of the variable to the left of the operator. Consider the statement below:

```
i = 25;
```

The statement `i = 25;` changes the value of variable `i` to 25, regardless of what `i` was before the statement.

Note

You have also used constructors to create numeric classes (class wrappers). For this chapter, we will focus on the simpler mathematical interactions that are implemented using the basic Java math operators.

EXERCISE 5-1 THE ASSIGNMENT OPERATOR

1. Start your Java compiler's text editor. In a new, blank document, enter the following program:

```
class iAssign
{
```

```
    public static void main(String args[])
    {
    System.out.println("Program Begins");

    int i = 10000;
    System.out.println("i = " + i);

    i = 25;
    System.out.println("i now = " + i);
    }
}    // end of iAssign
```

2. Save the source code file as **iAssign.java** to your student disk or folder. Compile and run the program. Notice the difference between the value of i when it is displayed after the first println and after the second.
3. Leave the source code file open for the next exercise.

MULTIPLE ASSIGNMENTS

Recall from Chapter 4 that you can declare more than one variable in a single statement. For example, instead of

```
int i;
int j;
int k;
```

you can use:

```
int i,j,k;
```

You can use a similar shortcut when initializing multiple variables. If you have more than one variable that you want to initialize to the same value, you can use a statement such as:

```
i = j = k = 25;
```

Multiple assignment is a trick that Java borrows from C++. It can be very handy under some circumstances when more than one variable needs to be assigned the same initial value. However, it is not a good idea to overuse this format. Remember, readability is one of your top goals.

EXERCISE 5-2 MULTIPLE ASSIGNMENTS

1. Modify the program on your screen to read as follows:

```
class iAssign
{
  public static void main(String args[])
  {
  System.out.println("Program Begins");
  int i, j, k;
```

```
        i = j = k = 25;

        System.out.println("i = " + i);
        System.out.println("j = " + j);
        System.out.println("k = " + k);
        }
    }     // end of iAssign
```

2. Compile and run the program. The program's output is:

```
i = 25
j = 25
k = 25
```

3. Close the source code file without saving.

As you have seen demonstrated already, variables may be declared and initialized in a single statement. Both of the following are valid Java statements:

```
int i = 2;
float n = 4.5;
```

ARITHMETIC OPERATORS

A specific set of ***arithmetic operators*** is used to perform calculations in Java. These arithmetic operators, shown in Table 5-1, may be somewhat familiar to you. Addition and subtraction are performed with the familiar + and - operators. Multiplication uses an asterisk (*), and division uses a forward slash (/). Java also uses what is known as a modulus operator (%) to determine the integer remainder of division. A more detailed discussion of the modulus operator is presented later in this section.

TABLE 5-1

SYMBOL	OPERATION	EXAMPLE	READ AS...
+	Addition	3 + 8	three plus eight
-	Subtraction	7 - 2	seven minus two
*	Multiplication	4 * 9	four times nine
/	Division	6 / 2	six divided by two
%	Modulus	7 % 3	seven modulo three

USING ARITHMETIC OPERATORS

The arithmetic operators are used with two operands, as in the examples in Table 5-1. The exception to this is the minus symbol, which can be used to change the sign of an operand. Arithmetic operators are most often used on the right side of an assignment operator, as in the examples in Table 5-2. The portion of the statement on the right side of the assignment operator is called an ***expression***.

The assignment operator (=) functions differently in Java from the way the equal sign functions in algebra. Consider the following statement:

```
x = x + 10;
```

STATEMENT	RESULT
`cost = price + tax;`	*cost* is assigned the value of *price* plus *tax*
`owed = total - discount;`	*owed* is assigned the value of *total* minus *discount*
`area = l * w;`	*area* is assigned the value of *l* times *w*
`one_eighth = 1 / 8;`	*one_eighth* is assigned the value of 1 divided by 8
`r = 5 % 2;`	r is assigned the remainder of 5 divided by 2 by using the modulus operator
`x = -y;`	x is assigned the value of *-y*

TABLE 5-2

The statement on the previous page is invalid for use in algebra because the equal sign is the symbol around which both sides of an equation are balanced. The left side equals the right side. But your Java compiler looks at the statement differently. The expression on the right side of the equal sign is evaluated, and the result is stored in the variable to the left of the equal sign. In the statement above, the value of **x** is increased by 10.

EXERCISE 5-3

USING ARITHMETIC OPERATORS

1. Retrieve the file **Assign.java** from the student data files. Save the file to your student disk as **Assign.java**. Compile the program.
2. Look at the source code and try to predict the program's output.
3. Run the program and see if you were correct in your prediction.
4. Close the source code file.

MORE ABOUT MODULUS

The *modulus operator,* which may be used only for integer division, returns the remainder rather than the result of the division. As shown in Figure 5-1, integer division is similar to the way you divide manually.

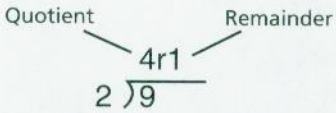


FIGURE 5-1
The division operator and modulus operator return the quotient and the remainder.

When integer division is performed, any fractional part that may be in the answer is lost when the result is stored into the integer variable. The modulus operator allows you to obtain the fractional part of the result as an integer remainder.

Consider the program in Figure 5-2. The user is prompted for two integers. Notice the program calculates the quotient using the division operator (/) and the remainder using the modulus operator (%).

EXERCISE 5-4

THE MODULUS OPERATOR

1. Enter, compile, and run the program from Figure 5-2. Save the source file as **Remain.java** to your student disk or folder.
2. Observe the output. Now, change dividend and divisor several times using values that will produce different remainders.
3. Leave the source file open for the next exercise.

```
// start of Remain.java
class Remain
{
  public static void main(String Args[])
  {
  System.out.println("Program begins");
  int dividend, divisor, quotient, remainder;
  // provide data quantities
  dividend = 37;
  divisor = 5;

  // process data, calculate answers
  quotient = dividend / divisor;
  remainder = dividend % divisor;

  // provide output
  System.out.println("The quotient is: " + quotient);
  System.out.println("and the remainder is: " + remainder);
  }
}    // end of class Remain
```

FIGURE 5-2
The program calculates the quotient and remainder using variables.

USING OPERATORS IN OUTPUT STATEMENTS

The program in Figure 5-2 required four variables and eight program statements. The program in Figure 5-3 accomplishes the same output with only two variables and six statements. Notice in Figure 5-3 that the calculations are performed in the output statements. Rather than storing the results of the expressions in variables, the program sends the results to the screen as part of the output.

```
class Remain2
{
  public static void main(String Args[])
  {
  System.out.println("Program begins");
  int dividend, divisor;
  // provide data quantities
  dividend = 37;
  divisor = 5;
  // process data, calculate answers and provide output
  System.out.println("The quotient is: " + dividend / divisor);
  System.out.println("and the remainder is: " + dividend % divisor);
  }
}     // end of class Remain2
```

FIGURE 5-3
Program calculates quotient and remainder in the output statements.

Avoid including the calculations in the output statements if you need to store the quotient or remainder and use them again in the program. But in situations like this, it is perfectly fine to use operators in the output statement. Always remember, though, that readability is one of your top goals.

EXERCISE 5-5 OPERATORS IN OUTPUT STATEMENTS

1. Modify the program on your screen to match Figure 5-3. Save the source file as **Remain2.java**. Compile it and run it.
2. Test the program to make sure you are still getting the same results.
3. Leave the source file open for the next exercise.

DIVIDING BY ZERO

Dividing by zero in math is without a practical purpose. The same is true with computers. In fact, division by zero always generates some type of error message.

EXERCISE 5-6 DIVISION BY ZERO

1. Change the program on your screen to set the divisor equal to zero. Compare and run the program again and see what exception message is generated.
2. Close the source code file without saving changes.

SPACES AROUND OPERATORS

Your Java compiler ignores blank spaces. Both of the statements shown below are valid.

```
x=y+z;
x = y + z;
```

The only time you must be careful with spacing is when using the minus sign to change the sign of a variable or number. For example, `x = y - -z;` is perfectly fine. The sign of the value in the variable `z` is changed and then it is subtracted from `y`. If you failed to include the space before the `-z`, you would have created a problem because two minus signs together (--) have another meaning, examined later in this chapter.

COMPOUND OPERATORS

Often you will write statements that have the same variable on both sides of the assignment operator. For example, you may write a statement such as `x = x + 2;` which adds 2 to the variable `x`. In cases like this, you may wish to use special operators, called ***compound operators***, available in Java that provide a shorthand notation for writing such statements.

Table 5-3 lists the compound operators, gives an example of each, and shows the longhand equivalent.

COMPOUND OPERATOR	EXAMPLE	LONGHAND EQUIVALENT
+=	i += 1;	i = i + 1;
-=	j -= 12;	j = j - 12;
*=	z *= 5.25;	z = z * 5.25;
/=	w /= 2;	w = w / 2;
%=	d %= 3;	d = d % 3;

TABLE 5-3

If you want to use a compound operator in a statement such as the one below, be careful.

```
price = price - discount + tax + shipping;
```

Because the compound operators have a lower precedence in the order of operations, you may get unexpected results. For example, simplifying the statement as shown below results in the wrong value because **`discount + tax + shipping`** is calculated and the sum of those are subtracted from **`price`**.

```
price -= discount + tax + shipping; // this gives the incorrect result!
```

In this case, you can work around the problem by using parentheses and changing the compound operator to +=. To properly calculate the discount, the unary operator (-) is used to subtract the discount by adding it as a negative value.

```
price += (-discount + tax + shipping);
```

The complexity of dealing with the order of operations in statements with more than one value on the right side of the operator sometimes makes the shorthand notation more trouble than it's worth. Until you are comfortable with the problems presented by the order of operators, you may want to use compound operators with simple statements only. And again, remember that you want people to be able to easily read your program code; don't let these or other shortcuts make your programming difficult to understand.

SECTION 5.1 QUESTIONS

SHORT ANSWER

1. What is the assignment operator?
2. What does the modulus operator do?
3. Write a statement that stores the remainder of dividing the variable **`i`** by **`j`** in a variable named **`k`**.
4. Write a statement that calculates the volume of a box (Vol = length × width × depth) given the dimensions of the box. Use appropriate identifiers for the variables. You do not have to declare types.
5. Write a statement that calculates the sales tax (use 7%) for an item given the cost of the item and the tax rate. Store the value in a variable named **`tax_due`**. Use appropriate identifiers for the other variables. You do not have to declare types.

PROBLEM 5.1.1

Write a program that uses the statement you wrote in question 4 above in a complete program that calculates the volume of a box. Save the source code file as **VOLBOX.java**.

PROBLEM 5.1.2

Write a program that uses the statement you wrote in question 5 above in a complete program that calculates the sales tax for an item given the cost of the item and the tax rate. Save the source code file as **SALESTAX.java**.

CHAPTER 5, SECTION 2

Counting by One and the Order of Operations

In this section, you will learn about two operators that allow you to increase or decrease the value stored in an integer variable by one. You will also learn the order that the computer applies operators in an expression.

INCREMENTING AND DECREMENTING

Adding or subtracting 1 from a variable is very common in programs. Adding 1 to a variable is called *incrementing,* and subtracting 1 from a variable is called *decrementing.* You will have the need to increment or decrement a variable when a program must execute a section of code a specified number of times, or when you need to count the number of times a process has been repeated.

Note

The ++ and -- operators can be used with any arithmetic data type, which includes all of the integer and floating-point types.

THE ++ AND -- OPERATORS

The C++ syntax provides Java with operators for incrementing and decrementing. In Java, you can increment an integer variable using the ++ *operator,* and decrement using the -- *operator,* as shown in Table 5-4.

TABLE 5-4

STATEMENT	EQUIVALENT TO...
counter++;	counter = counter + 1;
counter--;	counter = counter - 1;

You can increment or decrement a variable without the ++ or -- operators. For example, instead of `j++` you can use `j = j + 1`. With today's compilers, either way will produce efficient machine code.

EXERCISE 5-7 USING ++ AND --

1. Retrieve the file **Inc_Dec.java** from the student data files, and save it under the same name to your student disk.

```
// start of Inc_Dec.java
class Inc_Dec
{
  public static void main(String Args[])
  {
  int j;
```

```
    j = 10;
    j++;
    System.out.println("j = " + j);
    j--;
    System.out.println("j = " + j);
    }
}     // end of class Inc_Dec
```

2. Compile and run the program.
3. Examine the output and leave the source code file open for the next exercise.

VARIATIONS OF INCREMENT AND DECREMENT

At first glance, the ++ and -- operators seem very simple. But there are two ways each of these operators can be used. The operators can be placed either before or after the variable. The way you use the operators changes the way they work.

Used in a statement by themselves, the ++ and -- operators can be placed before or after the variable. For example, the two statements shown below both increment whatever value is in `j`.

```
j++;
++j;
```

The difference in where you place the operator becomes important if you use the ++ or -- operator in a more complex expression, or if you use the operators in an output statement. First let's look at how the placement of the operators affects the following statement. Assume that `j` holds a value of 10.

```
k = j++;
```

In the case of the statement above, `k` is assigned the value of the variable `j` before `j` is incremented. Therefore, the value of 10 is assigned to `k` rather than the new value of `j`, which is 11. If the placement of the ++ operator is changed to precede the variable `j` (for example `k = ++j;`), then `k` is assigned the value of `j` after `j` is incremented to 11.

EXERCISE 5-8 OPERATOR PLACEMENT

1. Add a statement to the **Inc_Dec.java** file that declares `k` as a variable of type int.
2. Add the following lines to the program before the closing brace.

```
k = j++;
System.out.println("k = " + k);
System.out.println("j = " + j);
k = ++j;
System.out.println("k = " + k);
System.out.println("j = " + j);
```

3. Save the new source code file as **Inc_Dec2.java**.
4. Don't forget to change the class name to **Inc_Dec2**. Compile and run the program to see the new output.
5. Close the source code file.

The increment and decrement operators can be used to reduce the number of statements necessary in a program. Consider the two statements below.

```
attempts = attempts + 1;
System.out.println("This is attempt number " + attempts);
```

Using the increment operator, one statement can do the job of two.

```
System.out.println("This is attempt number " + (++attempts));
// parentheses used around ++ for clarity
```

In both cases, the variable named *attempts* is incremented and its value is printed to the screen. In the second example, the variable is incremented within the println statement. The ++ operator must come before the variable named **attempts**. Otherwise, the value of the variable will be printed before it is incremented.

Note

Remember, the ++ operator actually changes the variable. If you replaced `++attempts` *with* `attempts + 1` *in the println statement, the value of the variable would remain the same. The screen, however, would print the result of* `attempts + 1`.

ORDER OF OPERATIONS

You may recall from your math classes the rules related to the order in which operations are performed. These rules are called the ***order of operations.*** The Java compiler uses a similar set of rules for its calculations. Operators in Java are applied in the following order:

1. Minus sign used to change sign (-)
2. Multiplication and division (* / %)
3. Addition and subtraction (+ -)

Java lets you use parentheses to override the order of operations. For example, consider the two statements in Figure 5-4.

There are additional operators you will learn about later. Also, see Appendix C for a complete table of the order of operators.

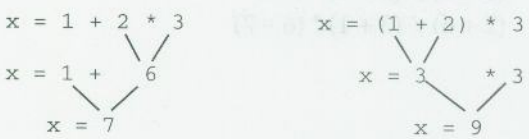


FIGURE 5-4 Parentheses can be used to change the order of operations.

EXERCISE 5-9 ORDER OF OPERATIONS

1. Open the file **Order.java** from the student data files.
2. Look at the source code and try to predict the program's output.
3. Run the program and see if your prediction is correct.
4. Close the source code file.

SECTION 5.2 QUESTIONS

SHORT ANSWER

1. Write a statement that increments a variable `m` using the increment operator.
2. If the value of `i` is 10 before the statement `j = i++;` is executed, what is the value of `j` after the statement is executed?
3. If the value of `i` is 4 before the statement `j = --i;` is executed, what is the value of `j` after the statement is executed?
4. What will the value of `j` be after the statement `j = 3 + 4 / 2 + 5 * 2 - 3;` is executed?
5. What will the value of `j` be after the statement `j = (3 + 4) / (2 + 5) * 2 - 3;` is executed?

PROBLEM 5.2.1

1. Write a program that declares an integer named `up_down` and initializes it to 3.
2. Have the program print the value of `up_down` to the screen.
3. Have the program increment the variable and print the value to the screen.
4. Add statements to the program to decrement the variable and print the value to the screen again.
5. Save the source code as **UpDown.java,** compile it, and run it.

PROBLEM 5.2.2

Write a program that evaluates the following expressions and prints the different values that result from the varied placement of the parentheses. Store the result in a float variable to allow for fractional values. Save the source code file as **Paren.java**.

$2 + 6 / 3 + 1 * 6 - 7$
$(2 + 6) / (3 + 1) * 6 - 7$
$(2 + 6) / (3 + 1) * (6 - 7)$

CHAPTER 5, SECTION 3

How Data Types Affect Calculations

Java allows you to mix data types in calculations (such as dividing a float value of 125.25 by an integer such as 5). Although you should avoid it whenever possible, this section will show you the way to do it should you need to.

You learned in Chapter 4 that each data type is able to hold a specific range of values. When performing calculations, the capacity of your variables must be kept in mind. It is possible for the result of an expression to be too large or too small for a given data type.

MIXING DATA TYPES

Sometimes you may need to perform operations that mix data types, but doing so is less than desirable. In fact, many programming languages do not allow it. But if you have to mix data types, it is important that you understand how to do it properly.

Java can automatically handle the mixing of data types using a process called *promotion.* Also, remember from Chapter 4 that you can direct the compiler how to handle (or how to treat) the data in your programming by using type casting.

PROMOTION

Consider the program in Figure 5-5. The variable **numberOfPeople** is an integer. The other variables involved in the calculation are floating-point numbers. Before you mix data types, you should understand the way the compiler is going to process the variables.

In cases of mixed data types, the compiler makes adjustments so as to produce the most accurate answer. In the program in Figure 5-5, for example, the integer value is temporarily converted to a float so that the fractional part of the variable **money** can be used in the calculation. This is promotion. The variable called **numberOfPeople** is not actually changed. Internally, the computer treats the data as if it were stored in a float. But after the calculation, the variable is still an integer.

```
// Share.java
class Share
{
  public static void main(String Args[])
  {
  int numberOfPeople;
  float money;
  float share;

  // Number of people with whom to share money
  numberOfPeople = 70;
  // Amount of money to be shared
  money = 7500;
  share = money / numberOfPeople;
  // output results
  System.out.println("Give each person " + share);
  }
}   // end of class Share
```

FIGURE 5-5
The compiler may promote variables used in calculations.

EXERCISE 5-10 MIXING DATA TYPES

1. Open the file **Share.java** from the student data files. The program from Figure 5-5 appears in your editor. Save the program to your student folder or disk.
2. Compile and run the program and observe how the mixed data types function.
3. Close the source file.

Promotion of the data type can occur only while an expression is being evaluated. Consider the program in Figure 5-6. Some compilers may not even allow this code to run without type casting the calculations.

```
// TESTPROMO.java
class TestPromo
{
  public static void main(String Args[])
  {
  int answer, i;
  float x;
  i = 3;
  x = 0.5;
  answer = x * i;
  System.out.println("answer = " + answer);
  }
}     // end of class TestPromo
```

FIGURE 5-6
Mixing data types can cause incorrect results.

EXERCISE 5-11 MIXING DATA TYPES

1. Enter the program shown in Figure 5-6 and save it as **TestPromo.java**.
2. Compile the program. Was your compiler able to compile the program without errors? If not, what error did you receive?
3. Make the following changes to the program and compile it again.

```
x = (float) 0.5;
answer = (int) (x * i);
```

4. Run the program. Is the answer correct?

The variable `i` is promoted to a float when the expression is calculated, which gives the result 1.5. But then the result is stored in the integer variable answer. You are unable to store a floating-point number in space reserved for an integer variable. The floating-point number is ***truncated,*** which means the digits after the decimal point are dropped. The number in the answer is 1, which is not correct.

OVERFLOW

Overflow is the condition in which an integer becomes too large for its data type. The program in Figure 5-7 shows a simple example of overflow. The expression `j = (short) (2 * i);` results in a value of 64000, which is too large for the short data type. Note that just to make this code compile, we had to type cast it. Java compilers provide intensive protection against these kinds of mistakes, allowing you to override the compiler only if that is what you really meant to do. In your code here, you were really trying to make a mistake, so you overrode the expression with a type cast. Be careful how you use this when you are writing code in which you do not want mistakes.

```
// start of class OverFlow.java
class OverFlow
{
  public static void main(String args[])
  {
  short i, j;

  i = 32000;
  j = (short)(2 * i);

  System.out.println("j = " + j);
  }
}     // end of class OverFlow
```

FIGURE 5-7
The result of the expression will not fit in the variable `j`.

EXERCISE 5-12 OVERFLOW

1. Enter, compile, and run the program in Figure 5-7. Notice that the calculation resulted in an overflow and an incorrect result.
2. Change the data type from short to long. Compile and run again. This time the result should not overflow.
3. Save the source file as **OverFlow.java** and then close the file.

UNDERFLOW

Underflow is similar to overflow. Underflow occurs with floating-point numbers when a number is too small for the data type. For example, the number 1.5×10^{-44} is too small to fit in a standard float variable. A variable of type double, however, can hold the value.

FLOATING-POINT ROUNDING ERRORS

Using floating-point numbers can produce incorrect results if you fail to take the precision of floating-point data types into account.

When working with very large or very small floating-point numbers, you can use a form of exponential notation, called ***"E" notation,*** in your programs. For example, the number 3.5×10^{20} can be represented as 3.5e20 in your program.

You must keep the precision of your data type in mind when working with numbers in "E" notation. Look at the program in Figure 5-8. At first glance, the two calculation statements appear simple enough. The first statement adds 3.9×10^{10} and 500. The second one subtracts the 3.9×10^{10}, which should leave the 500. The result assigned to `y`, however, is not 500. Actual values vary depending on the compiler, but the result is incorrect whatever the case.

The reason is that the float type is not precise enough for the addition of the number 500 to be included in its digits of precision. So when the larger value is subtracted, the result is not 500 because the 500 was never properly added.

```
// start of class UnderFlow
class UnderFlow
{
  public static void main(String args[])
  {
  System.out.println("Program Begins");
  float x,y;
  x = (float)(3.9e10 + 500.0);
  y = (float)(x-3.9e10);
  System.out.println("y = " + y);
  }
}     // end of class UnderFlow
```

FIGURE 5-8
The precision of floating-point data types must be considered to avoid rounding errors.

EXERCISE 5-13 FLOATING-POINT PRECISION

1. Enter, compile, and run the program in Figure 5-8. See that the result in the variable `y` is *not* 500. Note here that we have to type cast the expressions as floats to conduct this experiment.
2. Change the data type of `x` and `y` to double and run again. The increased precision of the double data type should result in the correct value in `y`, and you should not need to type cast the expressions this time.
3. Save the source code file as **UnderFlow.java** and close the source code file.

SECTION 5.3 QUESTIONS

SHORT ANSWER

1. What is the term that means the numbers to the right of the decimal point are removed?
2. What is the typecast operator that changes a variable to a double?
3. Define overflow.

4. Define underflow.
5. What is "E" notation?
6. How would you write 6.9×10^8 in "E" notation?

CHAPTER 5, SECTION 4

Using The Random Number Class

The ability to generate random numbers has many applications in programming. Random numbers are used in security encryption, games, and mathematical approximations. The Java library has a Random number class that allows you to generate and manage random numbers. Consider the code in Figure 5-9.

```
// start of class MakeRandom
import java.util.Random;
class MakeRandom
{
  public static void main(String args[])
  {
  System.out.println("Program begins");

  // GenerateOne seed is present date on computer
  Random GenerateOne = new Random();

  // GenerateTwo seed is the number given
  Random GenerateTwo = new Random(123456789);

  // create first double and output
  double First = GenerateOne.nextDouble() + 2.5e10;
  System.out.println("First = " + First);

  // create second double and output
  double Second = GenerateTwo.nextDouble() * 7.5e10;
  System.out.println("Second = " + Second);

  double Third;
  Third = First * Second;
  System.out.println("Third = " + Third);
  }
}     // end of class MakeRandom
```

FIGURE 5-9
Java has a standard class for generating random numbers.

There are a few things to notice about the program in Figure 5-9. First, a new line is added at the top of the code. To bring in the Random class, the compiler must be told to import that part of the Java utilities class, since this class is not automatically built in to the code like the simple mathematical operations were. Note in addition that you can also write `import java.util.*;`. This will bring

in all of the utilities, including the Random class, but it makes your code unnecessarily large if you do not use the other utilities classes.

There are actually two random number generators available to you. One is part of the Math class. A more extensive random number generator is found in the Random class. The Random class offers more ability to change the ***seed*** of the generator. The seed of the generator gives it a numeric starting point from which to calculate a random number. You can initialize the random number generator with or without a seed. As you can see from the program, if no seed is provided, the compiler provides the seed (from its clock).

The commands used to initialize the random number generator are called constructors. A ***constructor*** is the function (or part of the program) that *constructs* the object based on the "rules" of the class. In this case, the same constructor is used in two different ways: one with a seed and one without. This is called function or method ***overloading*** and makes some operations, such as constructing, very easy. It is possible to have several constructors in a class, and like the one used here, allows you to construct a new class with different initial conditions. There are some names for the different kinds of constructors, such as "default," "copy," and so on, but you will be reading about those in Chapter 10.

As you can see, using the class is very easy. Once again, you implement the process of creating a new object of the class—in this case, two new objects were created. Each one has the ability to generate different kinds of random numbers, but this program needed a double, so the method name was **`nextDouble`**. The term *method* refers to the segments of code that perform an action within an object. The method **`nextDouble`** is said to be a ***member*** of the class Random, and we used the ***dot operator (.)*** to identify that the first **`nextDouble`** was working for the **`GenerateOne`** object, and the second **`nextDouble`** was working for the **`GenerateTwo`** object. Again, remember that an object knows how to take care of itself and do its appropriate work, so **`GenerateOne`** and **`GenerateTwo`** both knew how to send back the next available random number using the member method **`nextDouble`**.

Finally, note that each number is multiplied by a large floating point number. It is common for random number generators to generate fractional quantities (i.e., decimal numbers between 0 and 1), so the program made these numbers much larger from the start. Then, the two numbers are multiplied together and assigned to the the third number, cleverly named **`Third`**. Work your way through this next exercise and make sure you understand what is happening.

EXERCISE 5-14 RANDOM NUMBER GENERATOR

1. Analyze the code in Figure 5-9. What is happening when the two new objects are created, and what does each piece of code do? Make sure you can see what the output would contain before you actually try to run it.
2. Now enter the code in your compiler, save it as **MakeRandom.java**, and run it. What are the results? Are they what you expected?
3. Is there a possibility of overflow here? In a sentence or two, argue for why or why not.

KEY TERMS

++ operator	incrementing
-- operator	member
arithmetic operators	modulus operator
assignment operator	order of operations
compound operators	overflow
constructor	overloading
decrementing	promotion
dot operator (.)	seed
"E" notation	truncate
expression	underflow

SUMMARY

- The assignment operator (=) changes the value of the variable to the left of the operator to the result of the expression to the right of the operator.
- You can initialize multiple variables to the same value in a single statement.
- The arithmetic operators are used to create expressions.
- The modulus operator (%) returns the remainder of integer division.
- Expressions can be placed in output statements.
- Dividing by zero generates an exception in Java, or it may generate a NaN (not a number), or it may generate positive or negative #INF (infinity) depending on where it comes into the error.
- Spaces can be placed around all operators but are not required in most cases; however, they can improve readability and should be used where possible.
- The ++ and -- operators increment and decrement arithmetic variables, respectively.
- The placement of the ++ and -- operators becomes important when the operators are used as part of a larger expression or in an output statement.
- Java calculations follow an order of operations.
- Java allows data types to be mixed in calculations. However, because it is not a good idea, the language will commonly require typecasting to override its own protection from making mixed data type mistakes. When Java is allowed to handle mixed data types automatically, variables are promoted to other types.
- Overflow, underflow, and floating-point rounding errors can occur if you are not aware of the data types used in calculations.

- The Random class generates random numbers for use by programs.
- When classes are needed, most commonly they must be imported from the Java library files.
- There may be more than one way for a class to take an action, such as constructing a new object. Java allows for the same member method name to be reused when it is accompishing the same task—the difference will be in the types and numbers of parameters.
- A member method is simply one that does the work an object needs done. Since there can be more than one object from the same class, they may all use the same method. However, the way to identify which object is in charge of that method at that moment is by noticing the name of the object on the left side of the dot operator (.).

PROJECTS

PROJECT 5-1 • MATHEMATICS

Write a program that calculates the area of an ellipse. Locate the formula for the calculation. Assign variables to appropriate values to test the program and output the area. Save the program as **Project5_1.java** to student disk or folder.

PROJECT 5-2 • MATHEMATICS

Write a program that converts degrees to radians using the formula below. Save the program as **Project5_2.java**.

radians = degrees / 57.3

PROJECT 5-3 • SPEED

Write a program that converts miles per hour to feet per second. Save the program as **Project5_3.java**.

6 I/O and Exception Handling

OBJECTIVES

- *Explain how to get console input from users.*
- *Implement the exception-handling process of console input.*
- *Determine how to develop programs with user input and output.*
- *Discuss the multilevel exception-handling process.*

Overview

The programs you have worked with up to this point have created output, such as writing text or numeric values to the screen. But getting input from the user is a little more complicated. To properly obtain input from the user, a program must incorporate a special program error management process called ***exception handling.*** Exception handling is a system to deal with situations that are ***exceptions*** to the normal flow of operation. Exception handling is sometimes called simply *error handling*. Because input and exception handling go together, you will learn about both in this chapter. Keep in mind, however, that the uses for exception handling are many.

CHAPTER 6, SECTION 1

Casting for Input

The console input process for Java is not as simple as it is in many other languages. However, the Java way has some advantages. For example, many languages can be set to accept certain data and then use that data for the program's calculations. But what happens if your user enters the wrong type of data? For instance, what happens if the user enters a *B* when the program needs a number? The answer depends on a variety of factors, yet often the program simply cannot handle the wrong type, and an unplanned shutdown occurs. In other words, the program might "crash."

Note

The code used in this chapter for accepting input is useful only in the Java console environment. In a later chapter, you will see how you can accept input through graphical user interfaces and when running a Java program through a browser.

Another situation that can occur when a user provides input is the problem of inappropriate or unrealistic data. Suppose you have a program to calculate the distances that an airplane can fly. Most commercial airliners do not fly much faster than 500 to 600 miles per hour. However, your user enters 5,000 miles per hour or some other number that is totally inappropriate. This can lead to calculations whose results are much larger than your declared variables can hold. Inappropriate input can also lead to a division by zero.

These kinds of problems have always plagued computer programmers. If an error does require a program to halt, the ability to exit gracefully and then restart is important. You probably at some time have been running a program and received a message that an illegal operation or system error has occurred. When this happens, you can usually return to the operating system and restart the program. However, that is not always the case. Sometimes the problem that invoked that message might have crashed the operating system as well. This makes exception handling even more important. The term *exception handling* is used because we assume that normally the program will run without error, with the *exception* of certain problems that may arise.

Refer to the code segment in Figure 6-1. This is an example of the simplest kind of input statement that can be used in the Java console environment.

Although this process is longer than what you might have done in some other languages, it is much safer. If anything goes wrong in the course of entering data to

```
import java.lang.StringBuffer;
import java.io.IOException;
class MyInput
{
  public static void main(String Args[])
  {
  System.out.println("Program begins");

  int noMoreInput = -1;
  char enterKeyHit = '\n';
  int InputChar;

  StringBuffer InputBuffer = new StringBuffer(30);

  System.out.println("Please enter a one word name:");
  try
    {
    InputChar = System.in.read();

    while(InputChar != noMoreInput)
      {
      if((char) InputChar != enterKeyHit)
        {
        InputBuffer.append((char) InputChar);
        }
      else
        {
        InputBuffer.setLength( InputBuffer.length() -1 );
        break;
        }
      InputChar = System.in.read();
      }
    }
  catch(IOException IOX)
    {
    System.err.println(IOX);
    }

  System.out.println("You entered: " + InputBuffer.toString());
  }
}     // end of class myInput
```

FIGURE 6-1
The input process brings in characters to form a line of text. When the **Enter** key is pressed, the line of text is then printed to the screen.

this program, Java will protect the user and the program by shutting down and ***throwing an exception.*** The phrase *throwing an exception* means that the program will report the exception and try to explain what went wrong if it can.

EXERCISE 6-1

THE FIRST INPUT PROCESS

1. Start your text editor and enter the program shown in Figure 6-1. Save the program to your course files folder or disk as **MyInput.java**. Compile and run the program.
2. Provide a variety of characters as input. Run the program a number of times.

3. Enter your first, middle, and last names with spaces between the names and observe what happens.
4. Leave the program on your screen as the code is analyzed below.

The program you ran in Exercise 6-1 handles almost any attempt you make to crash it. Let's analyze the code to see how it works.

First, you copy, or import, the StringBuffer and IOException classes from the Java library so you can use them. We'll use the StringBuffer to hold data coming in from the keyboard. The IOException class will do our error handling.

```
import java.lang.StringBuffer;
import java.io.IOException;
```

Next, you declare your class and your main method.

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

The code below shows that three variables are declared, along with a **StringBuffer** object named **InputBuffer**. At this point, we are most interested in the **StringBuffer** object. A buffer is like an escalator. The escalator holds people while it moves them upstairs where they can get off in a different place than where they got on. A **StringBuffer** object holds data in a line (called a *stream*) until the data is removed from the buffer. Initially, we have declared the **InputBuffer** to hold up to 30 characters.

```
int noMoreInput = -1;
char enterKeyHit = '\n';
int InputChar;

StringBuffer InputBuffer = new StringBuffer(30);
```

Next, the program prompts the user for some information. In this case, the user is asked to enter a one-word name. The prompt does not cause the program to stop and wait for input. However, the first line of code within the ***try*** process will pause the program until the **Enter** key is pressed (see the code below). The majority of the code within the try process will test what the user has entered for success to make sure it worked. If something does not work, the ***catch*** process will take the actions necessary to deal with the exception. In this case, it will print out a standard error message related to input and output processes because you have not asked it to do anything else. If no error occurs in the try block, the catch block will be ignored.

```
System.out.println("Please enter a one word name:");

try
  {
  InputChar = System.in.read();

  while(InputChar != noMoreInput)
    {
    if((char) InputChar != enterKeyHit)
      {
```

```
        InputBuffer.append((char) InputChar);
        }
      else
        {
        InputBuffer.setLength( InputBuffer.length() -1 );
        break;
        }
      InputChar = System.in.read();
      }
    }
  catch(IOException IOX)
    {
     System.err.println(IOX);
    }
```

The *try* and *catch* keywords are easy to remember because of how they are used. User input that may generate an exception is placed in the braces under the *try* keyword because we want to *try* that input and see if all goes well. If not, we will *catch* the problem before it causes other problems.

Inside the try block, the program accepts a character. The character is stored in a variable of type int because the input stream stores integers. As the characters are placed into the **InputBuffer**, the data type will be changed without harming the data.

Although you will study the use of *while* and *if* in the next two chapters, you can see roughly what happens next. The code says "While the incoming character is not equal to –1 (noMoreInput is equal to –1), keep doing what is between the next pair of braces."

The line that begins with the keyword *if* plays an important role next. The code says "if the incoming character is not the **Enter** key (meaning the user has not hit the **Enter** key yet), then take the newest incoming character and add it to the end of the InputBuffer." This is a process called ***appending.*** The *else* part defines what will happen if the incoming character *is* the **Enter** key. The *break* keyword causes the *while* loop to end when the user presses **Enter.**

Note

Using an identifier such as noMoreInput to equal –1 is one way to ***self-document****, or explain, what the code is doing.*

```
while(InputChar != noMoreInput)
  {
  if((char) InputChar != enterKeyHit)
    {
    InputBuffer.append((char) InputChar);
    }
  else
    {
    InputBuffer.setLength( InputBuffer.length() -1 );
    break;
    }
  InputChar = System.in.read();
  }
```

The line that reads **InputChar = System.in.read();** is placed inside the loop so that the loop may continue to accept characters until **Enter** is pressed.

Note

You can avoid including the statement that reads in the next character twice by rewriting the code. However, to make the code more clear, the program shown here repeats the read statement.

EXERCISE 6-2 RUN THE PROGRAM AGAIN

1. Now that you have analyzed the program, run the program again.
2. Enter the entire sentence below as input:

```
Exception handling is a concept used in languages other than Java.
```

3. The sentence entered in step 2 is greater than 30 characters long. This does not crash the program because the InputBuffer protects itself from such problems by resizing itself if necessary.

SECTION 6.1 QUESTIONS

SHORT ANSWER

1. Why do some of the Java classes have to be imported into your program?
2. Why would you self-document certain variables?
3. Explain in a sentence or two how the two keywords *try* and *catch* work together.
4. What does it mean to append a character to a string?
5. Three code lines down from the try, the word *char* is in parentheses. What does this accomplish, and why is it done at this point? **Hint:** the `'\n'` is considered to be a character.

CHAPTER 6, SECTION 2

Using Methods

As programs get larger, it is a good idea to break the source code into smaller, more manageable, and reusable blocks of code called *methods*. In this section, you will see how the program used to input data in the previous section could be divided into methods. You will learn more about methods in a later chapter. At this time, it is only important to learn how a program can be divided into logical parts using methods.

SEPARATING CODE INTO METHODS

A ***method*** is a block of code that performs some function. In other languages, you may have seen blocks of code referred to as functions or procedures. In Java, methods are often referred to as ***member functions***. The term *member function* comes from the fact that the function is a member of a certain class.

In the exercise that follows, the program from the previous section is divided into two pieces: the main function and a method for getting input from the user. The advantage to separating that code into a separate method may not immediately be apparent. By separating the code, however, the input method may be reused in other programs.

EXERCISE 6-3 USING METHODS

1. Open **NewMethod.java** from the student data files. The source code shown in Figure 6-2 appears.
2. Leave the source code open on your screen as you read the paragraphs that follow.

```
// NewMethod.JAVA
import java.lang.StringBuffer;
import java.io.IOException;

class NewMethod
{
  public static void main(String args[])
  {
  System.out.println("Program begins");

  System.out.print("Please enter a one word name:");

  String InputString = GetConsoleString();
  System.out.println("You entered: " + InputString);
  System.out.println("Program paused, Press the Enter key to continue");
  InputString = GetConsoleString();
  }

  public static String GetConsoleString()
  {
  int noMoreInput = -1;
  char enterKeyHit = '\n';
  int InputChar;
  StringBuffer InputBuffer = new StringBuffer(30);

  try
    {
    InputChar = System.in.read();

    while(InputChar != noMoreInput)
      {
      if((char) InputChar != enterKeyHit)
        {
        InputBuffer.append((char) InputChar);
        }
      else
        {
        InputBuffer.setLength( InputBuffer.length() -1 );
        break;
        }
      InputChar = System.in.read();
      }
    }
  catch(IOException IOX)
    {
    System.err.println(IOX);
    }

  return InputBuffer.toString();

  }     // end of method GetConsoleString

}     // end of class NewMethod
```

FIGURE 6-2
Creating an input method can simplify data input in programs.

Notice that the code in Figure 6-2 is not any smaller or much different from the code in the previous section. However, the main method is significantly smaller than it was. It now has only six lines, and the input work is being done elsewhere.

Let's look at some of the details of the code in Figure 6-2. Consider the statement below, which appears in the main method. The statement declares an object of type String named InputString. The object is immediately initialized by invoking the **GetConsoleString()** method. The main method is said to "call" the **GetConsoleString()** method.

```
String InputString = GetConsoleString();
```

In the **GetConsoleString()g** method (shown below), we see that, like the main method, it is a public and a static method. However, unlike the main method, which returns nothing or void, the **GetConsoleString()** method returns a String.

```
public static String GetConsoleString()
{
int noMoreInput = -1;
char enterKeyHit = '\n';
int InputChar;
StringBuffer InputBuffer = new StringBuffer(30);

try
  {
  InputChar = System.in.read();

  while(InputChar != noMoreInput)
    {
    if((char) InputChar != enterKeyHit)
      {
      InputBuffer.append((char) InputChar);
      }
    else
      {
        InputBuffer.setLength( InputBuffer.length() -1 );
      break;
      }
    InputChar = System.in.read();
    }
  }
catch(IOException IOX)
  {
  System.err.println(IOX);
  }

return InputBuffer.toString();

}     // end of method GetConsoleString
```

Think of your methods as couriers. You can send them out to get something, or to process something you give them, and then they will come back and hand you whatever you needed from them. Think of the **GetConsoleString** method as a courier who runs from your program out to the keyboard, gathers up whatever comes in, and returns it to your program. Now, look at the code again. At the top of the **GetConsoleString** method, the *String* keyword tells you that the

method will be returning a string. Also, notice the return statement at the bottom of the method. The return statement returns the contents of the InputBuffer to the function that called the method.

Now that the `GetConsoleInput` method has been written as a separate member function, it may be used every time input is required.

Note

Depending on the compiler you are using, the output of your programs may appear in a window and then disappear before you have enough time to read the contents of the window. By placing a call to the `GetConsoleString` *method at the end of the code, you can pause the program so you can see the output while it waits for input. Of course, since there is no variable to receive the data from the* `GetConsoleString` *"courier," nothing is transferred when you enter data. However, the desired result of pausing the output before the window closes is achieved. This method will be added to most of the programs in the text after this point.*

EXERCISE 6-4 RUNNING THE PROGRAM

1. Save the program on your screen as **NewMethod.java** to your course files folder or disk. Compile and run the program. It should operate exactly like the previously compiled program.
2. When the program terminates, close the source code file.

SECTION 6.2 QUESTIONS

SHORT ANSWER

1. From where does the term *member function* come?
2. What is another term for a member function?
3. What was one advantage of separating the `GetConsoleInput` code from the rest of the program?
4. What type of data is returned by the `GetConsoleInput` method?
5. Why might you place an additional call to `GetConsoleInput` at the end of a program before the program terminates?

CHAPTER 6, SECTION 3

More About Exceptions

You learned earlier that exception handling is applied to more than just getting input. In this section, you will see an example of another way that exceptions can be used. You will also learn about the concept of inheritance and how the object-oriented programming feature of inheritance can make exception handling easier to "handle."

ANOTHER EXCEPTIONAL EXAMPLE

Analyze the code segment in Figure 6-3. The program accepts two numbers from the user. Then, using the exception-handling feature, the program "tries" to divide the two numbers. Exception handling is implemented here because of the possibility that the division may be by zero. In most cases, the division will occur without error, with the *exception* of division by zero.

```
class DivideEm
{
  public static void main(String Args[])
  {
  System.out.println("Please enter one number:");

  String InputString = GetConsoleString();

  int NumberOne = Integer.parseInt(InputString);

  System.out.println("Please enter another number:");

  InputString = GetConsoleString();
  int NumberTwo = Integer.parseInt(InputString);

  try
    {
    int NumberThree = NumberOne / NumberTwo;
    System.out.println("NumberOne divided by NumberTwo = "
        + NumberThree);
    }
  catch(Exception E)
    {
    System.err.println("General Exception Thrown");
    }
  System.out.println("Press Enter key to continue ");
  GetConsoleString();

  }     // end of main

// DEFINITION OF GetConsoleString goes here

}     // end of class DivideEm
```

FIGURE 6-3 Exception handling can be used to trap errors caused by dividing by zero.

EXERCISE 6-5 TESTING THE EXCEPTION

1. Enter the source statements for the program shown in Figure 6-3. Enter the **GetConsoleString** definition from Figure 6-2 at the location indicated in

Figure 6-3. Add the two statements shown below at the top of the program and save the source file as **DivideEm.java**:

```
import java.lang.StringBuffer;
import java.io.IOException;
```

2. Compile and run this code trying different numbers to be divided.
3. Now try dividing a number by zero by entering 0 (zero) at the second prompt. Note the exception that is thrown.
4. Leave the program on your screen for the next exercise.

Sometimes it is desirable to get more specific about what kind of exception was thrown. The current code has a simple catch section (shown below) that catches any exception thrown. This code catches a general exception.

```
catch(Exception E)
  {
  System.err.println("General Exception Thrown");
  }
```

To be more specific, we can add code to catch a specific kind of exception. The code below shows how two catch blocks may be used to give more detailed information about what went wrong. If the exception thrown was an arithmetic exception, the first catch block will catch the exception and provide the more specific message. If the exception was not caused by an arithmetic problem, the general exception catch block will catch the exception.

```
catch(ArithmeticException AE)
  {
  System.err.println("Divide By Zero Exception Thrown");
  }
catch(Exception E)
  {
  System.err.println("General Exception Thrown");
  }
```

Note

Some Java compilers will produce a warning if you place a general exception catch before the specific ones.

Note

In simple programs such as the one here, finding the cause of the crash is easy to do, with or without exception handling. In more sophisticated programs, however, the causes of the exception could be many, and it might be difficult to determine what conditions caused the crash. Having more specific catch blocks might help determine the cause of the problem.

When using multiple catch blocks, place the general exception catch block last. The program will use the first catch block that can handle the type of exception that has been thrown. The general exception catch block will handle any kind of exception. Therefore, catch blocks that follow the general exception catch block will never be used. Be sure always to place the specific exception catch blocks ahead of the general exception catch block.

EXERCISE 6-6 MULTIPLE CATCH BLOCKS

1. Modify the program on your screen to include the two catch blocks: one for arithmetic exceptions and one for general exceptions.
2. Compile and run the program.
3. Provide input that will lead to division by zero. What happens? The **`ArithmeticException`** recognizes the problem first and catches it before it can get to the general exception.
4. Swap the two catch blocks so that your **`ArithmeticException`** process, including braces and output, follows the catch block for the general exception.
5. Compile the program again. If your compiler did not report any errors, run the program again and cause a division by zero again. This time the general exception code catches the exception before the exception-specific catch block is reached.
6. Swap the two catch blocks again to return them to the proper order.
7. Compile, run, and test the program.
8. Save and close the program.

EXCEPTIONS AND INHERITANCE

Now that you have some experience with exception handling, it is a good time to introduce an important topic of object-oriented programming: inheritance. ***Inheritance*** is the feature of object-oriented programming that allows you to take code already developed and debugged and build on it to create something new. Before we look at how inheritance is at work in exception handling, let's look at some real-world examples of inheritance.

A good example of how inheritance can make programming easier can be found in the graphical user interfaces in use on most computers today. A window is an important component of a graphical user interface. Therefore, code to create, position, move, and resize windows is an important part of the operating system. When it is time to create a dialog box, there is no need to start from scratch. A dialog box is a particular type of window. Every time a dialog box appears on the screen, you are witnessing an example of inheritance. Therefore, the properties of a window can be inherited and extended to create a dialog box. That is what inheritance is all about.

For another example, let's go back to the OOP County Fair. Suppose that you have designed a set of rules (a class) for a retail outlet (a store) at the fair. Once you have designed the RetailOutlet class, you know how to accept cash from a customer, how to work the cash register, how to do the bookkeeping, and so on. But now, someone wants to open up a little ice cream store, with banana splits and milkshakes. Knowing that an ice cream store is a kind of RetailOutlet means that you can use the rules made for RetailOutlets and add some rules specifically addressing how to make banana splits and milkshakes. In fact, in object-oriented programming, we say that the IceCreamStore can be a "child" of the "parent" RetailOutlet store. We would say that the IceCreamStore is a RetailOutlet. The IceCreamStore class inherits the properties of a RetailOutlet.

Notice this relationship does not go both ways. RetailOutlets can sell practically anything. IceCreamStores, however, sell only ice cream products. Therefore,

it is appropriate to say that an IceCreamStore is a RetailOutlet, but it is not appropriate to say that a RetailOutlet is an IceCreamStore.

With this in mind, consider the Exception class and its children in Java. The ***Exception classes*** in Java provide the code necessary to handle the different types of exceptions. The Exception classes build on one another using inheritance. Figure 6-4 shows a small sample of the Exception classes. The compiler you are using should have a reference in chart form, or in CD-ROM form, or in the paper documentation, that shows how all the Exception classes are tied together. On the basic chart of Java classes, there are more than 30 Exception classes.

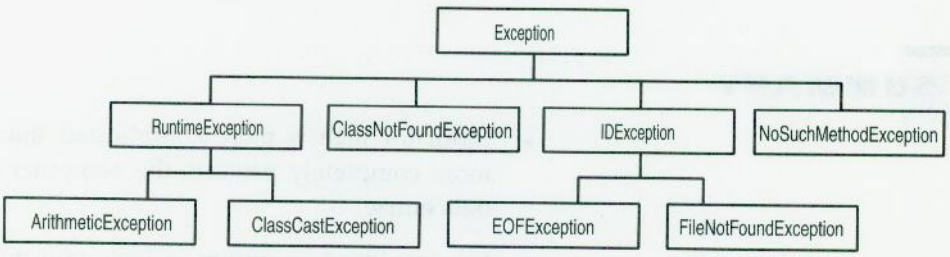


FIGURE 6-4
The Exception classes use inheritance to build on code that already exists.

The parent class for handling exceptions is called Exception. Other classes are derived from the Exception class. The RuntimeException class, for example, inherits the properties of the Exception class. The ArithmeticException, which we used in the earlier example, is a RuntimeException, which in turn is an Exception.

The chart also shows that the IOException class, which we used earlier in this chapter to capture input or output problems, is not a RuntimeException. Both the IOException class and the RuntimeException class inherit properties directly from the Exception class.

Note

There is practically no end to the number of levels of inheritance as long as it is appropriate for your new class to use the classes that act as parents.

SECTION 6.3 QUESTIONS

SHORT ANSWER

1. Why is the order of catch blocks important?
2. What type of catch block will catch any exception?
3. If given a bird class and a pelican class, which would be the parent class?
4. Explain why there are so many different kinds of Exceptions available in Java.
5. Explain why the ArithmeticException class is a child of the RuntimeException class.

KEY TERMS

appending	member function
catch	method
exception	self-document
Exception classes	stream
exception handling	throwing an exception
inheritance	try

SUMMARY

- Input for Java is more complicated than other languages because Java more completely protects the computer and program against improper data entry.
- For any input operation in Java, you must use the try/catch process to capture any potential inputting problems.
- Code that may generate an error is placed in a try block. If an error occurs in the try block, an exception is thrown. A catch block catches the exception and deals with it, either by printing a message or in some other way.
- You can have more than one catch block to catch different kinds of exceptions. A general exception catch block will catch any exception.
- There are more than 30 basic Java Exception classes.
- Exceptions, like all other classes in object-oriented programming, are derived, or inherited, from simpler, more general classes in a family-like hierarchy.

PROJECTS

PROJECT 6-1

Write a program that gets the time of day from the user and outputs the number of seconds since last midnight. Save the program to your student disk as **Project6_1.Java.** Compile and run the program. Close the source file.

PROJECT 6-2

Write a program that gets the month and day from the user and outputs the approximate number of minutes since the first of the year. Save the program to your student disk as **Project 6_2.Java.** Compile and run the program. Close the source file.

PROJECT 6-3

Write a program that gets from the user a first name, a middle initial, and a last name and outputs all three together as first - middle - last. Try using String objects to do this. Save the program to your student disk as **Project6_3.Java.** Compile and run the program. Close the source file.

7 Decision Making in Programs

OBJECTIVES

- *Describe how decisions are made in programs.*
- *Understand how true and false are represented in Java.*
- *Use relational operators and logical operators.*
- *Describe and program object-oriented comparisons.*
- *Use the if, if/else, nested if, and switch structures.*

Overview

One of the most notable powers of computers is that they can do things very quickly. However, computers would not be of significant value to us if they could only do exactly the same thing over and over. One of the more powerful capabilities of computers is the ability to choose from more than one path of action based on some condition (for example, if the left mouse button is clicked, do this, but if the right mouse button is clicked, do that). This is a process called ***branching***. In this chapter, you will learn how branching is accomplished in Java programs. You will learn about the building blocks of computer decision making and about programming structures that cause different parts of a program to be executed based on decisions made within the program.

CHAPTER 7, SECTION 1

The Building Blocks of Decision Making

When you make a decision, your brain goes through a process of comparisons. For example, when you shop for clothes you compare the prices with those you previously paid. You compare the quality to other clothes you have seen or owned. You probably compare the clothes to what other people are wearing or what is in style. You might even compare the purchase of clothes to other possible uses for your money.

Although your brain's method of decision making is much more complex than the processes a computer is capable of, decision making in computers is also based on comparing data. In this section, you will learn to use the basic tools of computer decision making.

DECISION MAKING IN PROGRAMS

Every program that is useful involves decision making. Although some algorithms progress sequentially from the first to the last instruction, most algorithms branch out into more than one path. At the point where the path branches out, a decision must be made as to which branch to take.

The flowchart in Figure 7-1 is part of an algorithm in which the program is preparing to output a document to the printer. The user enters the number of copies he or she wants to print. To make sure the number is valid, the program verifies that the number of copies is not less than zero. If the user enters a negative number, a message is printed and the user is asked to enter the value again. If the user's input passes the test, the program simply goes on to whatever is next.

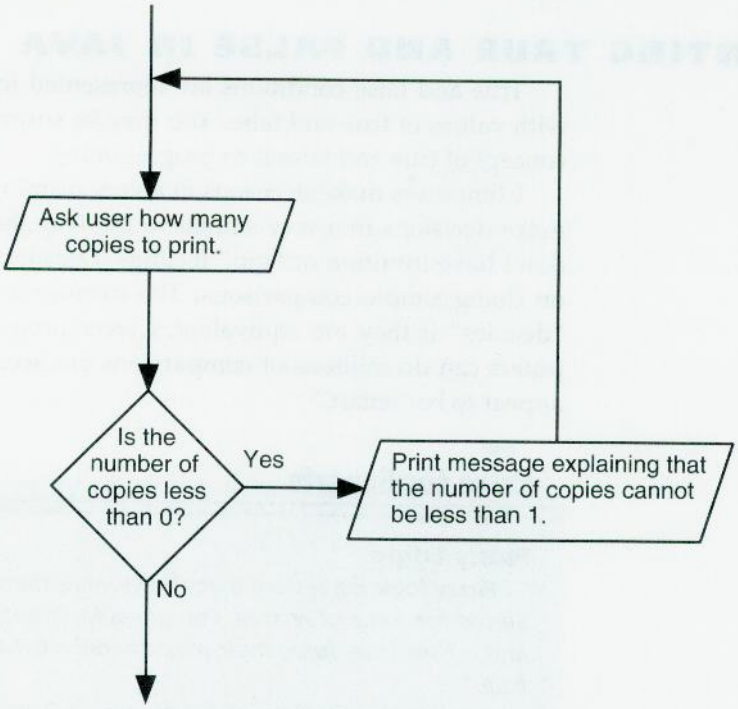


FIGURE 7-1
The decision-making part of this flowchart prevents the program from proceeding with invalid data.

Decisions may also have to be made based on the wishes of the user. The flowchart in Figure 7-2 shows how the response to a question changes the path the program takes. If the user wants instructions printed on the screen, the program displays the instructions. Otherwise, that part of the program is bypassed.

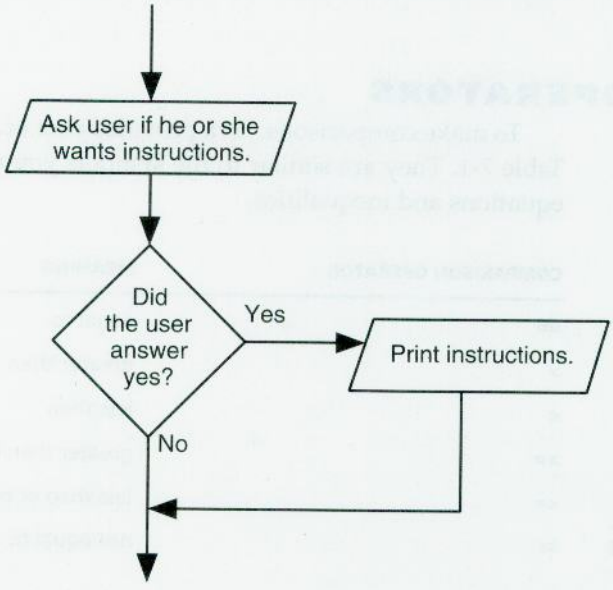


FIGURE 7-2
The path a program takes may be dictated by the user.

The examples in Figures 7-1 and 7-2 show two common needs for decisions in programs. There are many other instances in which decisions must be made. As you do more and more programming, you will use decision making in countless situations.

REPRESENTING TRUE AND FALSE IN JAVA

True and false conditions are represented in Java as Boolean type variables with values of true and false. You may be surprised to learn how important the concept of true and false is to programming.

Computers make decisions in a very primitive way. Even though computers make decisions in a way similar to the way the human brain does, computers don't have intuition or "gut" feelings. Decision making in a computer is based on doing simple comparisons. The microprocessor compares two values and "decides" if they are equivalent. Clever programming and the fact that computers can do millions of comparisons per second sometimes make computers appear to be "smart."

Extra for Experts

Fuzzy Logic

Fuzzy logic is a system that allows more than simply true or false. Fuzzy logic allows for some gray area. For example, instead of simply having a 0 for false and a 1 for true, fuzzy logic might allow a 0.9 as a way of saying "it's probably true."

Practical applications of fuzzy logic include the thermostat on your home's air conditioner. A standard thermostat turns the air conditioner on when the temperature goes above the desired comfort level. This causes the temperature in the house to rise and fall above and below the thermostat setting. A fuzzy logic thermostat could sense that the temperature is rising and turn on the air conditioner before the temperature rises above the desired level. The result is a more stable room temperature and conservation of energy.

COMPARISON OPERATORS

To make comparisons, Java provides a set of *comparison operators,* shown in Table 7-1. They are similar to the symbols you use in math when working with equations and inequalities.

COMPARISON OPERATOR	MEANING	EXAMPLE
==	equal to	i == 1
>	greater than	i > 2
<	less than	i < 0
>=	greater than or equal to	i >= 6
<=	less than or equal to	i <= 10
!=	not equal to	i != 12

TABLE 7-1

The comparison operators are used to create expressions like the examples in Table 7-1. The result of the expression is true if the data meets the requirements of the comparison. Otherwise, the result of the expression is false. For example, the result of 2 > 1 is true, and the result of 2 < 1 is false.

The program in Figure 7-3 demonstrates how expressions are made from comparison operators. The result of the expressions is to be displayed as either a true or false.

```
class Compare
{
  public static void main(String Args[])
  {
  System.out.println("Begin Program");
  int i = 2;
  int j = 3;
  boolean true_false;

  System.out.println("i == 2 => " + (i == 2));
  System.out.println("i == 1 => " + (i == 1));
  System.out.println("j > i  => " + (j > i));
  System.out.println("j < i  => " + (j < i));
  System.out.println("j <= 3 => " + (j <= 3));
  System.out.println("j >= i => " + (j >= i));
  System.out.println("j != i => " + (j != i) + '\n');

  true_false = (j < 4);
  System.out.println("true_false = " + true_false);
  }
}     // end of class Compare
```

FIGURE 7-3 Expressions created using relational operators return either a true or a false.

Note

Be careful when using the >= and <= operators. The order of the symbols is critical. Switching the symbols will result in an error.

EXERCISE 7-1 COMPARISON OPERATORS

1. Load the program **Compare.java** from the student data files and save it to your student disk or folder. The program from Figure 7-3 will appear. Can you predict its output?
2. Compile and run the program.
3. After you have analyzed the output, close the source code file.

LOGICAL OPERATORS

Sometimes it takes more than two comparisons to obtain the desired result. For example, if you want to test to see if an integer is in the range 1 to 10, you must do two comparisons. In order for the integer to fall within the range, it must be greater than 0 and less than 11.

Java provides three *logical operators* for multiple comparisons. Table 7-2 shows the three logical operators and their meaning.

LOGICAL OPERATOR	MEANING	EXAMPLE
&&	and	(j == 1 && k == 2)
||	or	(j == 1 || k == 2)
!	not	result = !(j == 1 && k == 2)

TABLE 7-2

Figure 7-4 shows three diagrams called ***truth tables.*** They will help you understand the result of comparisons with the logical operators *and*, *or*, and *not*.

AND

A	B	A && B
false (0)	false (0)	false (0)
false (0)	true (1)	false (0)
true (1)	false (0)	false (0)
true (1)	true (1)	true (1)

OR

A	B	A || B
false (0)	false (0)	false (0)
false (0)	true (1)	true (1)
true (1)	false (0)	true (1)
true (1)	true (1)	true (1)

NOT

A	!A
false (0)	true (1)
true (1)	false (0)

FIGURE 7-4 Truth tables illustrate the results of logical operators.

Note

The key used to enter the or operator (||) is usually located near the Enter or Return key. It is usually on the same key with the backslash (\).

Consider the following Java statement.

```
in_range = (i > 0 && i < 11);
```

The variable **in_range** is assigned the value true if the value of **i** falls into the defined range, and false if the value of **i** does not fall into the defined range.

The not operator (!) turns true to false and false to true. For example, suppose you have a program that catalogs old movies. Your program uses a Boolean variable named **InColor** that has the value false if the movie was filmed in black and white and the value true if the movie was filmed in color. In the statement below, the variable **i** is set to true if the movie is not in color. Therefore, if the movie is in color, **Black_and_White** is set to false.

```
Black_and_White = !InColor;
```

EXERCISE 7-2 LOGICAL OPERATORS

1. Start your text editor, and enter the following program. Save the source code to your student disk or folder as **logical.java**.

```
public class logical
{
  public static void main (String args[])
  {
  int i = 2;
  int j = 3;
  boolean true_false;

  true_false = (i < 3 && j > 3);
  System.out.println ("The result of (i < 3 && j > 3) is " + true_false);
```

```
        true_false = (i < 3 && j >= 3);
        System.out.println ("The result of (i < 3 && j >= 3) is " + true_false);

        System.out.println ("The result of (i == 1 || i == 2) is " + (i == 1 || i == 2));

        true_false = (j < 4);
        System.out.println ("The result of (j < 4) is " + true_false);

        true_false = !true_false;
        System.out.println("The result of !true_false is " + !true_false);

    }     // end of method main
}     // end of class logical
```

2. Compile and run the program to see the output.
3. After you have analyzed the output, close the source code file.

COMBINING MORE THAN TWO COMPARISONS

You can use logical operators to combine more than two comparisons. Consider the statement below that decides whether it is OK for a person to ride a roller coaster.

```
ok_to_ride = (height_in_inches > 45 && !back_trouble
          && !heart_trouble);
```

In the statement above, **back_trouble** and **heart_trouble** hold the value true or false depending on whether the person being considered has the problem. For example, if the person has back trouble, the value of **back_trouble** is set to true. The not operator (!) is used because it is OK to ride if the person does not have back trouble and does not have heart trouble. The entire statement says that it is OK to ride if the person's height is greater than 45 inches and the person has no back trouble and no heart trouble.

ORDER OF LOGICAL OPERATIONS

You can mix logical operators in statements as long as you understand the order in which the logical operators will be applied. The not operator (!) is applied first, then the and operator (&&), and finally the or operator (||).

Consider the statement below.

```
dog_acceptable = (white || black && friendly);
```

The example above illustrates why it is important to know the order in which logical operators are applied. At first glance, it may appear that the statement above would consider a dog to be acceptable if the dog is either white or black and also friendly. But in reality, the statement above considers a white dog that wants to chew your leg off to be an acceptable dog. Why? Because the and operator is evaluated first and then the result of the and operation is used for the or operation. The statement can be corrected with some additional parentheses, as shown below.

```
dog_acceptable = ((white || black) && friendly);
```

Java evaluates operations in parentheses first just as in arithmetic statements.

EXERCISE 7-3 MIXING LOGICAL OPERATORS

1. Open **logical2.java** from the student data files and save it to your course folder or disk.
2. Compile and run the program to see the effect of the parentheses.
3. Close the source code file.

SHORT-CIRCUIT EVALUATION

Suppose you have decided you want to go to a particular concert. You can only go, however, if you can get tickets and if you can get off work. Before you check whether you can get off work, you find out that the concert is sold out and you cannot get a ticket. There is no longer a need to check whether you can get off work because you don't have a ticket anyway.

Java has a feature called ***short-circuit evaluation*** that allows the same kind of determinations in your program. For example, in an expression such as `in_range = (i > 0 && i < 11);`, the program first checks to see if `i` is greater than 0. If it is not, there is no need to check any further because regardless of whether `i` is less than 11, `in_range` will be false. So the program sets `in_range` to false and goes to the next statement without evaluating the right side of the &&.

Short-circuiting also occurs with the or (||) operator. In the case of the or operator, the expression is short-circuited if the left side of the || is true because the expression will be true regardless of the right side of the ||.

Note

Compilers often have an option to enable or disable short-circuit evaluation.

THE OOP WAY TO TRUTH

Once again, the object-oriented programming paradigm allows you to look at things almost the same way you did before, but with some small changes. You read earlier that OOP tries to treat all things the same so that programming with ints, floats, buttons, or windows will all have common conditions. This leads to easier programming, because if you know how to test for one of these things, you should know how to test for all of them. While you will be learning more about the use of object-oriented Boolean testing in later chapters, look at Figure 7-5 for your first introduction to this process.

Notice that the program uses the member method *equals* to accomplish the Boolean test. More importantly, notice that when `IntOne` is compared to `IntTwo`, `IntTwo` is typecast as an Object. While this is a simple example, the implications are large. In object-oriented programming, everything is treated in your code as one common thing: an object. When you are writing large-scale programs, it means you do not have to worry about which kind of type or class or quantity you are working with. You just work to treat everything the same.

```
class IntegerTest
{
  public static void main(String Args[])
  {
  System.out.println("Begin Program");

  Integer IntOne = new Integer(2);        // Create three integer
  Integer IntTwo = new Integer(3);        // objects and give
  Integer IntThree = new Integer(3);      // them values.

  boolean true_false;

  // Test to see if IntOne is equal to IntTwo
  true_false = IntOne.equals((Object) IntTwo);
  System.out.println("It is " + true_false + " that IntOne equals IntTwo");

  // Test to see is IntTwo is equal to IntThree
  true_false = IntTwo.equals((Object) IntThree);
  System.out.println("It is " + true_false + " that IntTwo equals IntThree");

  }     // end of method main
}     // end of class IntegerTest
```

FIGURE 7-5
Objects of any kind can be compared in OOP.

SECTION 7.1 QUESTIONS

SHORT ANSWER

1. List two comparison operators.
2. Write an expression that returns true if the value in the variable **k** is 100 or more.
3. Write any valid expression that uses a logical operator.
4. Write an expression that returns false if the value in the variable **m** is equal to 5.
5. What is the value returned by the following expression?

```
(((2 > 3) || (5 > 4)) && !(3 <= 5))
```

PROBLEM 7.1.1

In the blanks beside the statements in the program below, write a T or F to indicate the result of the expression. Fill in the answers beginning with the first statement and follow the program in the order the statements would be executed by this method in a running program.

```
public class Prob711
{
  public static void main (String args[])
```

```
{
int i = 4;
int j = 3;
boolean true_false;

true_false = (j < 4);                                    _____

true_false = (j < 3);                                    _____

true_false = (j < i);                                    _____

true_false = (i < 4);                                    _____

true_false = (j <= 4);                                   _____

true_false = (4 > 4);                                    _____

true_false = (i != j);                                   _____

true_false = (i == j || i < 100);                        _____

true_false = (i == j && i < 100);                        _____

true_false = (i < j || true_false && j >= 3);            _____

true_false = (!(i > 2 && j == 4));                       _____

true_false = !true;                                      _____

}
```

CHAPTER 7, SECTION 2

Selection Structures

Programs consist of statements that solve a problem or perform a task. Up to this point, you have been creating programs with sequence structures. ***Sequence structures*** execute statements one after another without changing the flow of the program. Other structures, such as the ones that make decisions, do change the flow of the program. The structures that make decisions in Java programs are called ***selection structures.*** When a decision is made in a program, a selection structure controls the flow of the program based on the decision. In this section, you will learn how to use selection structures to make decisions in your programs. The three selection structures available in Java are the ***if structure***, the ***if/else structure,*** and the ***switch structure.***

USING if

Many programming languages include an if structure. Although the syntax varies among programming languages, the *if* keyword is usually part of every language. If you have used *if* in other programming languages, you should have

little difficulty using *if* in Java. The if structure is one of the easiest and most useful parts of Java.

The expression that makes the decision is called the ***control expression.*** Look at the code segment below. First the control expression (i == 3) is evaluated. If the result is true, the code in the braces that follows the *if* is executed. If the result is false, the code in the braces is skipped.

```
if (i == 3)
  { System.out.println ("The value of i is 3"); }
```

PITFALLS

Remember to be careful of confusing the == operator with the = (assignment) operator. Usage such as `if (i = 3)` *will cause* `i` *to be assigned the value 3 and the code in the braces that follow will be executed regardless of what the value of* `i` *was before the if structure.*

Note

When only one statement appears between the braces in an if structure, the braces are not actually necessary. It is, however, a good idea always to use braces in case other statements are added later.

You can place more than one line between the braces, as in the code segment below.

```
if (YesNo == 'Y')
  {
  System.out.println ("Press Enter when your printer is ready.\n");
  TempIn=5;
  }
```

Figure 7-6 shows the flowchart for an if structure. The if structure is sometimes called a ***one-way selection structure*** because the decision is whether to go "one way" or just bypass the code in the if structure.

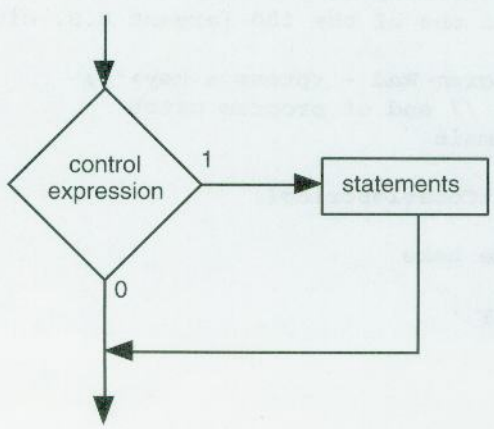


FIGURE 7-6
The if structure is sometimes called a one-way selection structure.

PITFALLS

You have become accustomed to using semicolons to end statements. However, using semicolons to end each line in if structures can cause problems, as shown below.

```
if (i == 3);      // don't do this!!
  { System.out.println ("The value of i is 3"); }
```

The statement in braces will execute in every case because the compiler interprets the semicolon as the end of the if structure.

Analyze the program in Figure 7-7. The program declares a String and an integer. The user is asked for the name of her city or town, and for the population of the city or town. The if structure compares the population to a value that would indicate whether the city is among the 100 largest U.S. cities. If the city is one of the 100 largest U.S. cities, the program prints a message saying so.

```
import java.lang.StringBuffer;
import java.io.IOException;

class City
{
  public static void main(String Args[])
  {
  System.out.println("Begin Program");

  String city_name = new String();
  String temp = new String();
  long population;

  System.out.print("Please enter the name of your city or town: ");
  city_name = GetConsoleString();

  System.out.print("Please enter the population of your city or town: ");
  temp = GetConsoleString();
  population = Long.parseLong(temp);
  if(population >= 171439)
    {
    System.out.println("According to the 1990 census, " + city_name);
    System.out.println("is one of the 100 largest U.S. cities.");
    }
  System.out.println("Program End - <press a key>");
  GetConsoleString();      // end of program catch
  }      // end of method main

  public static String GetConsoleString()
  {
  // GetConsoleString code here
  }
}      // end of class City
```

FIGURE 7-7
This program uses a one-way selection structure.

EXERCISE 7-4 USING if

1. Open **City.java** from the student data files. The program from Figure 7-7 appears without the if structure.
2. Add the if structure shown in Figure 7-7 to the program. Enter the code carefully. Save the source file to your course folder or disk as **City.java**.
3. Compile, link, and run the program. Enter your city or town to test the program.
4. If your city or town is not one of the 100 largest cities, enter Newport News, a city in Virginia with a population of 171,439.
5. Leave the source code file open for the next exercise.

USING if/else

The if/else structure is sometimes called a ***two-way selection structure.*** Using if/else, one block of code is executed if the control expression is true and another block is executed if the control expression is false. Consider the code fragment below.

```
if (i < 0)
  {
  System.out.println ("The number is negative.");
  }
else
  {
  System.out.println ("The number is zero or positive.");
  }
```

The *else* portion of the structure is executed if the control expression is false. Figure 7-8 shows a flowchart for a two-way selection structure.

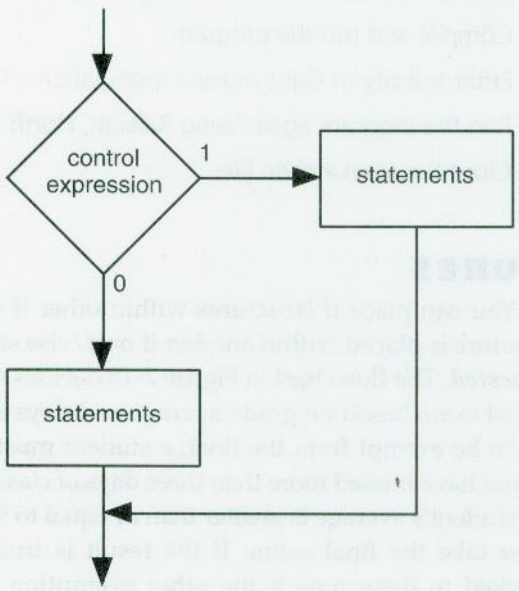


FIGURE 7-8
The if/else structure is a two-way selection structure.

The code shown in Figure 7-9 adds an else clause to the if structure in the program in Exercise 7-4. Output is improved by providing information on whether the city's population qualifies it as one of the 100 largest U.S. cities. If the population is 171,439 or more, the first output statement is executed; otherwise, the second output statement is executed. In every case, either one or the other output statement is executed.

```
if(population > 171439)
   {
   System.out.println("According to the 1990 census, " + city_name);
   System.out.println("is one of the\n100 largest U.S. cities.");
   }
else
   {
   System.out.println("According to the 1990 census, " + city_name);
   System.out.println("is not one of the\n100 largest U.S. cities.");
   }
```

FIGURE 7-9
With an if/else structure, one of the two blocks of code will always be executed.

PITFALLS

Many programmers make the mistake of using > or < when they really need >= or <=. In the code segment in Figure 7-9, using > rather than >= would cause Newport News, the 100th largest city, to be excluded because its population is 171,439, not greater than 171,439.

EXERCISE 7-5 USING if/else

1. Add the else clause shown in Figure 7-9 to the if structure in the program on your screen. Change the class name to CityElse and save the new program to your course disk or folder as **CityElse.java**.
2. Compile and run the program.
3. Enter the city of Gary, Indiana (population 116,646).
4. Run the program again using Raleigh, North Carolina (population 212,050).
5. Close the source code file.

NESTED if STRUCTURES

You can place if structures within other if structures. When an if or if/else structure is placed within another if or if/else structure, the structures are said to be *nested*. The flowchart in Figure 7-10 decides whether a student is exempt from a final exam based on grade average and days absent.

To be exempt from the final, a student must have a 90 average or better and cannot have missed more than three days of class. The algorithm first determines if the student's average is greater than or equal to 90. If the result is false, the student must take the final exam. If the result is true, the number of days absent is checked to determine if the other exemption requirement is met. Figure 7-11 shows the algorithm as a Java code segment.

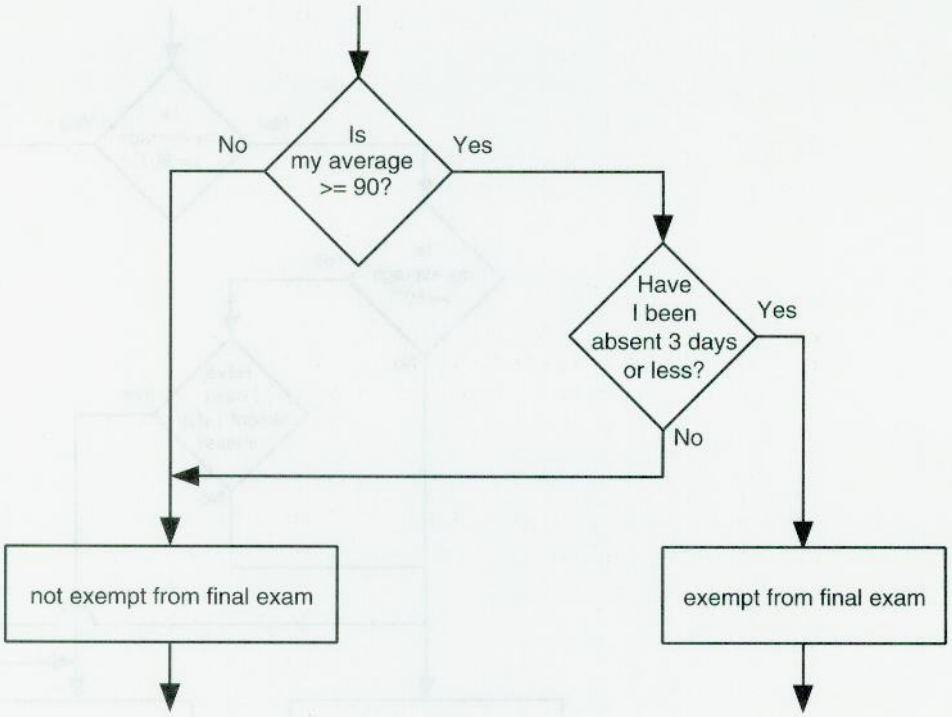


FIGURE 7-10
This flowchart can be programmed using nested if structures.

```
exemptFromFinal = false;
if(myAverage >= 90)
  {
  if(myDaysAbsent <= 3)
    {
    exemptFromFinal = true;
    }
  }
```

FIGURE 7-11
Nested if structures can be used to check two requirements before making a final decision.

Note also the use of both indenting and braces in this code segment. It is clear which responses are a result of which decision-making processes. The indenting (horizontally) helps other programmers see what you meant to happen if the condition were true. In addition, the braces provide vertical separation so the code is easy to read. This cannot always be done in textbook examples because of space restrictions, but you should always implement coding formats that support clear understanding of your intentions.

Algorithms involving nested if structures can get more complicated than the one in Figure 7-10. Figure 7-12 shows the flow chart from Figure 7-10 expanded to include another way to be exempted from the final exam. In this expanded algorithm, students can also be exempted if they have an 80 or higher average, as long as they have been present every day or missed only once.

Note

In the code segment in Figure 7-11, the variable that tells whether the person is exempt from the final is set to false. The program assumes that the student fails to meet the exemption qualification and tests to determine otherwise.

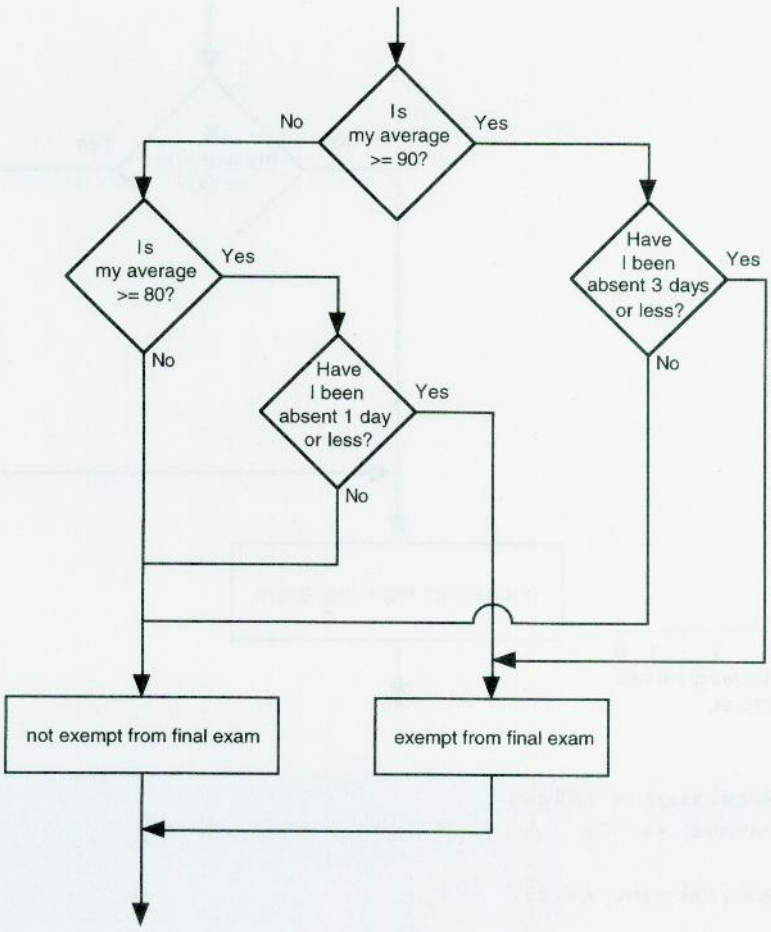


FIGURE 7-12
This algorithm provides two paths to exemption from the final.

As you can probably imagine, programming the algorithm in Figure 7-12 will require careful construction and nesting of if and if/else structures. Figure 7-13 shows you how it is done.

```
if(myAverage >= 90)       // if your average is 90 or better
  {    // and you have missed three days
  if(myDaysAbsent <= 3)  // or less, you are exempt
    {
    exemptFromFinal = true;
    }  // End of if (myDays... clause
  }   // End of if(myAverage... clause
else
  {
  if(myAverage >= 80)     // if your average is 80 or
    {  // better and you have missed
    if(myDaysAbsent <= 1)       // one day or less,
      {       // you are exempt
      exemptFromFinal = true;
      }       // End of if (myDays... clause
    }   // End of if(myAverage... clause
  }
```

FIGURE 7-13
Nested if structures require careful construction.

Note

Earlier you learned that it is a good idea to always use braces with if structures. Figure 7-12 illustrates another reason why you should do so. Without the braces, the compiler may assume that the else clause goes with the nested if structure rather than the first if.

EXERCISE 7-6 NESTED if STRUCTURES

1. Open the source for **Final.java** from the student data files and save it to your student disk as **Final.java**.
2. Compile and run the program. Run the program several times, testing different combinations of values.
3. When you have verified that the output is correct, close the source code file.

Extra for Experts

The nested if structure in Figure 7-12 is used to set the **exempt_from_final** *variable to true or false. Any type of statements could appear in the nested if statement. However, because the code in this case simply sets the value of the* **exempt_from_final** *variable, the entire nested if structure can be replaced with the statement below.*

```
exempt_from_final = (((my_average >= 90) && (my_days_absent <= 3)) ||
        ((my_average >= 80) && (my_days_absent <= 1)));
```

Using logical operators, the statement above combines the expressions used in the nested if structure. The result is the value desired for **exempt_from_final**. *To prove it, replace the nested if structure in* ***Final.java*** *with the statement above and run the program again.*

Do not be fooled, however, into thinking that statements like the one above make if structures unnecessary. Ordinarily, a selection structure cannot be replaced with a sequence structure and achieve the same result.

Figure 7-14 shows a simple program that includes a nested if structure. The program asks the user to input the amount of money he or she wishes to deposit in order to open a new checking account. Based on the value provided by the user, the program recommends a type of account.

EXERCISE 7-7 MORE NESTED if

1. Open **Deposit.java** from the student data files. The program shown in Figure 7-14 appears. Save the program to your course folder or disk as **Deposit.java**.
2. Compile and run the program. Run the program several times using values that are less than $100, between $100 and $1000, and greater than $1000.
3. Leave the source code open for the next exercise.

```
class Deposit
{
  public static void main(String args[])
  {
  String inString = new String();
  float depositAmount;

  System.out.println("Begin Program");
  System.out.print("Enter deposit amount to open account:");
  inString = GetConsoleString();
  depositAmount = toFloat(inString);
  if(depositAmount < 1000)
    {
    if(depositAmount < 100)
      {
      System.out.println("Please consider our EconoCheck account");
      }
    else
      {
      System.out.println("Please consider our FreeCheck account");
      }
    }
   else
    {
    System.out.println("Please consider our InterestMaker account");
    }

  System.out.println("Program End - <press Enter key>");
  GetConsoleString();
  }     // end of main method

  public static String GetConsoleString()
  {
  // CODE FOR GETCONSOLESTRING NOT SHOWN HERE DUE TO SPACE
  }     // end of method GetConsoleString

  public static float toFloat(String inString)
  {
  Float tempFloat = new Float(inString);
  return tempFloat.floatValue();
  }     // end of toFloat method

}     // end of class Deposit
```

FIGURE 7-14
The nested if structure can be used to make recommendations.

Note the addition of another class member method **toFloat**. While this is a fairly simple method, it is an example of modularizing your code so that it is more readable. Remember that virtually all input and output for Java is in the form of a string.

Your program captures the string and then converts it into the float for decision-making use. It does this by passing **inString** to the method, and then returning a float from the method. If you remember from previous chapter discussions, a member method is like a courier that does little chores for you. This courier picks up **inString** and then returns with the value to be placed in the variable **depositAmount**.

There is a way to tighten up your code even more. You can remove the following two lines of code:

```
String inString = new String();
inString = GetConsoleString();
```

and replace the line containing:

```
depositAmount = toFloat(inString);
```

with

```
depositAmount = toFloat(GetConsoleString());
```

Because **GetConsoleString** returns a String, there is no need to temporarily store the String in the object inString. The call to **GetConsoleString** may be placed directly into the line that calls the **toFloat** method.

EXERCISE 7-8 REFINING THE CODE

1. Remove the two lines shown below.

```
String inString = new String();
inString = GetConsoleString();
```

2. Replace the line

```
depositAmount = toFloat(inString);
```

with

```
depositAmount = toFloat(GetConsoleString());
```

3. Run this program again, testing it with different numbers. You will see that it works just the same as it did.
4. Save and close the source code file.

There are advantages and disadvantages to making methods into parameters for other methods. While doing so does reduce the amount of code, there are times when it makes the code harder to read or more confusing. As you pack more action into a line of code, it is sometimes difficult to grasp all that is going on in the code.

Programmers are always faced with trade-offs. This is an example of a situation where you may give up some readability for more efficient code. Sometimes the more efficient code is necessary. Often, however, the more readable code will reduce development and maintenance costs, which may be more important than a slight increase in efficiency. In most cases, a clearly written program is preferable.

THE switch STRUCTURE

You have studied one-way (if) and two-way (if/else) selection structures. Java has another method of handling multiple options known as the switch structure. The switch structure has many uses but may be most often used when working with menus. Figure 7-15 is a code segment that displays a menu of choices and asks the user to enter a number that corresponds to one of the choices. Then, a case statement is used to handle each of the options.

```
import java.lang.StringBuffer;
import java.io.IOException;

class Shipping
{
  public static void main(String Args[])
  {
  int shippingMethod;
  double shippingCost;

  System.out.println("SHIPPING PRICE MENU");
  System.out.println("How do you want your order shipped?");
  System.out.println("1 - Ground");
  System.out.println("2 - Two Day Air");
  System.out.println("3 - Over Night Air");
  System.out.print("Please enter desired shipping method: ");

  shippingMethod = Integer.parseInt(GetConsoleString());
  switch(shippingMethod)
    {
    case 1:
      shippingCost = 5.25;
      break;
    case 2:
      shippingCost = 7.75;
      break;
    case 3:
      shippingCost = 10.25;
      break;
    default:
      shippingCost = 0.0;
      break;
    }

  System.out.println("The price for delivery is: $" + shippingCost);
  System.out.println("Press Enter to end program");
  GetConsoleString();
  }     // end of main method

  public static String GetConsoleString()
  {
  // code not shown due to space limitations
  }     // end of method GetConsoleString

}     // end of class Shipping
```

FIGURE 7-15
The switch structure takes action based on the user's input.

Note

*A **menu** is a set of options presented to the user of a program.*

Let's analyze the switch structure in Figure 7-15. It begins with the keyword *switch*, followed by the control expression (the variable **shipping_method**) to be compared in the structure. Within the braces of the structure are a series of ***case*** keywords. Each one provides the code that is to be executed in the event that **shipping_method** matches the value that follows case. The *default* keyword tells the compiler that if nothing else matches, execute the statements that follow.

The *break* keyword, which appears at the end of each case segment, causes the flow of logic to jump to the first executable statement after the switch structure.

Note

In Java, only integer or character types may be used as control expressions in switch statements.

EXERCISE 7-9 USING switch

1. Open **Shipping.java** from the student data files. The program includes the segment from Figure 7-15. Save the program to your course folder or disk as **Shipping.java**.
2. Compile and run the program. Choose shipping method 2.
3. Add a fourth shipping option called Carrier Pigeon to the menu and add the necessary code to the switch structure. You decide how much it should cost to ship by carrier pigeon.
4. Compile and run to test your addition to the options.

Nested if/else structures could be used in the place of the switch structure. But the switch structure is easier to use and a programmer is less prone to making errors that are related to braces and indentions. Remember, however, that an integer, character, byte, or short data type is required in the control expression of a switch structure. Nested ifs must be used if you are comparing floats and most other objects.

When using character types in a switch structure, enclose the characters in single quotes like any other character literal. The following code segment is an example of using character literals in a switch structure.

```
switch(character_entered)
  {
  case 'A':
    system.out.println ("The character entered was A, as in albatross.");
    break;
  case 'B':
    system.out.println ("The character entered was B, as in butterfly.");
    break;
  default:
    system.out.println ("Illegal entry");
    break;
  }
```

Extra for Experts

Java allows you to place your case statements in any order. You can, however, increase the speed of your program by placing the more common choices at the top of the switch structure and less common ones toward the bottom. The reason is that the computer makes the comparisons in the order they appear in the switch structure. The sooner a match is found, the sooner the computer can move on to other processing.

SECTION 7.2 QUESTIONS

SHORT ANSWER

1. What is the purpose of the *break* keyword?
2. Write an if structure that prints the word *help* to the screen if the variable **need_help** is equal to 1.
3. Write an if/else structure that prints the word *Full* to the screen if the float variable **fuel_level** is equal to 1, and prints the value of **fuel_level** if it is not equal to 1.
4. What is wrong with the if structure below?

```
if (x > y);
        { System.out.println ("x is greater than y"); }
```

5. What is wrong with the if structure below?

```
if (x = y)
        { System.out.println ("x is equal to y"); }
```

PROBLEM 7.2.1

Write a program that asks for an integer and displays for the user whether the number is even or odd. **Hint:** Use if/else and the modulus operator. Save the source code file to your student disk or folder as **evenodd.java**.

PROBLEM 7.2.2

Rewrite the code in the **Final.java** file that you saved earlier in this chapter so that it begins with the assumption that the student is exempt and makes comparisons to see if the student must take the test. Save the revised source code to your student disk or folder as **Final2.java**.

KEY TERMS

branching
case
comparison operators
control expression
fuzzy logic
if structure
if/else structure
logical operators
menu
nested
one-way selection structure
selection structures
sequence structures
short-circuit evaluation
switch structure
truth tables
two-way selection structure

SUMMARY

- Computers make decisions by comparing data.
- Comparison operators are used to create expressions that result in a value of true or false.
- Logical operators can combine comparison expressions.
- Selection structures are how Java programs make decisions.
- The if structure is a one-way selection structure. When a control expression in an if statement is evaluated to be true, the statements associated with the structure are executed.
- The if/else structure is a two-way selection structure. If the control expression in the if statement evaluates to true, one block of statements is executed; otherwise another block is executed.
- The switch structure is a multi-way selection structure that executes one of many sets of statements depending on the value of the control expression. The control expression must evaluate to an integer, character, byte, or short value.

PROJECTS

PROJECT 7-1 • LENGTH CONVERSION

1. Open **lengths.java** from the student data files, save it to your student disk using the same name, and analyze the source code.
2. Compile and run it several times and try different conversions and values.
3. Add a conversion for miles to the program. Use 0.00018939 for the conversion factor.
4. Test the program.

PROJECT 7-2 • FINANCE

Obtain the exchange rates for at least three foreign currencies. Write a program similar in form to **lengths.java** that asks for an amount of money in American dollars and then prompts the user to select the currency into which the dollars are to be converted. Save the program to your student disk as **Project 7_2.java.** Compile and run the program.

PROJECT 7-3 • COMPUTERIZED TESTING

Write a program that asks the user a multiple-choice question on a topic of your choice. Test the user's answer to see if the correct response was entered. When the program works for one question, add two or three more. Save the program to your student disk as **Project7_3.java.** Compile and run the program.

8 Loops

OBJECTIVES

- *Explain the importance of loops in programs.*
- *Write a program that uses for loops, while loops, do while loops, and nested loops.*
- *Demonstrate the use of the* break *and* continue *keywords.*

Overview

You have probably noticed that much of the work a computer does is actually repeated over and over. For example, a computer can print a personalized letter to each person in a database. What happens, of course, is the basic operation of printing the letter repeats for each person in the database. When a program repeats a group of statements a given number of times, the repetition is accomplished using a ***loop.***

In Chapter 7 you learned about sequence structures and selection structures. The final category of structures is ***iteration structures.*** Loops are iteration structures, for each "loop" or pass through a group of statements is called an ***iteration.*** A condition specified in the program controls the number of iterations performed. For example, a loop may iterate a set number of times or until a specific condition occurs, such as a variable reaching the value 100.

In this chapter you will learn about the three iteration structures available in Java: the for loop, the while loop, and the do while loop.

CHAPTER 8, SECTION 1

The for Loop

The *for loop* repeats one or more statements a specified number of times. A for loop is difficult to read the first time you see one. Like an if statement, the for loop uses parentheses. In the parentheses are three items called ***parameters*** that are needed to make a for loop work. Each parameter in a for loop is an expression. Figure 8-1 shows the format of a for loop.

```
for(initializing expression; control expression; step expression)
  {
  statement or statement block
  }
```

FIGURE 8-1
A for loop repeats one or more statements a specified number of times.

Look at the main method in Figure 8-2. The variable `i` is used as a counter variable. Notice that the counter variable is used in all three of the for loop's expressions. The first parameter, called the ***initializing expression,*** gives the counter variable a starting value. The second parameter is the expression that will end the loop, called the ***control expression.*** As long as the control expression is true, the loop continues to iterate. The third parameter is the ***step expression.*** It changes the counter variable, usually by adding to it.

```
class ForLoop
{
  public static void main(String Args[])
  {
  int i;
  for(i = 1; i <= 3; i++)
    System.out.println("i = " + i);
  }     // end of method main
}     // end of class ForLoop
```

FIGURE 8-2
A for loop uses a counter variable to test the control expression. Here the statements in the for loop will repeat three times.

Let's look at each part of the loop in Figure 8-2.

- **initializing expression: `i=1`** The variable `i`, which was declared as an integer, initialized to 1.
- **control expression: `i<= 3`** The control expression tests to see if the value of `i` is still less than or equal to 3. When `i` exceeds 3, the loop will end.
- **step expression: `i++`** The step expression increments `i` by one each time the loop iterates.

PITFALLS

Placing a semicolon after the closing parenthesis of a for loop will prevent any lines from being iterated.

EXERCISE 8-1 USING A for LOOP

1. Start your text editor and key the program from Figure 8-2.
2. Save the source code file to your student disk or folder as **ForLoop.java**.
3. Compile and run the program.
4. Close the source file.

COUNTING BACKWARD AND OTHER TRICKS

A counter variable can also count backward by having the step expression decrement the value rather than increment it.

EXERCISE 8-2 USING A DECREMENTING COUNTER VARIABLE

1. Key the following program into a blank editor screen:

```
class BackWard
{
  public static void main(String Args[])
  {
```

```
    int i;
    for(i = 5; i >= 0; i--)
      System.out.println(i);
    System.out.println("End of loop.\n");
    }     // end of main method
}     // end of class BackWard
```

2. Save the source file to your student disk or folder as **BackWard.java**.
3. Compile and run the program. Figure 8-3 shows the output you should see.
4. Close the source code file.

```
5
4
3
2
1
0
End of loop.
```

FIGURE 8-3
A for loop can decrement the counter variable.

The output prints numbers from 5 to 0 because `i` is being decremented in the step expression. The phrase "End of loop." is printed only once because the loop ends with the semicolon that follows the first System.out statement. Note that the line to be repeated by the loop is clearly indented.

The counter variable can do more than step by one. In the program in Figure 8-4, the counter variable is doubled each time the loop iterates. In the next exercise, you will see the effect of this for loop.

EXERCISE 8-3 INCREMENTING BY A STEP OTHER THAN ONE

1. Key the program from Figure 8-4 into a blank editor screen.
2. Save the source file to your student disk or folder as **DblStep.java.** Can you predict the program's output?
3. Compile and run the program to see if your prediction was right.
4. Close the source file.

```
class DblStep
{
  public static void main(String Args[])
  {
  int i;      // counter variable
  for(i = 1; i <= 100; i = i + i)
    System.out.println("i is now: " + i);
  }     // end of main method
}     // end of class DblStep
```

FIGURE 8-4
The counter variable in a for loop can be changed by any valid expression.

The for statement gives you a great deal of flexibility. As you have already seen, the step expression can increment, decrement, or count in other ways. Some more examples of for statements are shown in Table 8-1.

TABLE 8-1

FOR STATEMENT	COUNT PROGRESSION
for (i = 2; i <= 10; i = i + 2)	2, 4, 6, 8, 10
for (i = 1; i < 10; i = i + 2)	1, 3, 5, 7, 9
for (i = 10; i <= 50; i = i + 10)	10, 20, 30, 40, 50

USING A STATEMENT BLOCK IN A for LOOP

If you need to include more than one statement in the loop, use braces to make a statement block below the for statement. If the first character following a for statement is an open brace ({), all of the statements between the braces are repeated. The same rule applies as with if structures.

EXERCISE 8-4 USING A STATEMENT BLOCK IN A for LOOP

1. Key in the following source code:

```
class BackWard2
{
  public static void main(String Args[])
  {
  int i;
  for(i = 10; i >= 0; i--)
    {
    System.out.println (i);
    System.out.println ("This is in the loop.\n");
    }
  }     // end of main method
}     // end of class BackWard2
```

2. Compile and run the program to see that the phrase beginning with "This" is in the loop that prints on every line. The second println statement is now part of the loop because it is within the braces.
3. Close the source file without saving changes.

Extra for Experts

Earlier in this chapter, you learned that placing a semicolon at the end of the parentheses in a for statement will cause the loop to do nothing. While it is true that the statements that follow will not be iterated, there are cases where you might actually want to do just that.

Suppose you are given an integer and asked to calculate the largest three-digit number that can be produced by repeatedly doubling the given integer. For example, if the given integer is 12, repeatedly doubling the integer would produce the sequence 12, 24, 48, 96, 192, 384, 768, 1536. Therefore, 768 is the largest three-digit number produced by repeatedly doubling the number 12.

The program segment below uses an empty loop to achieve the result outlined above.

```
public static void main(String args[])
{
long i;
i = Integer.parseInt(GetConsoleString());
for(; i <= 1000; i = i * 2);
  System.out.println (i/2);
}
```

The first parameter of the for statement is left blank because `i` *is initialized by the user in the GetConsoleString statement. Even though the loop is empty, the stepping of the counter variable continues and the value is available after the loop terminates. The value of* `i` *is 1000 or more when the loop ends. The println statement then divides the counter by two to return it to the highest three-digit number.*

SECTION 8.1 QUESTIONS

SHORT ANSWER

1. What category of structures includes loops?
2. What is the name of the for loop parameter that ends the loop?
3. What for loop parameter changes the counter variable?
4. What happens if you key a semicolon after the parentheses of a for statement?
5. How many statements can be included in a loop?
6. Write a for statement that will print the numerals 3, 6, 12, 24.
7. Write a for statement that will print the numerals 24, 12, 6, 3.

PROBLEM 8.1.1

Write a program that uses a for loop to print the odd numbers from 1 to 21. Save the source file to your course disk or folder as **OddLoop.java.**

CHAPTER 8, SECTION 2

while Loops

A while loop is similar to a for loop. Actually, while loops are sometimes easier to use than for loops and are better suited in many situations. With a for loop, the parameters in the parentheses control the number of times the loop iterates; in a while loop, something

inside the loop triggers the loop to stop. While loops are often called "conditional" because they will continue to repeat until a certain condition occurs. For example, a while loop may be written to ask a user to input a series of numbers until the number 0 is entered. The loop would repeat until 0 is entered.

There are two kinds of while loops: the standard while loop and the do while loop. The difference between the two is where the control expression is tested. Let's begin with the standard while loop.

THE STANDARD while LOOP

The *while loop* repeats a statement or group of statements as long as a control expression is true. Unlike a for loop, a while loop does not use a counter variable. The control expression in a while loop can be any valid expression. The program in Figure 8-5 uses a while loop to divide a number by 2 repeatedly until the number is less than or equal to 1.

```
public static void main(String args[])
{
float num;
System.out.println("Please enter the number to divide:");
num = toFloat(GetConsoleString());
while(num > 1)
  {
  System.out.println("num is: " + num);
  num = num / 2;
  }
}     // end of method main
```

FIGURE 8-5
A while loop may not use a counter variable.

In a while loop, the control expression is tested before the statements in the loop begin. Figure 8-6 shows a flowchart of the program segment in Figure 8-5. If the number provided by the user is less than or equal to 1, the statements in the loop are never executed.

Something else to know about while loops is that like the if statements, while loops can work with objects as well as the basic data types. In other words, the following code segment is possible (but not commonly used):

```
while(! FloatObject.equals((Object) floatVariable))
  {
  // do some action
  }
```

The value that might be gained by using object-oriented programming here is lost in having to write additional code to manage the Float object, so this operation will not be used often. However, you will see some conditions that work in this manner.

PITFALLS

As with the for loop, placing a semicolon after the closing parenthesis of a while loop will prevent any lines from being iterated.

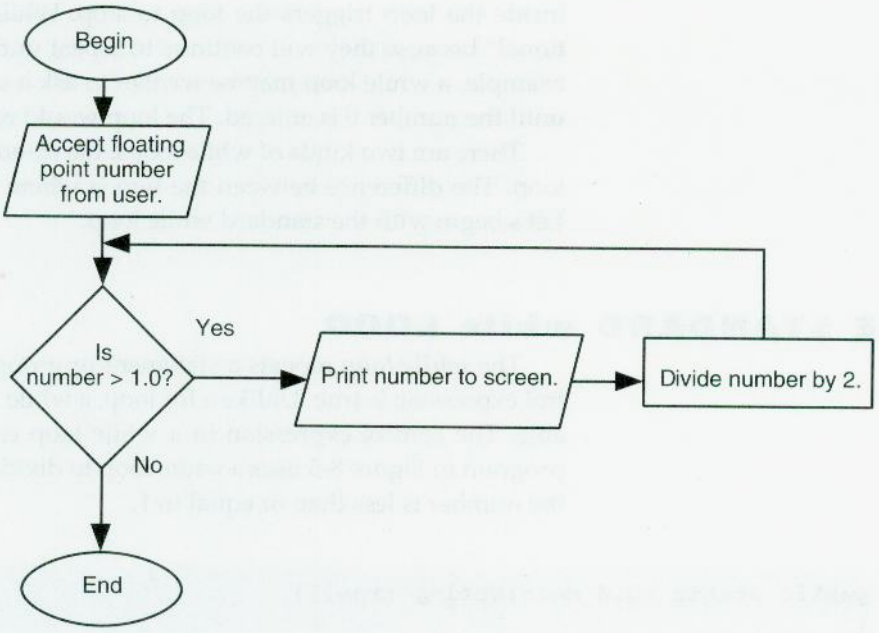


FIGURE 8-6
A while loop tests the control expression before the loop begins.

EXERCISE 8-5 USING A while LOOP

1. Enter the program segment shown in Figure 8-5 into a blank editor screen. Copy in the `toFloat` and `GetConsoleString` methods from a previous file (don't forget the import statements) and make your own While1 class.
2. Save the source file to your student disk or folder as **While1.java.**
3. Compile and run the program. Run the program several times. Try the following numbers as input: 8, 21, 8650, 1, 2.1, 0.5.
4. Close the source file.

Note

At this point in the book, you should be able to develop a program on your own. Many of the code segments given from here forward will be just that—segments that most commonly will show the main method of a program. Using the given main method, and the supporting methods `toFloat` *and* `GetConsoleString`*, you can develop your own programs. If you need a reminder, refer back to your previous programming examples. The method* `toFloat` *is introduced in Figure 7-14, Chapter 7; and the method* `GetConsoleString` *is introduced in Chapter 6, using the file* ***Newmethod.java.*** *Having it in a file like this means that you can copy the method code right from the file and place it in your own program.*

In order for a while loop to come to an end, the statements in the loop must change a variable used in the control expression. The result of the control expression must be false for a loop to stop. Otherwise, iterations continue indefinitely in what is called an *infinite loop*. In the program you compiled in Exercise 8-5, the statement `num = num / 2;` divides the number by two each time the loop repeats. Even if the user enters a large value, the loop will eventually end when the number becomes less than 1.

A while loop can be used to replace any for loop. So why have a for loop in the language? Because sometimes a for loop offers a better solution. Figure 8-7 shows two program segments that produce the same output. The program using the for loop is better in this case because the counter variable is initialized, tested, and incremented in the same statement. In a while loop, a counter variable must be initialized and incremented in separate statements.

However, the opposite is not always true; that is, a for loop cannot always replace a while loop. That's because many while loops iterate until a particular condition exists. Often you will not know how many times the loop will repeat until that condition occurs (as when asking a user to enter 0 when he or she has no more data to enter). In those instances, a while loop must be used.

```
public static void main(String args[])
{
int i;
for(i = 1; i <= 100; i++)
  System.out.println("i = " + i);
}     // end of method main
```

```
public static void main(String args[])
{
int i = 1;
while(i <= 100)
  {
  System.out.println("i = " + i)
    i++;
  }
}     // end of method main
```

FIGURE 8-7
Although both of these programs produce the same output, the for loop gives a more efficient solution.

THE do while LOOP

The last iteration structure in Java is the *do while* loop. A do while loop repeats a statement or group of statements as long as a control expression is true at the end of the loop. Because the control expression is tested at the end of the loop, a do while loop is executed at least one time. Figure 8-8 shows an example of a do while loop.

```
class DoWhile
{
  public static void main(String args[])
  {
  float num, squared;

  do
    {
    System.out.println("Enter a number (Enter 0 to quit):");
    num = toFloat(GetConsoleString());
    squared = num * num;
    System.out.println("The number " + num + " squared is: " + squared);
    }while(num != 0);
  }     // end of method main
```

FIGURE 8-8
In a do while loop, the control expression is tested at the end of the loop.

To help illustrate the difference between a while and a do while loop, compare the two flowcharts in Figure 8-9. Use a while loop when you need to test the control expression before the loop is executed the first time. Use a do while loop when the statements in the loop need to be executed at least once.

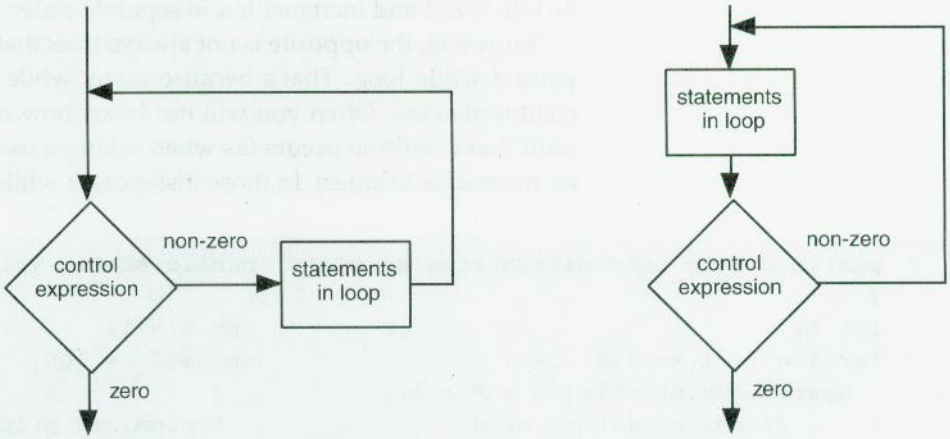


FIGURE 8-9 The difference between a while loop and a do while loop is where the control expression is tested.

EXERCISE 8-6 USING A do while LOOP

1. Enter the main method from Figure 8-8 into a blank editor screen, and add the `GetConsoleString` and `toFloat` methods to your DoWhile class.
2. Save the source file to your student disk or folder as **DoWhile.java**.
3. Compile and run the program. Enter several numbers greater than 0 to cause the loop to repeat. Enter 0 to end the program.
4. Close the source file.

STOPPING IN THE MIDDLE OF A LOOP

The keyword *break*, also utilized with switch statements, can be used to end a loop before the conditions of the control expression are met. Once a break terminates a loop, the execution begins with the first statement following the loop. In the program you ran in Exercise 8-6, entering zero caused the program to end. But the program squares zero before it ends, even though the step is unnecessary. The program segment in Figure 8-10 uses a break statement to correct the problem.

Note

You should allow the control expression to end an iteration structure whenever practical. When you are tempted to use a break statement to exit a loop, make sure that using the break statement is the best way to end the loop.

In the program in Figure 8-10, the value entered by the user is tested with an if statement as soon as it is input. If the value is zero, the break statement is executed to end the loop. If the value is any number other than zero, the loop continues. The control expression can remain `num == 0` without affecting the function of the program. In this case, however, the break statement will stop the loop before the control expression is reached. Therefore, the control expression can be changed to true to create an infinite loop. The true creates an infinite loop because the loop continues to iterate as long as the control expression is true (which is represented by the value true). The loop will repeat until the break statement is executed.

```
public static void main(String args[])
{
float num, squared;

do
  {
  System.out.println("Enter a number (Enter 0 to quit):");
  num = toFloat(GetConsoleString());
  if(num == 0)
    {
    break;
    }
  squared = num * num;
 System.out.println("The number " + num + " squared is: " + squared);
  }while(num != 0);
}     // end of method main
```

FIGURE 8-10
The break statement ends the loop as soon as the value of zero is input.

The *continue statement* is another way to stop a loop from completing each statement. But instead of continuing with the first statement after the loop, the continue statement skips the lines below it in the loop and starts over with the next iteration of the loop. Figure 8-11 shows an example of how the continue statement can be used to cause a for loop to skip an iteration.

```
public static void main(String args[])
{
int i;
for(i = 1; i <= 10; i ++)
  {
  if(i == 5)
    continue;
  System.out.println("i = " + i);
  }
}     // end of method main
```

FIGURE 8-11
The continue keyword eliminates the number 5 from the outputted list of numbers.

The continue statement in Figure 8-11 causes the statements in the for loop to be skipped when the counter variable is 5. The continue statement also can be used in while and do while statements.

EXERCISE 8-7

USING THE CONTINUE STATEMENT

1. Open the source code file **Continue.java** from the student data files and save it to your student disk or folder using the same name.
2. Compile and run the program. Notice that the number 5 does not appear in the output because of the continue statement.
3. Close the source file.

NESTING LOOPS

In Chapter 7 you learned how to nest if structures. Loops can also be nested. In fact, loops within loops are very common. You must trace the steps of the program carefully to understand how ***nested loops*** behave. The program in Figure 8-12 provides output that will give you insight into the behavior of nested loops.

```
public static void main(String args[])
{
int i, j;
System.out.println("Program Begin");
for(i = 1; i <=3; i++)
  {
  System.out.println(" Outer loop:  i = " + i);
  for(j = 1; j <= 3; j++)
    {
    System.out.println("      Inner loop:  j = " + j);
    }
  }
System.out.println("Program End");
}
```

FIGURE 8-12
Even though this program has little practical use, it illustrates what happens when loops are nested.

The important thing to realize is that the inner for loop (the one that uses j) will complete its count from 1 to 4 every time the outer for loop (the one that uses i) iterates. That is why in the output, for every loop the outer loop makes, the inner loop starts over (see Figure 8-13).

```
Program Begin
Outer loop:  i = 1
  Inner loop:  j = 1
  Inner loop:  j = 2
  Inner loop:  j = 3
Outer loop:  i = 2
  Inner loop:  j = 1
  Inner loop:  j = 2
  Inner loop:  j = 3
Outer loop:  i = 3
  Inner loop:  j = 1
  Inner loop:  j = 2
  Inner loop:  j = 3
Program End
```

FIGURE 8-13
The output of the program in Figure 8-12 illustrates the effect of the nested loops.

EXERCISE 8-8 NESTED LOOPS

1. Open **NestLoop.java** from the student data files.
2. Compile and run the program.

Note

If you know how to use your compiler's debugger, step through the program to trace the flow of logic.

3. Close the source file without saving it.

Nesting may also be used with while loops and do while loops, or in combinations of loops. The program in Figure 8-14 nests a do while loop in a for loop.

```
public static void main(String Args[])
{
CHAR PARTY;
char ERROR = '\0';
int i, numReps;
int democrats = 0, republicans = 0, independents = 0;
String tempVal = new String();

System.out.println("\nHow many U.S. representatives does your state have?");
numReps = Integer.parseInt(GetConsoleString());

System.out.println("Enter the party affiliation for each representative");
System.out.println("Enter D for Democrat, R for Republican,");
System.out.println("and I for Independents or other parties");

for (i = 1; i <= numReps; i++)
  {
  do
    {
    System.out.println("Party of representative #" + i);
    tempVal = GetConsoleString();
    party = tempVal.charAt(0);

    switch(party)
      {
      case 'D':
      case 'd':
        democrats++;
        break;
      case 'R':
      case 'r':
        republicans++;
        break;
      case 'I':
      case 'i':
        independents++;
        break;
      default:
        System.out.println("Invalid entry.  Enter D, R, or I.");
        party = ERROR;
        break;
      }     // end of switch statement
    }while(party == ERROR);  //  end of do . . while loop
  }     // end of for loop
System.out.println("Your state is represented by:");
System.out.println("      " + democrats + " Democrats");
System.out.println("      " + republicans + " Republicans");
System.out.println(" and " + independents + " Independents");
}     // END OF MAIN METHOD
```

FIGURE 8-14
This program has a do while loop nested within a for loop.

The program segment in Figure 8-14 asks the user for the number of U.S. representatives in his or her state. A for loop is used to ask the user to identify the party of each representative. The do while loop is used to repeat the prompt if the user enters an invalid party choice.

EXERCISE 8-9 MORE NESTED LOOPS

1. Open **Reps.java** from the student data files and save it to your student disk using the same name.
2. Study the program carefully before you run it.
3. Compile and run the program. Enter some invalid data to cause the nested loop to iterate. If you have trouble understanding the program, study the source code and run it again.
4. Close the source code file without saving.

SECTION 8.2 QUESTIONS

SHORT ANSWER

1. Where does a do while loop test the control expression?
2. What is the term for a loop without a way to end?
3. What is the loop control expression in the code segment below?

```
while (!done)
{
  if(i < 1)
    {
    done = true;
    }
  i--;
}
```

4. What is the error in the code segment below?

```
do;
  {
  if(i < 1)
    {
    done = true;
    }
  i--;
  }
while(!done);
```

5. Write a loop that prints your name to the screen once and then asks you to enter 0 (zero) to stop the program or any other number to print the name again.
6. Write a for loop to print the odd numbers from 1 to 999.

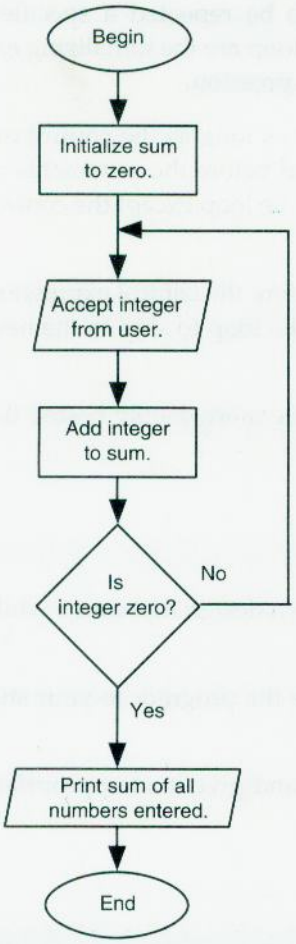


FIGURE 8-15

PROBLEM 8.2.1

Write a program that implements the flowchart in Figure 8-15. Save the source file to your student disk or folder as **Sumitup.java.**

PROBLEM 8.2.2

Write a program that prints the numbers 1 to 20, but skips the numbers 15, 16, and 17. Save the source code file to your student disk or folder as **Skipthem.java**.

PROBLEM 8.2.3

Modify the program from Exercise 8-9 (**Reps.java**) so that it calculates the percentage of your state's representatives that belong to each party. Save the modified source code to your student disk or folder as **Reps2.java**.

PROBLEM 8.2.4

1. Write a program that asks the user for a series of integers one at a time. When the user enters the integer 0, the program displays the following information:
   - the number of integers in the series (not including zero)
   - the average of the integers
   - the largest integer in the series
   - the smallest integer in the series
   - the difference between the largest and smallest integer in the series
2. Save the source file to your course disk or folder as **Ints.java**.

KEY TERMS

continue statement
control expression
do while loop
for loop
infinite loop
initializing expression
iteration
iteration structures
loop
nested loop
parameter
step expression
while loop

SUMMARY

- A loop is a group of statements that is repeated a number of times. A loop is an iteration structure.

- A for loop causes one or more statements to be repeated a specified number of times. The three parameters of a for loop are the initializing expression, the control expression, and the step expression.
- A while loop executes one or more statements as long as the control expression is true. The control expression is tested before the statements in the loop begin. A do while loop works like a while loop except the control expression is tested at the end of the loop.
- A break statement can be used to exit a loop before the control expression ends the loop. The continue statement causes the loop to skip to the next iteration of the loop.
- A loop within a loop is called a nested loop. The more deeply nested the loop, the more times the loop will be executed.

PROJECTS

PROJECT 8-1

1. Draw a flowchart for a simple program of your own design that uses a while loop.
2. Write the Java source code for the program. Save the program to your student disk as **project8_1.java.**
3. Enter the source code into a blank editor screen and give it an appropriate file name.
4. Compile and run the program.

PROJECT 8-2 • GAME PROGRAMMING

Write a program that asks the user to think of a number between 1 and 100, then attempts to guess the number. The program should make an initial guess of 50. The program should then ask the user if 50 is the number the user has in mind, or if 50 is too high or too low. Based on the response given by the user, the program should make another guess. Your program must continue to guess until the correct number is reached. Save the source file to your student disk as **Hi_Lo.java**. Compile and run the program, and close the source file.

PROJECT 8-3 • NUMBER SYSTEMS

1. Open **Binary.java** from the student data files. The program uses four nested loops to print the binary equivalent of 0 to 15 to the screen. Study the source code, save it to your student disk, compile, and run the program to see its output.
2. Modify the program to generate an additional column of digits. The resulting output should be the binary equivalent of 0-31. Save the modified source file to your student disk as **Binary31.java**.
3. Close the source code file.

APPENDIX A ASCII Table

ASCII CHARACTER	DECIMAL	HEXADECIMAL	BINARY
NUL	0	00	000 0000
SOH	1	01	000 0001
STX	2	02	000 0010
ETX	3	03	000 0011
EOT	4	04	000 0100
ENQ	5	05	000 0101
ACK	6	06	000 0110
BEL	7	07	000 0111
BS	8	08	000 1000
HT	9	09	000 1001
LF	10	0A	000 1010
VT	11	0B	000 1011
FF	12	0C	000 1100
CR	13	0D	000 1101
SO	14	0E	000 1110
SI	15	0F	000 1111
DLE	16	10	001 0000
DC1	17	11	001 0001
DC2	18	12	001 0010
DC3	19	13	001 0011
DC4	20	14	001 0100
NAK	21	15	001 0101
SYN	22	16	001 0110
ETB	23	17	001 0111
CAN	24	18	001 1000
EM	25	19	001 1001
SUB	26	1A	001 1010
ESC	27	1B	001 1011
FS	28	1C	001 1100
GS	29	1D	001 1101
RS	30	1E	001 1110
US	31	1F	001 1111
space	32	20	010 0000
!	33	21	010 0001
"	34	22	010 0010
#	35	23	010 0011
$	36	24	010 0100
%	37	25	010 0101
&	38	26	010 0110
'	39	27	010 0111
(	40	28	010 1000
)	41	29	010 1001
*	42	2A	010 1010
+	43	2B	010 1011
,	44	2C	010 1100

ASCII CHARACTER	DECIMAL	HEXADECIMAL	BINARY
-	45	2D	010 1101
.	46	2E	010 1110
/	47	2F	010 1111
0	48	30	011 0000
1	49	31	011 0001
2	50	32	011 0010
3	51	33	011 0011
4	52	34	011 0100
5	53	35	011 0101
6	54	36	011 0110
7	55	37	011 0111
8	56	38	011 1000
9	57	39	011 1001
:	58	3A	011 1010
;	59	3B	011 1011
<	60	3C	011 1100
=	61	3D	011 1101
>	62	3E	011 1110
?	63	3F	011 1111
@	64	40	100 0000
A	65	41	100 0001
B	66	42	100 0010
C	67	43	100 0011
D	68	44	100 0100
E	69	45	100 0101
F	70	46	100 0110
G	71	47	100 0111
H	72	48	100 1000
I	73	49	100 1001
J	74	4A	100 1010
K	75	4B	100 1011
L	76	4C	100 1100
M	77	4D	100 1101
N	78	4E	100 1110
O	79	4F	100 1111
P	80	50	101 0000
Q	81	51	101 0001
R	82	52	101 0010
S	83	53	101 0011
T	84	54	101 0100
U	85	55	101 0101
V	86	56	101 0110
W	87	57	101 0111
X	88	58	101 1000
Y	89	59	101 1001
Z	90	5A	101 1010

ASCII CHARACTER	DECIMAL	HEXADECIMAL	BINARY
[	91	5B	101 1011
\	92	5C	101 1100
]	93	5D	101 1101
^	94	5E	101 1110
_	95	5F	101 1111
`	96	60	110 0000
a	97	61	110 0001
b	98	62	110 0010
c	99	63	110 0011
d	100	64	110 0100
e	101	65	110 0101
f	102	66	110 0110
g	103	67	110 0111
h	104	68	110 1000
i	105	69	110 1001
j	106	6A	110 1010
k	107	6B	110 1011
l	108	6C	110 1100
m	109	6D	110 1101
n	110	6E	110 1110
o	111	6F	110 1111
p	112	70	111 0000
q	113	71	111 0001
r	114	72	111 0010
s	115	73	111 0011
t	116	74	111 0100
u	117	75	111 0101
v	118	76	111 0110
w	119	77	111 0111
x	120	78	111 1000
y	121	79	111 1001
z	122	7A	111 1010
{	123	7B	111 1011
|	124	7C	111 1100
}	125	7D	111 1101
~	126	7E	111 1110
DEL	127	7F	111 1111

Glossary

-- operator an operator used to decrement a variable.

++ operator an operator used to increment a variable.

A

accessor a method that allows data in an object to be accessed from outside the object.

ActionListener an event handler that is invoked when an event occurs for an object.

action method an event handler for events generated by key press operations.

actual parameter the name of a parameter that a method uses when calling another method.

alternate HTML HTML code that will be displayed by a Web browser that does not support Java.

American Standard Code for Information Interchange (ASCII) a standardized set of numeric values used to represent letters, symbols, and numeric values in a computer.

analog a device that uses quantities that are variable or exist in a range

appending adding new data characters to the end of an existing buffer.

applet a small Java program that can be included in Web pages to enhance the appearance or functionality of a Web page.

<APPLET> tag a tag that tells a Web browser a Java applet is included in the HTML document.

arithmetic operators operators used in mathematical calculations, such as addition, subtraction, division, or multiplication.

array an area of storage holding multiple variables of the same data type.

assembler a program that converts a programmer's statements into machine language code.

assembly language a programming language that is very close to machine language.

assignment operator used to change the value of a variable to the left of an equal sign to the value on the right side of the equal sign.

B

binary number system a number system that uses the digits 0 and 1 to represent data values.

bit a binary digit that can contain either a 0 or a 1.

Boolean variable a variable that has only two states, true or false.

BorderLayout a Java interface layout that divides the screen into five areas where objects can be placed.

braces characters used to mark the beginning and end of blocks of code.

branching a method a program uses to test a condition and jump to the appropriate function.

break a keyword that can be used to stop processing of a loop or a switch structure.

buffer an area used to temporarily store data while it is being processed.

bus electronic circuits used to move data and instruction between the microprocessor chip and RAM, ROM, and peripheral devices.

byte a combination of 8 bits used to represent a single character.

byte codes Java instructions that have been partially compiled and can then be interpreted by a Java virtual machine.

byte variable an 8-bit area used to hold an integer value.

C

called method a method that is called by another method.

calling method a method that originates a call to another method.

capacity the maximum size or quantity an object can hold.

case a keyword used to handle a specific condition in a switch structure.

case sensitive interpreting characters differently based on the case (upper or lower) of the characters.

casting indicating to the compiler that a variable should be treated as a certain data type even if it looks like a different type to the compiler. Also called *type casting*.

catch process a block of code that catches program exceptions and reports them.

char variable a 16-bit area used to hold a single character.

character an alphabetic letter, symbol, or numeral that can be processed by a computer.

check boxes an interface object that can be checked to indicate it is selected.

CheckBoxGroup a set of checkboxes that only allows one box to be checked at a time.

class a set of instructions that create an object.

class wrapper an object-oriented data type. Also called *object wrapper*.

client computer a computer running a Web browser and requesting Web pages.

close a process used when finished working with a file.

comments nonexecutable statements in a program that can be used to document the purpose of the program or to track changes to the program.

comparison operators operators used to create expressions to evaluate data or variables.

compiler a program that translates a high-level language into machine language code.

components things that can make up a program, such as buttons, check boxes, etc.

compound operators special operators that provide a shorthand notation for modifying variables.

constant a storage area that retains its value for the duration of the program's run.

constraint conditions that an object must abide by.

constructor a function that creates an object based on the rules of the object's class.

containers an object derived from a component that can hold things.

context an environment, or theoretical place.

continue a keyword that stops the processing of a loop pass but continues the loop with the next iteration.

control expression an expression that makes a decision.

copy constructor a constructor that takes the information from an object and puts it in a newly created instance of the object.

D

data a computer's representation of something that exists in the real world, such as names and addresses, dollar amounts, letters, spreadsheets, etc.

data hiding a means of preventing programmers from accessing some variables.

data type a way to specify what type of data is held in a variable or a constant.

debugging a step-by-step method of testing a program and correcting programming errors.

decimal number system a number system that uses the digits 0 through 9 to represent data values.

declare the process of telling a compiler the name of a variable or constant and the type of data it will contain.

decrementing subtracting 1 from the value of a variable.

default constructor a constructor that sets a class to its default values.

destroy method a method within an applet called when the Web browser exits.

digital a device that uses switches (or digits) in combination to represent something in the real world.

do while loop a loop that always executes at least once and continues to execute until a condition is met.

dot operator a period used to associate a member with a method or class.

double buffer a method of reducing on-screen flicker when animating objects.

double variable a 64-bit area used to hold floating point values.

E

"E" notation a method for expressing very large and very small numeric values.
element an entry in an array.

encapsulation hiding data and code within a class.

event an interaction that requires a computer program to perform processing.

event listener an object in a program that waits for another object to generate an event.

event manager (handler) a program that is given control when an event occurs and transfers control to a program capable of processing the event.

event source an object in a program that generates, or initiates, an event.

exception a situation contrary to the normal flow of processing.

Exception classes Java programs that provide the code necessary to handle various types of exceptions.

exception handling a system designed to deal with program exceptions.

executable file a program file that can be run repeatedly without having to translate the program each time.

exponential notation also called *scientific notation*, a method of specifying very large and very small values.

expression the portion of a programming statement on the right side of an assignment operator.

F

file stream a method of accessing files using Java.

float variable a 32-bit area used to hold floating-point values

floating-point unit the area of a computer where math operations are performed.

floppy disk a storage device containing a magnetic disk used to hold small quantities of data and/or programs.

FlowLayout a simple Java interface layout where objects placed on the screen flow from left to right, top to bottom, as they are placed.

flush the process of forcing all data for a file from a buffer.

for loop a loop that repeats one or more statements a specific number of times.

formal parameter the name of a parameter as it is defined in the method processing the parameter.

fuzzy logic a system that allows for more than a true or false condition.

G

garbage collection the process of retrieving memory and resources no longer being used.

global variable a variable that is in scope as long as the class is active.

graphical user interface (GUI) a method of communicating with a computer by manipulating pictures and icons.

GridBagLayout a Java interface that allows objects of different shapes and sizes.

GridLayout a Java interface that must hold equally shaped objects in rows and columns.

H

handleEvent an event handler that handles any event not handled by a specific event handler.

hard coded a value, string, or expression coded into a program that can be externally changed when the program runs.

hard disk a storage device consisting of one or more magnetic platters used to permanently store programs and data files.

hardware the physical devices and components that make up a computer.

high-level language an English-like or easy-to-write language that uses instructions that do not necessarily correspond one-to-one with a computer's instruction set.

HTML (Hypertext Markup Language) a programming language used to create Web pages.

HTTP (Hypertext Transfer Protocol) a communications protocol used by Web servers to transmit HTML and other document types to a Web browser.

I

identifier a name given to a variable or a constant.

if structure a structure that executes one or more programming statements if a condition is true.

if/else structure a structure that executes one or more programming statements if a condition is true, or a different statement or statements if the condition is false.

image a graphic representation of an object.

implicitly an inference, or assumption, based on the most likely possibility.

incrementing adding one to the value of a variable.

infinite loop a program loop that executes indefinitely.

inheritance a term that refers to a child object inheriting the properties of its parent object.

init method an applet method that is given control when the page containing an applet is loaded.

initialize the process of setting a variable to its starting value.

initializing constructor a constructor that initializes a class with values other than the default values.

initializing expression a loop parameter that sets a counter to a starting value.

inner class a class defined within a class that has access to all objects within the primary class.

input data data that is to be processed by a computer

instantiate to declare an object.

interact the process of getting information and providing a response.

interface classlike groups of methods that can work with a program by inheriting from their parent classes.

interpreter a program that translates a programs source code or byte codes into machine language while the program is being executed.

intrinsic data types the eight standard data types supported by Java.

int variable a 32-bit area used to hold integer values.

iteration a single loop through a block of program statements.

iteration structure a series of structures that perform loops.

J

Java a programming language originally created for specific-purpose computers that has now found widespread acceptance in general-purpose computers.

Java virtual machine (JVM) interprets Java byte codes into a machine language instruction that can be executed on the platform running the Java program.

K

keyword a word reserved for use by the Java language.

L

linker a program that links multiple software object files into a single executable module.

local variable a variable that is only in scope within the method it is defined in.

logical operators operators that allow the use of and, or, and not to be used in an expression.

long variable a 64-bit area used to hold integer values.

lookup a portion of a program that is used to find a specific entry in an array.

loop a process where a program repeats a series of statements a specific number of times

low-level language a programming language that is very precise and non-English-like, such as machine language or assembly language.

lowercase the non-capital letters of the alphabet.

M

machine language a series of numeric values that a computer interprets as program instructions and addresses.

main method the primary method, or routine, in a Java application program.

math coprocessor the area of a computer's processor where math operations are performed.

member a method defined as part of a class of objects.

member function a method that is a member of a class.

menu a means of selecting from more than one option or function.

method one or more statements in a class performing a specific task within an object.

method overloading creating multiple methods with the same name within a class.

microprocessor a circuit board that controls all processing within a small computer.

modem a device that connects a computer to a telephone line and can be used for sending and receiving data.

modifier also called a *mutator*, a method of modifying data within an object.

modulus operator an arithmetic operator that returns the remainder value in a divide operation.

multidimensional array an array that can hold items in a two- or three-dimensional grid.

mutator a modifier used to make changes to data inside an object.

N

nested loop a loop contained within another loop.

nested structure a programming structure contained within another structure.

new a keyword used to allocate space for an object or variable.

O

Oak the original name for the Java language.

object an instance of a class containing data and the functions that manipulate the data.

object code machine language code that results from assembling or compiling a source code program file.

object file the result of processing a program source file through a compiler.

object-oriented programming (OOP) building programs by creating, controlling, and modifying one or more objects.

object wrappers a set of object-oriented data types within Java.

one-way selection structure a structure that allows to go only one way if a tested condition is met.

open the first step required to use a file for input or for output.

open mode a means of specifying if a random-access file is to be used for input or for output.

operating system computer software that controls the operation of a computer.

order of operations the order in which a mathematical expression is evaluated.

output the data resulting from the processing of input.

overflow a condition that occurs when a variable becomes too large for its defined type.

overloading using a constructor in more than one way.

overriding a method in a child class that has the same name as a method in a parent class.

P

packages groups of classes that are available for use by a compiler.

paint method a method within an applet that repaints the screen each time the program is updated.

panel a container and an object that can hold containers.

<PARAM> tag an HTML tag that can be used to pass parameters to a Java applet.

parameters information that can be passed to a Java applet that can be used to customize the applet.

pixel a very small dot printed on a computer screen representing a chunk of information about some object.

primary storage internal memory called RAM that is the place where the computer stores active programs and data being processed.

primitive data types basic data types, such as Boolean, int, float, etc.

procedural programming creating a program by using a step-by-step process to perform specific tasks.

process a task being performed in a multitasking environment.

programming language a means of providing processing instructions to a computer without having to learn machine language.

promotion a process of temporarily changing a variable of one type to another type to perform a math operation.

prompt a method for asking a user for input.

R

radio button another name for a CheckBoxGroup check box.

RAM (random access memory) the primary storage area in a computer used to hold programs and data being processed.

random-access file a file consisting of equal length records or data that can be read in any sequence.

reading data the process of retrieving data from a file used as an input file.

relational database a collection of multiple files containing various types of information related to each other by some common field or data.

robust program code code that does everything it is supposed to do and also attempts to handle any unusual conditions.

ROM (read-only memory) memory circuits that have data and programs permanently stored on them, normally used to start up the computer.

S

scope a term that describes the location where a variable is alive.

secondary storage storage devices that retain data even when power is turned off, such as hard disks, floppy disks, or CD-ROMs.

seed a value given to a randomizing class to use as a starting point when generating a random number.

selection structures structures in a Java program that make decisions.

self-document using variable names to indicate the purpose or content of a variable or constant.

sequence structures structures that execute one program statement after another without changing the program's flow.

sequential-access file a method of reading a file whereby data is read or written from the beginning of the file to the end of the file.

short-circuit evaluation terminating processing of an expression if a required condition fails its test.

short variable a 16-bit area used to hold integer values.

size how many items are contained in an array or vector class.

source code a program in its native form, before being assembled or compiled.

start method a method within an applet that is called after the init method completes and every time that the page containing the applet is loaded.

statements instructions or commands within a method that make a program work.

states the conditions that an object can exist in, such as on or off.

static a keyword that causes only one variable to be created and hold one piece of data for all the objects.

step expression the part of a for loop that modifies a counter variable.

stop method a method within an applet that is called every time a user leaves the page containing the applet.

stream a place where data is contained during processing, or the flow of data within a program.

string a group of characters representing data.

subscript a numeric value or set of values used to reference a particular entry in a array.

substring a smaller portion, or segment, of a string.

super a keyword that allows a Java class to access its parent's methods or data.

switch structure a structure capable of handling multiple options.

T

tags a word to describe the delimiters used in the HTML language.

text editor a program that can be used to create a program by entering text strings and values.

text file a file saved by a text editor.

this a keyword that implicitly tells Java which object is being referenced.

thread a portion of a process that can function independently.

throwing an exception a program's method of reporting an exception and an attempt to explain the exception.

truncate the loss of precision or data values because a variable is larger or smaller than the field defined for the value.

truth tables a means of illustrating the results of logical operators.

try process a block of code that examines data to see if it is valid.

two-way selection structure a structure that allows two ways to proceed if a condition is met, or not met.

U

underflow a condition that occurs when a variable is too small for its defined type.

Unicode a standard that defines how characters and symbols are represented in a computer.

uppercase the capital letters of the alphabet.

V

variable an area that holds data that can be modified during program execution.

vector a Java class that can hold multiple objects and is dynamically expandable.

volatile a term to describe a computer component, such as RAM, that cannot retain its contents when power is shut off.

W

Web server a computer connected to the Internet that hosts Web pages and transmits requested pages to client computers.

while loop a loop that continues to execute until a specified condition is met.

writing data the process of placing data in a file being used as an output file.

Index

C

D

E

F

P

R

S

T

U