

2.1 Designing a Program

CONCEPT: Programs must be carefully designed before they are written. During the design process, programmers use tools such as pseudocode and flowcharts to create models of programs.

In [Chapter 1](#) you learned that programmers typically use high-level languages to write programs. However, all professional programmers will tell you that a program should be carefully designed before the code is actually written. When programmers begin a new project, they never jump right in and start writing code as the first step. They begin by creating a design of the program.

After designing the program, the programmer begins writing code in a high-level language. Recall from [Chapter 1](#) that each language has its own rules, known as syntax, that must be followed when writing a program. A language's syntax rules dictate things such as how key words, operators, and punctuation characters can be used. A syntax error occurs if the programmer violates any of these rules.

If the program contains a syntax error, or even a simple mistake such as a misspelled key word, the compiler or interpreter will display an error message indicating what the error is. Virtually all code contains syntax errors when it is first written, so the programmer will typically spend some time correcting these. Once all of the syntax errors and simple typing mistakes have been corrected, the program can be compiled and translated into a machine language program (or executed by an interpreter, depending on the language being used).

Once the code is in an executable form, it is then tested to determine whether any logic errors exist. A *logic error* is a mistake that does not prevent the program from running, but causes it to produce incorrect results. (Mathematical mistakes are common causes of logic errors.)

If there are logic errors, the programmer *debugs* the code. This means that the programmer finds and corrects the code that is causing the error. Sometimes during this process, the programmer discovers that the original design must be changed. This entire process, which is known as the *program development cycle*, is repeated until no errors can be found in the program. [Figure 2-1](#) shows the steps in the process.

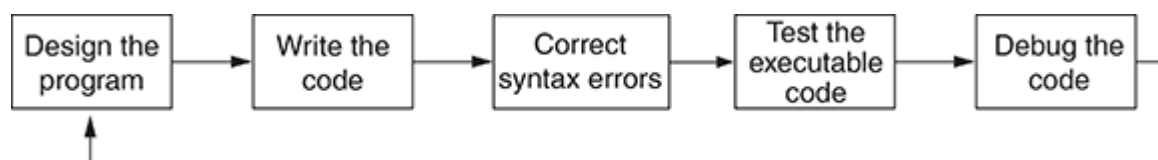


Figure 2-1 The program development cycle

This book focuses entirely on the first step of the program development cycle: designing the program. The process of designing a program is arguably the most important part of the cycle. You can think of a program's design as its foundation. If you build a house on a poorly constructed foundation, eventually you will find yourself doing a lot of work to fix the house! A program's design should be viewed no differently. If your program is designed poorly, eventually you will find yourself doing a lot of work to fix the program.

Designing a Program

The process of designing a program can be summarized in the following two steps:

1. Understand the task that the program is to perform.
2. Determine the steps that must be taken to perform the task.

Let's take a closer look at each of these steps.

Understand the Task That the Program Is to Perform

It is essential that you understand what a program is supposed to do before you can determine the steps that the program will perform. Typically, a professional programmer gains this understanding by working directly with the customer. We use the term *customer* to describe the person, group, or organization that is asking you to write a program. This could be a customer in the traditional sense of the word, meaning someone who is paying you to write a program. It could also be your boss, or the manager of a department within your company. Regardless of who it is, the customer will be relying on your program to perform an important task.

To get a sense of what a program is supposed to do, the programmer usually interviews the customer. During the interview, the customer will describe the task that the program should perform, and the programmer will ask questions to uncover as many details as possible about the task. A follow-up interview is usually needed because customers rarely mention everything they want during the initial meeting, and programmers often think of additional questions.

The programmer studies the information that was gathered from the customer during the interviews and creates a list of different software requirements. A *software requirement* is simply a single function that the program must perform in order to satisfy the customer. Once the customer agrees that the list of requirements is complete, the programmer can move to the next phase.



TIP:

If you choose to become a professional software developer, your customer will be anyone who asks you to write programs as part of your job. As long as you are a student, however, your customer is your instructor! In every programming class that you will take, it's practically guaranteed that your instructor will assign programming problems for you to complete. For your academic success, make sure that you understand your instructor's requirements for those assignments and write your programs accordingly.

Determine the Steps That Must Be Taken to Perform the Task

Once you understand the task that the program will perform, you begin by breaking down the task into a series of steps. This is similar to the way you would break down a task into a series of steps that another person can follow. For example, suppose your little sister asks you how to boil water. Assuming she is old enough to be trusted around the stove, you might break down that task into a series of steps as follows:

1. Pour the desired amount of water into a pot.
2. Put the pot on a stove burner.
3. Turn the burner to high.
4. Watch the water until you see large bubbles rapidly rising. When this happens, the water is boiling.

This is an example of an *algorithm*, which is a set of well-defined logical steps that must be taken to perform a task. Notice that the steps in this algorithm are sequentially ordered. Step 1 should be performed before Step 2, and so on. If your little sister follows these steps exactly as they appear, and in the correct order, she should be able to boil water successfully.

A programmer breaks down the task that a program must perform in a similar way. An algorithm is created, which lists all of the logical steps that must be taken. For example, suppose you have been asked to write a program to calculate and display the gross pay for an hourly paid employee. Here are the steps that you would take:

1. Get the number of hours worked.
2. Get the hourly pay rate.
3. Multiply the number of hours worked by the hourly pay rate.
4. Display the result of the calculation that was performed in Step 3.

Of course, this algorithm isn't ready to be executed on the computer. The steps in this list have to be translated into code. Programmers commonly use two tools to help them accomplish this: pseudocode and flowcharts. Let's look at each of these in more detail.

Pseudocode

Recall from [Chapter 1](#) that each programming language has strict rules, known as syntax, that the programmer must follow when writing a program. If the programmer writes code that violates these rules, a syntax error will result and the program cannot be compiled or executed. When this happens, the programmer has to locate the error and correct it.

Because small mistakes like misspelled words and forgotten punctuation characters can cause syntax errors, programmers have to be mindful of such small details when writing code. For this reason, programmers find it helpful to write their programs in pseudocode

(pronounced "sue doe code") before they write it in the actual code of a programming language.

The word *pseudo* means fake, so *pseudocode* is fake code. It is an informal language that has no syntax rules, and is not meant to be compiled or executed. Instead, programmers use pseudocode to create models, or "mock-ups" of programs. Because programmers don't have to worry about syntax errors while writing pseudocode, they can focus all of their attention on the program's design. Once a satisfactory design has been created with pseudocode, the pseudocode can be translated directly to actual code.

Here is an example of how you might write pseudocode for the pay calculating program that we discussed earlier:

```
Display "Enter the number of hours the employee worked."  
Input hours  
Display "Enter the employee's hourly pay rate."  
Input payRate  
Set grossPay = hours * payRate  
Display "The employee's gross pay is $", grossPay
```

Each statement in the pseudocode represents an operation that can be performed in any high-level language. For example, all languages provide a way to display messages on the screen, read input

that is typed on the keyboard, and perform mathematical calculations. For now, don't worry about the details of this particular pseudocode program. As you progress through this chapter you will learn more about each of the statements that you see here.



NOTE:

As you read the examples in this book, keep in mind that pseudocode is not an actual programming language. It is a generic way to write the statements of an algorithm, without worrying about syntax rules. If you mistakenly write pseudocode into an editor for an actual programming language, such as Python or Visual Basic, errors will result.

Flowcharts

Flowcharting is another tool that programmers use to design programs. A *flowchart* is a diagram that graphically depicts the steps that take place in a program. [Figure 2-2](#) shows how you might create a flowchart for the pay calculating program.

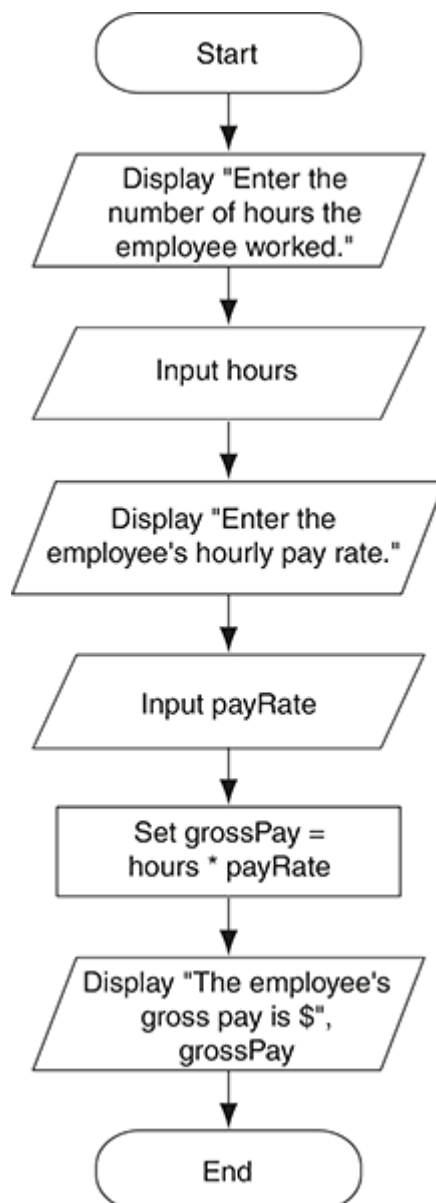


Figure 2-2 Flowchart for the pay calculating program

Notice that there are three types of symbols in the flowchart: ovals, parallelograms, and rectangles. The ovals, which appear at the top and bottom of the flowchart, are called *terminal symbols*. The *Start* terminal symbol marks the program's starting point and the *End* terminal symbol marks the program's ending point.

Between the terminal symbols are parallelograms, which are used for both *input symbols* and *output symbols*, and rectangles, which are called *processing symbols*. Each of these symbols represents a step in the program. The symbols are connected by arrows that represent the "flow" of the program. To step through the symbols in the proper order, you begin at the *Start* terminal and follow the arrows until you reach the *End* terminal. Throughout this chapter we will look at each of these symbols in greater detail. For your reference, [Appendix B](#) summarizes all of the flowchart symbols that we use in this book.

There are a number of different ways that you can draw flowcharts, and your instructor will most likely tell you the way that he or she prefers you to draw them in class. Perhaps the simplest and least expensive way is to simply sketch the flowchart by hand with pencil and paper. If you need to make your hand-drawn flowcharts look more professional, you can visit your local office supply store (or possibly your campus bookstore) and purchase a flowchart template, which is a small plastic sheet that has the flowchart symbols cut into it. You can use the template to trace the symbols onto a piece of paper.

The disadvantage of drawing flowcharts by hand is that mistakes have to be manually erased, and in many cases, require that the entire page be redrawn. A more efficient and professional way to create flowcharts is to use software. There are several specialized software packages available that allow you to create flowcharts.

Flowchart Connector Symbols

Often, a flowchart is too long to fit on a page. Sometimes you can remedy this by breaking the flowchart into two or more smaller flowcharts, and placing them side-by-side on the page. When you do this, you use a *connector symbol* to connect the pieces of the flowchart. A connector symbol is a small circle with a letter or number written inside it. [Figure 2-3](#) shows an example of a flowchart with a connector symbol.

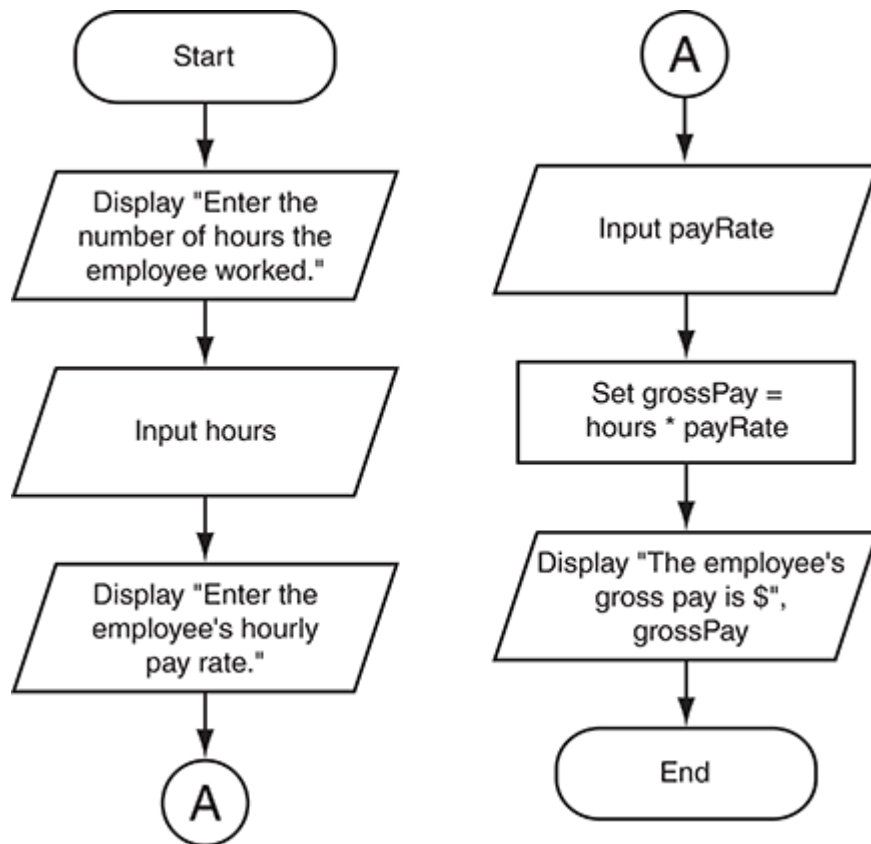



Figure 2-3 Flowchart with a connector symbol

In **Figure 2-3**, the  connector symbol indicates that the second flowchart segment begins where the first flowchart segment ends.

When a flowchart is simply too large to fit on a single page, you can break the flowchart into parts, and place the parts on separate pages. You then use the *off-page connector symbol* to connect the pieces of the flowchart. The off-page connector symbol is the “home plate” shape, with a page number shown inside it. If the connector is at an exit point in the flowchart, the number indicates the page where the next part of the flowchart is located. If the connector is at an entry point in the flowchart, it indicates the page where the previous part of the flowchart is located. **Figure 2-4** shows an example. In the figure, the flowchart on the left is on page **1**. At the bottom of the flowchart is an off-page connector indicating that the flowchart continues on page **2**. In the flowchart on the right (which is page **2**), the off-page connector at the top indicates that the previous part of the flowchart is on page **1**.

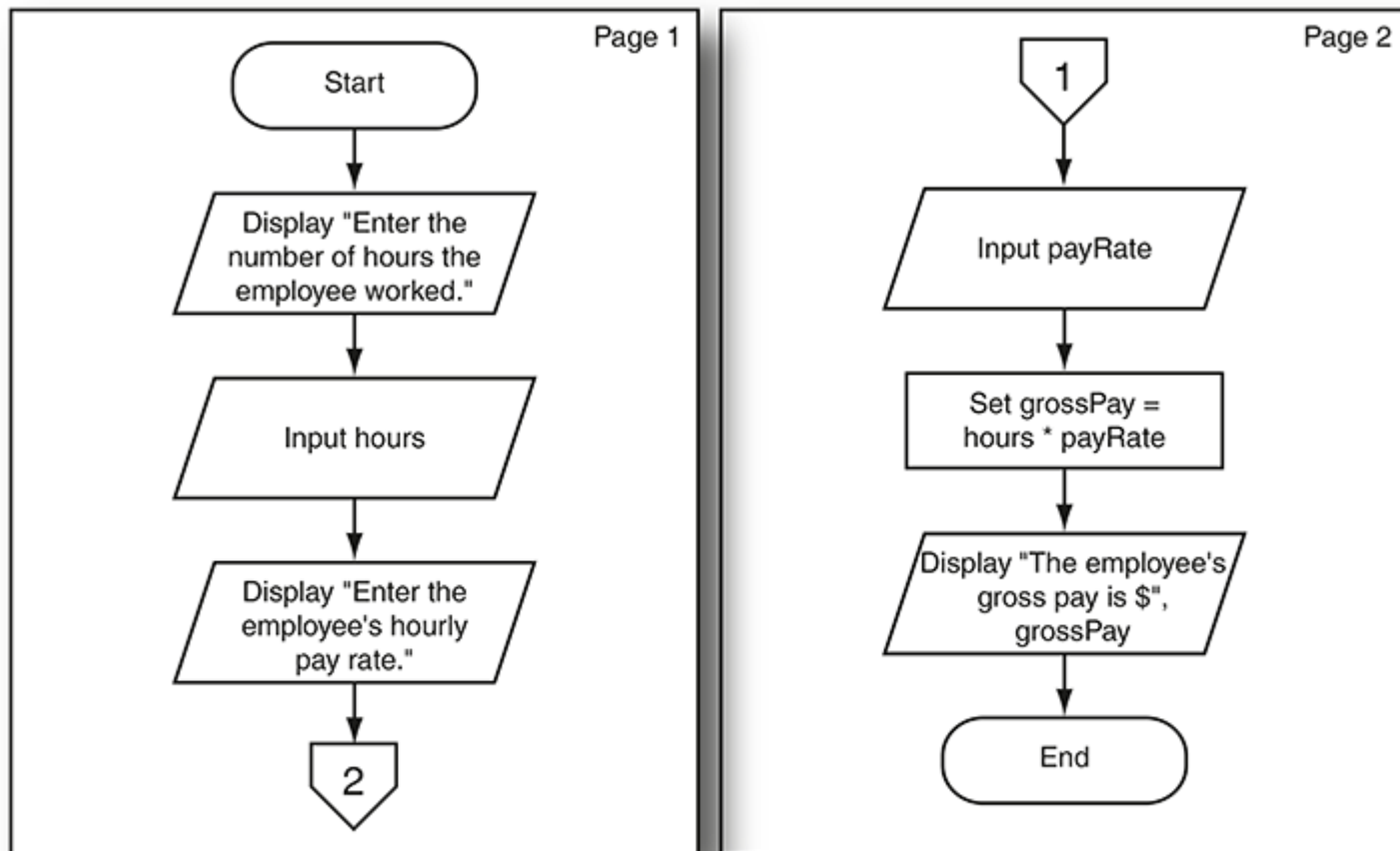


Figure 2-4 Flowchart with an off-page connector



NOTE:

Flowcharting symbols and techniques can vary from one book to another, or from one software package to another. If you are using specialized software to draw flowcharts, you might notice slight differences between some of the symbols that it uses, compared to some of the symbols used in this book.



Checkpoint

- 2.1 Who is a programmer's customer?
- 2.2 What is a software requirement?
- 2.3 What is an algorithm?
- 2.4 What is pseudocode?
- 2.5 What is a flowchart?
- 2.6 What are each of the following symbols in a flowchart?
 - Oval
 - Parallelogram
 - Rectangle

2.2 Output, Input, and Variables

CONCEPT: Output is data that is generated and displayed by the program. Input is data that the program receives. When a program receives data, it stores it in variables, which are named storage locations in memory.

Computer programs typically perform the following three-step process:

- 1. Input is received.
- 2. Some process is performed on the input.
- 3. Output is produced.

Input is any data that the program receives while it is running. One common form of input is data that is typed on the keyboard. Once input is received, some process, such as a mathematical calculation, is usually performed on it. The results of the process are then sent out of the program as output.

Figure 2-5 illustrates these three steps in the pay calculating program that we discussed earlier. The number of hours worked and the hourly pay rate are provided as input.

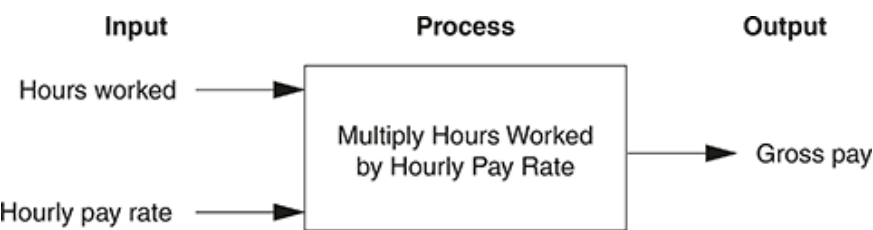


Figure 2-5 The input, processing, and output of the pay calculating program

The program processes this data by multiplying the hours worked by the hourly pay rate. The results of the calculation are then displayed on the screen as output.

IPO Charts

An IPO chart is a simple but effective tool that programmers commonly use while designing programs. IPO stands for *input*, *processing*, and *output*, and an *IPO chart* describes the input, processing, and output of a program. These items are usually laid out in columns. The input column shows a description of the data that is required as input. The processing column shows a description of the process, or processes, that the program performs. The output column describes the output that is produced by the program. For example, Figure 2-6 shows an IPO chart for the pay calculating program.

IPO Chart for the Pay Calculating Program		
Input	Processing	Output
Number of hours worked Hourly pay rate	Multiply the number of hours worked by the hourly pay rate. The result is the gross pay.	Gross pay

Figure 2-6 IPO chart for the pay calculating program

In the remainder of this section, you will look at some simple programs that perform output and input. In the next section, we will discuss how to process data.

Displaying Screen Output

Perhaps the most fundamental thing that you can do in a program is to display a message on the computer screen. As previously mentioned, all high-level languages provide a way to display screen output. In this book, we use the word *Display* to write pseudocode statements for displaying output on the screen. Here is an example:

```
Display "Hello world"
```

The purpose of this statement is to display the message *Hello world* on the screen. Notice that after the word *Display*, we have written *Hello world* inside quotation marks. The quotation marks are not to be displayed. They simply mark the beginning and the end of the text that we wish to display.

Suppose your instructor tells you to write a pseudocode program that displays your name and address on the computer screen. The pseudocode shown in **Program 2-1** is an example of such a program.

Program 2-1

```
Display "Kate Austen"  
Display "1234 Walnut Street"  
Display "Asheville, NC 28899"
```

It is important for you to understand that the statements in this program execute in the order that they appear, from the top of the program to the bottom. This is shown in **Figure 2-7**. If you translated this pseudocode into an actual program and ran it, the first statement would execute, followed by the second statement, and followed by the third statement. If you try to visualize the way this program's output would appear on the screen, you should imagine something like that shown in **Figure 2-8**. Each *Display* statement produces a line of output.

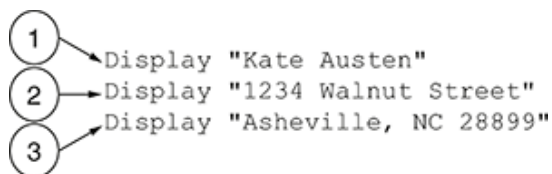


Figure 2-7 The statements execute in order

(Courtesy of Microsoft Corporation)

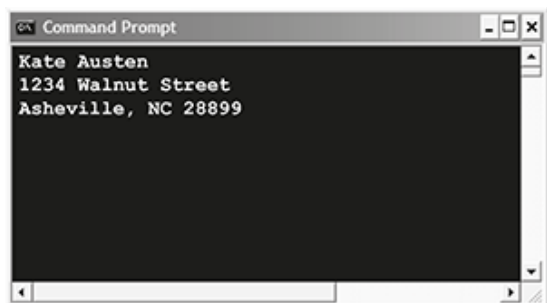


Figure 2-8 Output of **Program 2-1**

(Courtesy of Microsoft Corporation)

**NOTE:**

Although this book uses the word *Display* for an instruction that displays screen output, some programmers use other words for this purpose. For example, some programmers use the word *Print*, and others use the word *Write*. Pseudocode has no rules that dictate the words that you may or may not use.

Figure 2-9 shows the way you would draw a flowchart for this program. Notice that between the *Start* and *End* terminal symbols there are three parallelograms. A parallelogram can be either an output symbol or an input symbol. In this program, all three parallelograms are output symbols. There is one for each of the *Display* statements.

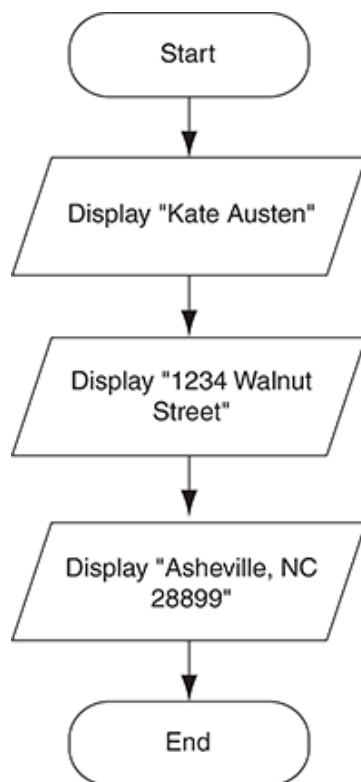


Figure 2-9 Flowchart for **Program 2-1**

Sequence Structures

It was mentioned earlier that the statements in **Program 2-1** execute in the order that they appear, from the top of the program to the bottom. A set of statements that execute in the order that they appear is called a *sequence structure*. In fact, all of the programs that you will see in this chapter are sequence structures.

A *structure*, also called a *control structure*, is a logical design that controls the order in which a set of statements executes. In the 1960s, a group of mathematicians proved that only three program structures are needed to write any type of program. The simplest of these structures is the sequence structure. Later in this book, you will learn about the other two structures—decision structures and repetition structures.

Strings and String Literals

Programs almost always work with data of some type. For example, **Program 2-1** uses the following three pieces of data:

```
"Kate Austen"  
"1234 Walnut Street"  
"Asheville, NC 28899"
```

These pieces of data are sequences of characters. In programming terms, a sequence of characters that is used as data is called a *string*. When a string appears in the actual code of a program (or in pseudocode, as it does in **Program 2-1**) it is called a *string literal*. In program code, or pseudocode, a string literal is usually enclosed in quotation marks. As mentioned earlier, the quotation marks simply mark where the string begins and ends.

In this book, we will always enclose string literals in double quote marks (`"`). Most programming languages use this same convention, but a few use single quote marks (`'`).

Variables and Input

Quite often a program needs to store data in the computer's memory so it can perform operations on that data. For example, consider the typical online shopping experience: You browse a Web site and add the items that you want to purchase to the shopping cart. As you add items to the shopping cart, data about those items is stored in memory. Then, when you click the checkout button, a program running on the Web site's computer calculates the total of all the items you have in your shopping cart, applicable sales taxes, shipping costs, and the total of all these charges. When the program performs these calculations, it stores the results in the computer's memory.



Variables and Input

Programs use variables to store data in memory. A *variable* is a storage location in memory that is represented by a name. For example, a program that calculates the sales tax on a purchase might use a variable named `tax` to hold that value in memory. And a program that calculates the distance from Earth to a distant star might use a variable named `distance` to hold that value in memory.

In this section, we will discuss a basic input operation: reading data that has been typed on the keyboard. When a program reads data from the keyboard, usually it stores that data in a variable so it can be used later by the program. In pseudocode we will read data from the keyboard with the *Input* statement. As an example, look at the following statement, which appeared earlier in the pay calculating program:

```
Input hours
```

The word *Input* is an instruction to read a piece of data from the keyboard. The word `hours` is the name of the variable in which that data will be stored. When this statement executes, two things happen:

- The program pauses and waits for the user to type something on the keyboard, and then press the **Enter** key.
- When the **Enter** key is pressed, the data that was typed is stored in the `hours` variable.

Program 2-2 is a simple pseudocode program that demonstrates the *Input* statement. Before we examine the program, we should mention a couple of things. First, you will notice that each line in the program is numbered. The line numbers are not part of the pseudocode. We will refer to the line numbers later to point out specific parts of the program. Second, the program's output is shown immediately following the pseudocode. From now on, all pseudocode programs will be shown this way.

Program 2-2

```
1 Display "What is your age?"
2 Input age
3 Display "Here is the value that you entered:"
4 Display age
```

Program Output (with Input Shown in Bold)

```
What is your age?
24 [Enter]
Here is the value that you entered:
24
```

The statement in line 1 displays the string *"What is your age?"* Then, the statement in line 2 waits for the user to type a value on the keyboard and press **Enter**. The value that is typed will be stored in the *age* variable. In the example execution of the program, the user has entered 24. The statement in line 3 displays the string *"Here is the value that you entered:"* and the statement in line 4 displays the value that is stored in the *age* variable.

Notice that in line 4 there are no quotation marks around *age*. If quotation marks were placed around *age*, it would have indicated that we want to display the word *age* instead of the contents of the *age* variable. In other words, the following statement is an instruction to display the contents of the *age* variable:

```
Display age
```

This statement, however, is an instruction to display the word *age*:

```
Display "age"
```



NOTE:

In this section, we have mentioned the user. The *user* is simply any hypothetical person that is using a program and providing input for it. The user is sometimes called the *end user*.

Figure 2-10 shows a flowchart for **Program 2-2**. Notice that the *Input* operation is also represented by a parallelogram.

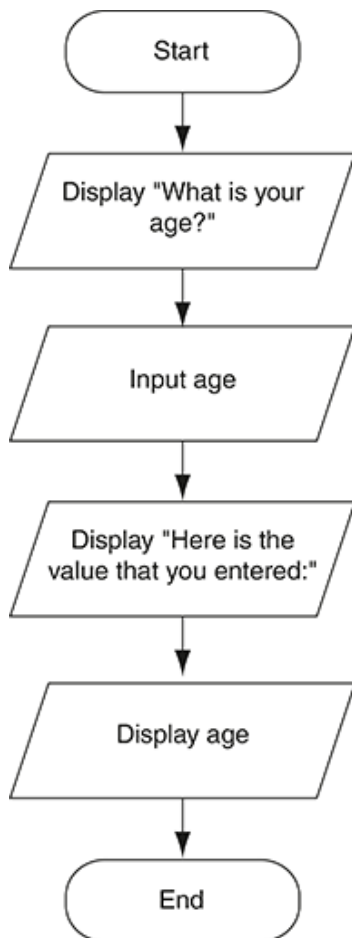


Figure 2-10 Flowchart for **Program 2-2**

Variable Names

All high-level programming languages allow you to make up your own names for the variables that you use in a program. You don't have complete freedom in naming variables, however. Every language has its own set of rules that you must abide by when creating variable names.

Although the rules for naming variables differ slightly from one language to another, there are some common restrictions:

- Variable names must be one word. They cannot contain spaces.
- In most languages, punctuation characters cannot be used in variable names. It is usually a good idea to use only alphabetic letters and numbers in variable names.
- In most languages, the first character of a variable name cannot be a number.

In addition to following the programming language rules, you should always choose names for your variables that give an indication of what they are used for. For example, a variable that holds the temperature might be named *temperature*, and a variable that holds a car's speed might be named *speed*. You may be tempted to give variables names like *x* and *b2*, but names like these give no clue as to what the variable's purpose is.

Because a variable's name should reflect the variable's purpose, programmers often find themselves creating names that are made of multiple words. For example, consider the following variable names:

```
grosspay
payrate
hotdogssoldtoday
```

Unfortunately, these names are not easily read by the human eye because the words aren't separated. Because we can't have spaces in variable names, we need to find another way to separate the words in a multiword variable name, and make it more readable to the human eye.

One way to do this is to use the underscore character to represent a space. For example, the following variable names are easier to read than those previously shown:

```
gross_pay
pay_rate
hot_dogs_sold_today
```

Another way to address this problem is to use the *camelCase* naming convention. camelCase names are written in the following manner:

- You begin writing the variable name with lowercase letters.
- The first character of the second and subsequent words is written in uppercase.

For example, the following variable names are written in camelCase:

```
grossPay
payRate
hotDogsSoldToday
```

Because the camelCase convention is very popular with programmers, we will use it from this point forward. In fact, you have already seen several programs in this chapter that use camelCase variable names. The pay calculating program shown at the beginning of the chapter uses the variable name *payRate*. Later in this chapter, **Program 2-9** uses the variable names *originalPrice* and *salePrice*, and **Program 2-11** uses the variable names *futureValue* and *presentValue*.



NOTE:

This style of naming is called camelCase because the uppercase characters that appear in a name are sometimes reminiscent of a camel's humps.

Displaying Multiple Items with One *Display* Statement

If you refer to **Program 2-2** you will see that we used the following two *Display* statements in lines 3 and 4:

```
Display "Here is the value that you entered:"
Display age
```

We used two *Display* statements because we needed to display two pieces of data. Line 3 displays the string literal *"Here is the value that you entered:"* and line 4 displays the contents of the *age* variable.

Most programming languages provide a way to display multiple pieces of data with one statement. Because this is a common feature of programming languages, frequently we will write *Display* statements in our pseudocode that display multiple items. We will simply separate the items with a comma, as shown in line 3

of **Program 2-3**.

Program 2-3

```
1  Display "What is your age?"
2  Input age
3  Display "Here is the value that you entered: ", age
```

Program Output (with Input Shown in Bold)

```
What is your age?
24 [Enter]
Here is the value that you entered: 24
```

Take a closer look at line 3 of **Program 2-3**:

Display "Here is the value that you entered: ", age

Notice the space.

Notice that the string literal `"Here is the value that you entered: "` ends with a space. That is because in the program output, we want a space to appear after the colon, as shown here:

Here is the value that you entered: 24

Notice the space.

In most cases, when you are displaying multiple items on the screen, you want to separate those items with spaces between them. Most programming languages do not automatically print spaces between multiple items that are displayed on the screen. For example, look at the following pseudocode statement:

```
Display "January", "February", "March"
```

In most programming languages, such as statement would produce the following output:

```
JanuaryFebruaryMarch
```

To separate the strings with spaces in the output, the `Display` statement should be written as:

```
Display "January ", "February ", "March"
```

String Input

Programs 2-2 and **2-3** read numbers from the keyboard, which were stored in variables by `Input` statements. Programs can also read string input. For example, the pseudocode in **Program 2-4** uses two `Input` statements: one to read a string and one to read a number.

Program 2-4

```
1  Display "Enter your name."  
2  Input name  
3  Display "Enter your age."  
4  Input age  
5  Display "Hello ", name  
6  Display "You are ", age, " years old."
```

Program Output (with Input Shown in Bold)

```
Enter your name.  
Andrea [Enter]  
Enter your age.  
24 [Enter]  
Hello Andrea  
You are 24 years old.
```

The *Input* statement in line 2 reads input from the keyboard and stores it in the *name* variable. In the example execution of the program, the user entered Andrea. The *Input* statement in line 4 reads input from the keyboard and stores it in the *age* variable. In the example execution of the program, the user entered 24.

Prompting the User

Getting keyboard input from the user is normally a two-step process:

1. Display a prompt on the screen.
2. Read a value from the keyboard.

A *prompt* is a message that tells (or asks) the user to enter a specific value. For example, the pseudocode in **Program 2-3** gets the user to enter his or her age with the following statements:

```
Display "What is your age?"  
Input age
```

In most programming languages, the statement that reads keyboard input does not display instructions on the screen. It simply causes the program to pause and wait for the user to type something on the keyboard. For this reason, whenever you write a statement that reads keyboard input, you should also write a statement just before it that tells the user what to enter. Otherwise, the user will not know what he or she is expected to do. For example, suppose we remove line 1 from **Program 2-3**, as follows:

```
Input age  
Display "Here is the value that you entered: ", age
```

If this were an actual program, can you see what would happen when it is executed? The screen would appear blank because the *Input* statement would cause the program to wait for something to be typed on the keyboard. The user would probably think the computer was malfunctioning.

The term *user-friendly* is commonly used in the software business to describe programs that are easy to use. Programs that do not display adequate or correct instructions are frustrating to use, and are not considered user-friendly. One of the simplest things that you can do to increase a program's user-friendliness is to make sure that it displays clear, understandable prompts prior to each statement that reads keyboard input.



TIP:

Sometimes we computer science instructors jokingly tell our students to write programs as if “Uncle Joe” or “Aunt Sally” were the user. Of course, these are not real people, but imaginary users who are prone to making mistakes if not told exactly what to do. When you are designing a program, you should imagine that someone who knows nothing about the program's inner workings will be using it.



Checkpoint

- 2.7 What are the three operations that programs typically perform?
- 2.8 What is an IPO chart?
- 2.9 What is a sequence structure?
- 2.10 What is a string? What is a string literal?
- 2.11 A string literal is usually enclosed inside a set of what characters?
- 2.12 What is a variable?
- 2.13 Summarize three common rules for naming variables.
- 2.14 What variable naming convention do we follow in this book?
- 2.15 Look at the following pseudocode statement:

```
Input temperature
```

What happens when this statement executes?

- 2.16 Who is the user?
- 2.17 What is a prompt?
- 2.18 What two steps usually take place when a program prompts the user for input?
- 2.19 What does the term *user-friendly* mean?

2.3 Variable Assignment and Calculations

CONCEPT: You can store a value in a variable with an assignment statement. The value can be the result of a calculation, which is created with math operators.

Variable Assignment

In the previous section, you saw how the *Input* statement gets a value typed on the keyboard and stores it in a variable. You can also write statements that store specific values in variables. The following is an example, in pseudocode:

```
Set price = 20
```

This is called an assignment statement. An *assignment statement* sets a variable to a specified value. In this case, the variable *price* is set to the value 20. When we write an assignment statement in pseudocode, we will write the word *Set*, followed by the name of the variable, followed by an equal sign (=), followed by the value we want to store in the variable. The pseudocode in [Program 2-5](#) shows another example.

Program 2-5

```
1 Set dollars = 2.75
2 Display "I have ", dollars, " in my account."
```

Program Output

```
I have 2.75 in my account.
```

In line 1, the value 2.75 is stored in the *dollars* variable. Line 2 displays the message “I have 2.75 in my account.” Just to make sure you understand how the *Display* statement in line 2 is working, let’s walk through it. The word *Display* is followed by three pieces of data, so that means it will display three things. The first thing it displays is the string literal *"I have "*. Next, it displays the contents of the *dollars* variable, which is 2.75. Last, it displays the string literal *" in my account."*

Variables are called “variable” because they can hold different values while a program is running. Once you set a variable to a value, that value will remain in the variable until you store a different value in the variable. For example, look at the pseudocode in [Program 2-6](#).

Program 2-6

```
1 Set dollars = 2.75
2 Display "I have ", dollars, " in my account."
3 Set dollars = 99.95
4 Display "But now I have ", dollars, " in my account!"
```

Program Output

```
I have 2.75 in my account.
But now I have 99.95 in my account!
```

Line 1 sets the *dollars* variable to 2.75, so when the statement in line 2 executes, it displays “I have 2.75 in my account.” Then, the statement in line 3 sets the *dollars* variable to 99.95. As a result, the value 99.95 replaces the value 2.75 that was previously stored in the variable. When line 4 executes, it displays “But now I have 99.95 in my account!” This program illustrates two important characteristics of variables:

- A variable holds only one value at a time.
- When you store a value in a variable, that value replaces the previous value that was in the variable.



NOTE:

When writing an assignment statement, all programming languages require that you write the name of the variable that is receiving the value on the left side of For example, the following statement is incorrect:

Set 99.95 = dollars ← This is an error!
A statement such as this would be considered a syntax error.



NOTE:

In this book, we have chosen to start variable assignment statements with the word *Set* because it makes it clear that we are setting a variable to a value. In most programming languages, however, assignment statements do not start with the word *Set*. In most languages, an assignment statement looks similar to the following:

```
dollars = 99.95
```

If your instructor allows it, it is permissible to write assignment statements without the word *Set* in your pseudocode. Just be sure to write the name of the variable that is receiving the value on the left side of the equal sign.

In flowcharts, an assignment statement appears in a processing symbol, which is a rectangle. **Figure 2-11** shows a flowchart for **Program 2-6**.

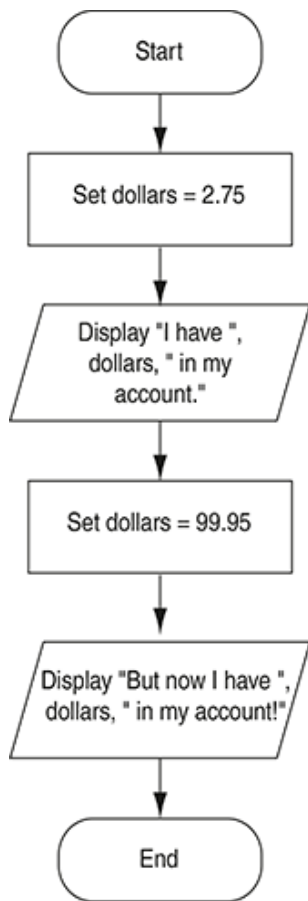


Figure 2-11 Flowchart for [Program 2-6](#)

Performing Calculations

Most real-world algorithms require calculations to be performed. A programmer's tools for performing calculations are *math operators*. Programming languages commonly provide the operators shown in [Table 2-1](#).

Table 2-1 Common math operators¹

Symbol	Operator	Description
+	Addition	Adds two numbers
–	Subtraction	Subtracts one number from another
*	Multiplication	Multiplies one number by another
/	Division	Divides one number by another and gives the quotient
MOD	Modulus	Divides one number by another and gives the remainder
^	Exponent	Raises a number to a power

¹ In some programming languages, the % character is used as the modulus operator, and sometimes the ** characters are used as the exponent operator.



Performing Calculations

Programmers use the operators shown in [Table 2-1](#) to create math expressions. A *math expression* performs a calculation and gives a value. The following is an example of a simple math expression:

```
12 + 2
```

The values on the right and left of the `+` operator are called *operands*. These are values that the `+` operator adds together. The value that is given by this expression is 14.

Variables may also be used in a math expression. For example, suppose we have two variables named `hours` and `payRate`. The following math expression uses the `*` operator to multiply the value in the `hours` variable by the value in the `payRate` variable:

```
hours * payRate
```

When we use a math expression to calculate a value, normally we want to save that value in memory so we can use it again in the program. We do this with an assignment statement. [Program 2-7](#) shows an example.

Program 2-7

```
1 Set price = 100
2 Set discount = 20
3 Set sale = price - discount
4 Display "The total cost is $", sale
```

Program Output

```
The total cost is $80
```

Line 1 sets the `price` variable to 100, and line 2 sets the `discount` variable to 20. Line 3 sets the `sale` variable to the result of the expression `price - discount`. As you can see from the program output, the `sale` variable holds the value 80.

In the Spotlight:



Calculating Cell Phone Overage Fees

Suppose your cell phone calling plan allows you to use 700 minutes per month. If you use more than this limit in a month, you are charged an overage fee of 35 cents for each excess minute. Your phone

shows you the number of excess minutes that you have used in the current month, but it does not show you how much your overage fee currently is. Until now, you've been doing the math the old-fashioned way (with pencil and paper, or with a calculator), but you would like to design a program that will simplify the task. You would like to be able to enter the number of excess minutes, and have the program perform the calculation for you.

First, you want to make sure that you understand the steps that the program must perform. It will be helpful if you closely look at the way you've been solving this problem, using only paper and pencil, or calculator:

Manual Algorithm (Using pencil and paper, or calculator)

1. You get the number of excess minutes that you have used.
2. You multiply the number of excess minutes by 0.35.
3. The result of the calculation is your current overage fee.

Ask yourself the following questions about this algorithm:

<i>Question:</i>	What input do I need to perform this algorithm?
<i>Answer:</i>	I need the number of excess minutes.
<i>Question:</i>	What must I do with the input?
<i>Answer:</i>	I must multiply the input (the number of excess minutes) by 0.35. The result of that calculation is the overage fee.
<i>Question:</i>	What output must I produce?
<i>Answer:</i>	The overage fee.

Now that you have identified the input, the process that must be performed, and the output, you can write the general steps of the program's algorithm:

Computer Algorithm

1. Get the number of excess minutes as input.
2. Calculate the overage fee by multiplying the number of excess minutes by 0.35.
3. Display the overage fee.

In Step 1 of the computer algorithm, the program gets the number of excess minutes from the user. Any time a program needs the user to enter a piece of data, it does two things: (1) it displays a message prompting the user for the piece of data, and (2) it reads the data that the user enters on the keyboard, and stores that data in a variable. In pseudocode, Step 1 of the algorithm will look like this:

```
Display "Enter the number of excess minutes."  
Input excessMinutes
```

Notice that the *Input* statement stores the value entered by the user in a variable named *excessMinutes*.

In Step 2 of the computer algorithm, the program calculates the overage fee by multiplying the number

of excess minutes by 0.35. The following pseudocode statement performs this calculation, and stores the result in a variable named *overageFee*:

```
Set overageFee = excessMinutes * 0.35
```

In Step 3 of the computer algorithm, the program displays the overage fee. Because the overage fee is stored in the *overageFee* variable, the program will display a message that shows the value of the *overageFee* variable. In pseudocode we will use the following statement:

```
Display "Your current overage fee is $", overageFee
```

Program 2-8 shows the entire pseudocode program, with example output. **Figure 2-12** shows the flowchart for this program.

Program 2-8

```
1 Display "Enter the number of excess minutes."  
2 Input excessMinutes  
3 Set overageFee = excessMinutes * 0.35  
4 Display "Your current overage fee is $", overageFee
```

Program Output (with Input Shown in Bold)

```
Enter the number of excess minutes.  
100 [Enter]  
Your current overage fee is $35
```

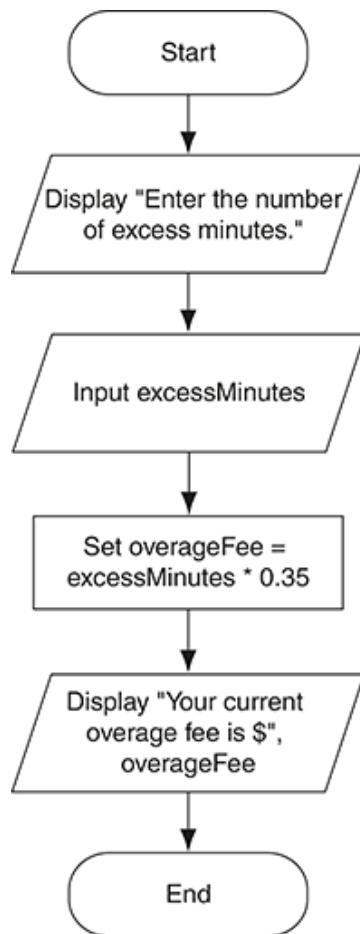


Figure 2-12 Flowchart for Program 2-8

In the Spotlight:



Calculating a Percentage

Determining percentages is a common calculation in computer programming. In mathematics, the % symbol is used to indicate a percentage, but most programming languages don't use the % symbol for this purpose. In a program, you usually have to convert a percentage to a decimal number. For example, 50 percent would be written as 0.5 and 2 percent would be written as 0.02.

Let's step through the process of writing a program that calculates a percentage. Suppose a retail business is planning to have a storewide sale where the prices of all items will be 20 percent off. We have been asked to write a program to calculate the sale price of an item after the discount is subtracted. Here is the algorithm:

1. Get the original price of the item.
2. Calculate 20 percent of the original price. This is the amount of the discount.
3. Subtract the discount from the original price. This is the sale price.
4. Display the sale price.

In Step 1 we get the original price of the item. We will prompt the user to enter this data on the keyboard. Recall from the previous section that prompting the user is a two-step process: (1) display a message telling the user to enter the desired data, and (2) reading that data from the keyboard. We will use the following pseudocode statements to do this. Notice that the value entered by the user will be

stored in a variable named `originalPrice`.

```
Display "Enter the item's original price."  
Input originalPrice
```

In Step 2, we calculate the amount of the discount. To do this we multiply the original price by 20 percent. The following statement performs this calculation and stores the result in the discount variable.

```
Set discount = originalPrice * 0.2
```

In Step 3, we subtract the discount from the original price. The following statement does this calculation and stores the result in the `salePrice` variable.

```
Set salePrice = originalPrice - discount
```

Last, in Step 4, we will use the following statement to display the sale price:

```
Display "The sale price is $", salePrice
```

Program 2-9 shows the entire pseudocode program, with example output. **Figure 2-13** shows the flowchart for this program.

Program 2-9

```
1 Display "Enter the item's original price."  
2 Input originalPrice  
3 Set discount = originalPrice * 0.2  
4 Set salePrice = originalPrice - discount  
5 Display "The sale price is $", salePrice
```

Program Output (with Input Shown in Bold)

```
Enter the item's original price.  
100 [Enter]  
The sale price is $80
```

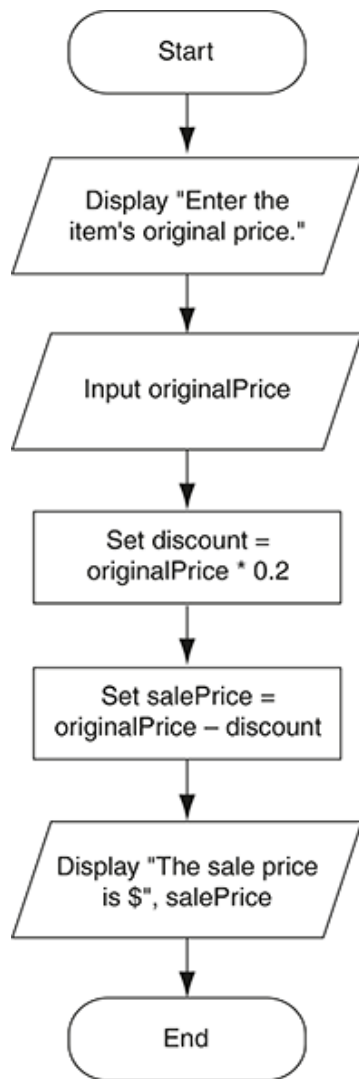


Figure 2-13 Flowchart for Program 2-9

The Order of Operations

It is possible to build mathematical expressions with several operators. The following statement assigns the sum of 17, the variable `x`, 21, and the variable `y` to the variable `answer`.

```
Set answer = 17 + x + 21 + y
```

Some expressions are not that straightforward, however. Consider the following statement:

```
Set outcome = 12 + 6 / 3
```

What value will be stored in `outcome`? The number 6 is used as an operand for both the addition and division operators. The `outcome` variable could be assigned either 6 or 14, depending on when the division takes place. The answer is 14 because the *order of operations* dictates that the division operator works before the addition operator does.

In most programming languages, the order of operations can be summarized as follows:

1. Perform any operations that are enclosed in parentheses.
2. Perform any operations that use the exponent operator to raise a number to a power.
3. Perform any multiplications, divisions, or modulus operations as they appear from left to right.
4. Perform any additions or subtractions as they appear from left to right.

Mathematical expressions are evaluated from left to right. When two operators share an operand, the order of operations determines which operator works first. Multiplication and division are always performed before addition and subtraction, so the statement

```
Set outcome = 12 + 6 / 3
```

works like this:

1. 6 is divided by 3, yielding a result of 2
2. 12 is added to 2, yielding a result of 14

It could be diagrammed as shown in [Figure 2-14](#).

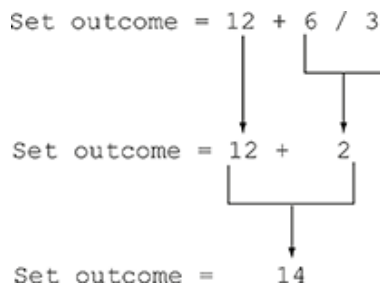


Figure 2-14 The order of operations at work

[Table 2-2](#) shows some other sample expressions with their values.

Table 2-2 Some expressions and their values

Expression	Value
$5 + 2 * 4$	13
$10 / 2 - 3$	2
$8 + 12 * 2 - 4$	28
$6 - 3 * 2 + 7 - 1$	6

Grouping with Parentheses

Parts of a mathematical expression may be grouped with parentheses to force some operations to be performed before others. In the following statement, the variables *a* and *b* are added together, and their sum is divided by 4:

```
Set result = (a + b) / 4
```

Without the parentheses, however, *b* would be divided by 4 and the result added to *a*. **Table 2-3** shows more expressions and their values.

Table 2-3 More expressions and their values

Expression	Value
$(5 + 2) * 4$	28
$10 / (5 - 3)$	5
$8 + 12 * (6 - 2)$	56
$(6 - 3) * (2 + 7) / 3$	9



NOTE:

Parentheses can be used to enhance the clarity of a math expression, even when they are unnecessary to get the correct result. For example, look at the following statement:

```
Set fahrenheit = celsius * 1.8 + 32
```

Even when it is unnecessary to get the correct result, we can insert parentheses to clearly show that *celsius * 1.8* happens first in the math expression:

```
Set fahrenheit = (celsius * 1.8) + 32
```



In the Spotlight:

Calculating an Average

Determining the average of a group of values is a simple calculation: You add all of the values and then divide the sum by the number of values. Although this is a straightforward calculation, it is easy to make a mistake when writing a program that calculates an average. For example, let's assume that the variables *a*, *b*, and *c* each hold a value and we want to calculate the average of those values. If we are careless, we might write a statement such as the following to perform the calculation:

```
Set average = a + b + c / 3
```

Can you see the error in this statement? When it executes, the division will take place first. The value in *c* will be divided by 3, and then the result will be added to *a + b*. That is not the correct way to calculate an average. To correct this error we need to put parentheses around *a + b + c*, as shown

here:

```
Set average = (a + b + c) / 3
```

Let's step through the process of writing a program that calculates an average. Suppose you have taken three tests in your computer science class, and you want to write a program that will display the average of the test scores. Here is the algorithm:

1. Get the first test score.
2. Get the second test score.
3. Get the third test score.
4. Calculate the average by adding the three test scores and dividing the sum by 3.
5. Display the average.

In Steps 1, 2, and 3 we will prompt the user to enter the three test scores. We will store those test scores in the variables `test1`, `test2`, and `test3`. In Step 4 we will calculate the average of the three test scores. We will use the following statement to perform the calculation and store the result in the `average` variable:

```
Set average = (test1 + test2 + test3) / 3
```

Last, in Step 5, we display the average. **Program 2-10** shows the pseudocode for this program, and **Figure 2-15** shows the flowchart.

Program 2-10

```
1  Display "Enter the first test score."
2  Input test1
3  Display "Enter the second test score."
4  Input test2
5  Display "Enter the third test score."
6  Input test3
7  Set average = (test1 + test2 + test3) / 3
8  Display "The average score is ", average
```

Program Output (with Input Shown in Bold)

```
Enter the first test score.
90 [Enter]
Enter the second test score.
80 [Enter]
Enter the third test score.
100 [Enter]
The average score is 90
```

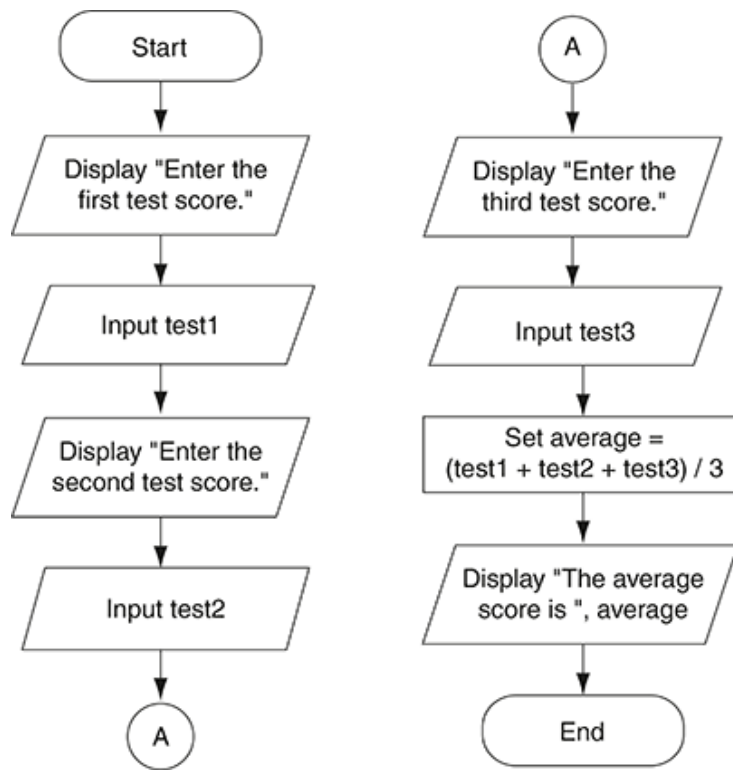



Figure 2-15 Flowchart for Program 2-10

Advanced Arithmetic Operators: Exponent and Modulus

In addition to the basic math operators for addition, subtraction, multiplication, and division, many languages provide an exponent operator and a modulus operator. The `^` symbol is commonly used as the exponent operator, and its purpose is to raise a number to a power. For example, the following pseudocode statement raises the `length` variable to the power of 2 and stores the result in the `area` variable:

```
Set area = length^2
```

The word `MOD` is used in many languages as the modulus operator. (Some languages use the `%` symbol for the same purpose.) The modulus operator performs division, but instead of returning the quotient, it returns the remainder. The following statement assigns 2 to `leftover`:

```
Set leftover = 17 MOD 3
```

This statement assigns 2 to `leftover` because 17 divided by 3 is 5 with a remainder of 2. You will not use the modulus operator frequently, but it is useful in some situations. It is commonly used in calculations that detect odd or even numbers, determine the day of the week, measure the passage of time, and other specialized operations.

Converting Math Formulas to Programming Statements

You probably remember from algebra class that the expression $2xy$ is understood to mean 2 times x times y . In math, you do not always use an operator for multiplication. Programming languages, however, require an operator for any mathematical operation. Table 2-4 shows some algebraic expressions that perform

multiplication and the equivalent programming expressions.

Table 2-4 Algebraic expressions

Algebraic Expression	Operation Being Performed	Programming Expression
$6B$	6 times B	$6 * B$
$(3)(12)$	3 times 12	$3 * 12$
$4xy$	4 times x times y	$4 * x * y$

When converting some algebraic expressions to programming expressions, you may have to insert parentheses that do not appear in the algebraic expression. For example, look at the following formula:

$$x = \frac{a + b}{c}$$

To convert this to a programming statement, $a + b$ will have to be enclosed in parentheses:

```
Set x = (a + b) / c
```

Table 2-5 shows additional algebraic expressions and their pseudocode equivalents.

Table 2-5 Algebraic expressions and pseudocode statements

Algebraic Expression	Pseudocode Statement
$y = \frac{x}{2 * 3}$	Set y = x / 2 * 3
$z = 3bc + 4$	Set z = 3 * b * c + 4
$a = \frac{x + 2}{a - 1}$	Set a = (x + 2) / (a - 1)

In the Spotlight:



Converting a Math Formula to a Programming Statement

Suppose you want to deposit a certain amount of money into a savings account, and then leave it alone to draw interest for the next 10 years. At the end of 10 years you would like to have \$10,000 in the account. How much do you need to deposit today to make that happen? You can use the following formula to find out:

$$P = \frac{F}{(1 + r)^n}$$

The terms in the formula are as follows:

P is the present value, or the amount that you need to deposit today.

- F is the future value that you want in the account. (In this case, F is \$10,000.)
- r is the annual interest rate.
- n is the number of years that you plan to let the money sit in the account.

It would be nice to write a computer program to perform the calculation, because then we can experiment with different values for the terms. Here is an algorithm that we can use:

1. Get the desired future value.
2. Get the annual interest rate.
3. Get the number of years that the money will sit in the account.
4. Calculate the amount that will have to be deposited.
5. Display the result of the calculation in Step 4.

In Steps 1 through 3, we will prompt the user to enter the specified values. We will store the desired future value in a variable named `futureValue`, the annual interest rate in a variable named `rate`, and the number of years in a variable named `years`.

In Step 4, we calculate the present value, which is the amount of money that we will have to deposit. We will convert the formula previously shown to the following pseudocode statement. The statement stores the result of the calculation in the `presentValue` variable.

```
Set presentValue = futureValue / (1 + rate)^years
```

In Step 5, we display the value in the `presentValue` variable. **Program 2-11** shows the pseudocode for this program, and **Figure 2-16** shows the flowchart.

Program 2-11

```
1  Display "Enter the desired future value."
2  Input futureValue
3  Display "Enter the annual interest rate."
4  Input rate
5  Display "How many years will you let the money grow?"
6  Input years
7  Set presentValue = futureValue / (1 + rate)^years
8  Display "You will need to deposit $", presentValue
```

Program Output (with Input Shown in Bold)

```
Enter the desired future value.
10000 [Enter]
Enter the annual interest rate.
0.05 [Enter]
How many years will you let the money grow?
10 [Enter]
You will need to deposit $6139
```

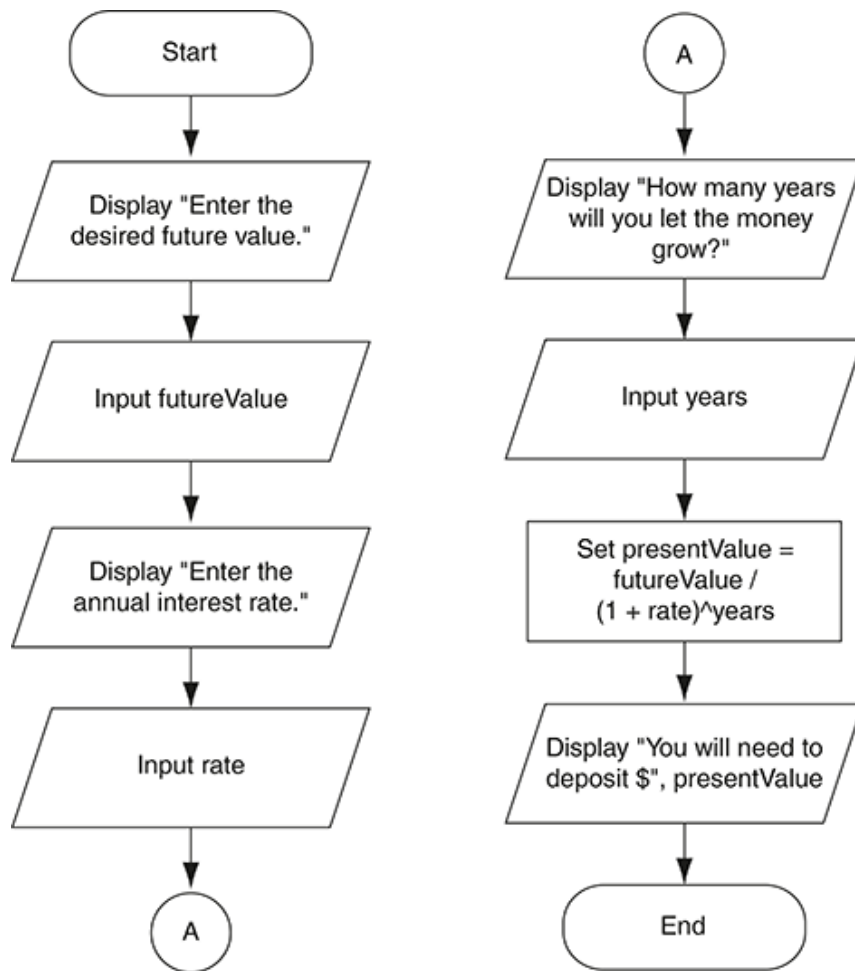


Figure 2-16 Flowchart for Program 2-11



Checkpoint

- 2.20 What is an assignment statement?
- 2.21 When you assign a value to a variable, what happens to any value that is already stored in the variable?
- 2.22 Summarize the mathematical order of operations, as it works in most programming languages.
- 2.23 What is the purpose of the exponent operator?
- 2.24 What is the purpose of the modulus operator?

2.4 Variable Declarations and Data Types

CONCEPT: Most languages require that variables be declared before they are used in a program. When a variable is declared, it can optionally be initialized with a value. Using an uninitialized variable is the source of many errors in programming.



Variable Declarations

Most programming languages require that you *declare* all of the variables that you intend to use in a program. A *variable declaration* is a statement that typically specifies two things about a variable:

- The variable's name
- The variable's data type

A variable's *data type* is simply the type of data that the variable will hold. Once you declare a variable, it can be used to store values of only the specified data type. In most languages, an error occurs if you try to store values of other types in the variable.

The data types that you are allowed to use depend on the programming language. For example, the Java language provides four data types for integer numbers, two data types for real numbers, one data type for strings, and others.

So far, we haven't declared any of the variables that we have used in our example pseudocode programs. We have simply used the variables without first declaring them. This is permissible with short pseudocode programs, but as programs grow in length and complexity, it makes sense to declare them. When you declare variables in a pseudocode program, it will make the job of translating the pseudocode to actual code easier.

In most of the programs in this book, we will use only three data types when we declare variables: *Integer*, *Real*, and *String*. Here is a summary of each:

- A variable of the *Integer* data type can hold whole numbers. For example, an *Integer* variable can hold values such as 42, 0, and -99. An *Integer* variable cannot hold numbers with a fractional part, such as 22.1 or -4.9
- A variable of the *Real* data type can hold either whole numbers or numbers with a fractional part. For example, a *Real* variable can hold values such as 3.5, -87.95, and 3.0.
- A variable of the *String* data type can hold any string of characters, such as someone's name, address, password, and so on.

In this book, we will begin variable declarations with the word *Declare*, followed by a data type, followed by the variable's name. Here is an example:

```
Declare Integer length
```

This statement declares a variable named *length*, of the *Integer* data type. Here is another example:

```
Declare Real grossPay
```

This statement declares a variable named *grossPay*, of the *Real* data type. Here is one more example:

```
Declare String name
```

This statement declares a variable named *name*, of the *String* data type.

If we need to declare more than one variable of the same data type, we can use one declaration statement. For example, suppose we want to declare three variables, *length*, *width*, and *height*, all of the *Integer* data type. We can declare all three with one statement, as shown here:

```
Declare Integer length, width, height
```



NOTE:

In addition to a *String* data type, many programming languages also provide a *Character* data type. The difference between a *String* variable and a *Character* variable is that a *String* variable can hold a sequence of characters of virtually any length, and a *Character* variable can hold only one character. In this book, we will keep things simple. We will use *String* variables to hold all character data.

Declaring Variables Before Using Them

The purpose of a variable declaration statement is to tell the compiler or interpreter that you plan to use a particular variable in the program. A variable declaration statement typically causes the variable to be created in memory. For this reason, you have to write a variable's declaration statement *before* any other statements in the program that use the variable. This makes perfect sense because you cannot store a value in a variable if the variable has not been created in memory.

For example, look at the following pseudocode. If this code were converted to actual code in a language like Java or C++, it would cause an error because the *Input* statement uses the *age* variable before the variable has been declared.

```
Display "What is your age?"  
Input age  
Declare Integer age
```

s This pseudocode has an error!

Program 2-12 shows the correct way to declare a variable. Notice that the declaration statement for the *age* variable appears before any other statements that use the variable.

Program 2-12

```
1 Declare Integer age  
2 Display "What is your age?"  
3 Input age  
4 Display "Here is the value that you entered:"  
5 Display age
```

Program Output (with Input Shown in Bold)

```
What is your age?  
24 [Enter]  
Here is the value that you entered:  
24
```

Program 2-13 shows another example. This program declares a total of four variables: three to hold test scores and another to hold the average of those test scores.

Program 2-13

```
1  Declare Real test1  
2  Declare Real test2  
3  Declare Real test3  
4  Declare Real average  
5  
6  Set test1 = 88.0  
7  Set test2 = 92.5  
8  Set test3 = 97.0  
9  Set average = (test1 + test2 + test3) / 3  
10 Display "Your average test score is ", average
```

Program Output

```
Your average test score is 92.5
```

This program shows a common technique for declaring variables: they are all declared at the beginning of the program, before any other statements. This is one way of making sure that all variables are declared before they are used.

Notice that line 5 in this program is blank. This blank line does *not* affect the way the program works because most compilers and interpreters ignore blank lines. For the human reader, however, this blank line visually separates the variable declarations from the other statements. This makes the program appear more organized and easier for people to read.

Programmers commonly use blank lines and indentations in their code to create a sense of organization visually. This is similar to the way that authors visually arrange the text on the pages of a book. Instead of writing each chapter as one long series of sentences, they break it into paragraphs. This does not change the information in the book; it only makes it easier to read.

Although you are generally free to place blank lines and indentations anywhere in your code, you should not do this haphazardly. Programmers follow certain conventions when it comes to this. For example, you have just learned that one convention is to use a blank line to separate a group of variable declaration statements from the rest of the statements in a program. These conventions are known as *programming style*. As you progress through this book you will see many other programming style conventions.

Variable Initialization

When you declare a variable, you can optionally assign a value to it in the declaration statement. This is known as *initialization*. For example, the following statement declares a variable named *price* and assigns the value 49.95 to it:


```
Declare Real price = 49.95
```

We would say that this statement *initializes* the *price* variable with the value 49.95. The following statement shows another example:

```
Declare Integer length = 2, width = 4, height = 8
```

This statement declares and initializes three variables. The *length* variable is initialized with the value 2, *width* is initialized with the value 4, and *height* is initialized with the value 8.

Uninitialized Variables

An *uninitialized variable* is a variable that has been declared, but has not been initialized or assigned a value. Uninitialized variables are a common cause of logic errors in programs. For example, look at the following pseudocode:

```
Declare Real dollars  
Display "I have ", dollars, " in my account."
```

In this pseudocode, we have declared the *dollars* variable, but we have not initialized it or assigned a value to it. Therefore, we do not know what value the variable holds. Nevertheless, we have used the variable in a *Display* statement.

You're probably wondering what a program like this would display. An honest answer would be "I don't know." This is because each language has its own way of handling uninitialized variables. Some languages assign a default value such as 0 to uninitialized variables. In many languages, however, uninitialized variables hold unpredictable values. This is because those languages set aside a place in memory for the variable, but do not alter the contents of that place in memory. As a result, an uninitialized variable holds the value that happens to be stored in its memory location. Programmers typically refer to unpredictable values such as this as "garbage."

Uninitialized variables can cause logic errors that are hard to find in a program. This is especially true when an uninitialized variable is used in a calculation. For example, look at the following pseudocode, which is a modified version of [Program 2-13](#). Can you spot the error?

```
1 Declare Real test1  
2 Declare Real test2  
3 Declare Real test3  
4 Declare Real average  
5  
6 Set test1 = 88.0  
7 Set test2 = 92.5  
8 Set average = (test1 + test2 + test3) / 3  
9 Display "Your average test score is ", average
```

This pseudocode
contains an error!

This program will not work properly because the *test3* variable is never assigned a value. The *test3* variable will contain garbage when it is used in the calculation in line 8. This means that the calculation will result in an unpredictable value, which will be assigned to the *average* variable. A beginning programmer

might have trouble finding this error because he or she would initially assume that something is wrong with the math in line 8.

In the next section, we will discuss a debugging technique that will help uncover errors such as the one in the modified version of **Program 2-13**. However, as a rule you should always make sure that your variables either (1) are initialized with the correct value when you declare them, or (2) receive the correct value from an assignment statement or an *Input* statement before they are used for any other purpose.

Numeric Literals and Data Type Compatibility

Many of the programs that you have seen so far have numbers written into their pseudocode. For example, the following statement, which appears in **Program 2-6**, has the number 2.75 written into it.

```
Set dollars = 2.75
```

And, the following statement, which appears in **Program 2-7**, has the number 100 written into it.

```
Set price = 100
```

A number that is written into a program's code is called a *numeric literal*. In most programming languages, if a numeric literal is written with a decimal point, such as 2.75, that numeric literal will be stored in the computer's memory as a *Real* and it will be treated as a *Real* when the program runs. If a numeric literal does not have a decimal point, such as 100, that numeric literal will be stored in the computer's memory as an *Integer* and it will be treated as an *Integer* when the program runs.

This is important to know when you are writing assignment statements or initializing variables. In many languages, an error will occur if you try to store a value of one data type in a variable of another data type. For example, look at the following pseudocode:

Declare Integer i

Set i = 3.7 ← This is an error!

The assignment statement will cause an error because it attempts to assign a real number, 3.7, in an *Integer* variable. The following pseudocode will also cause an error.

Declare Integer i

Set i = 3.0 ← This is an error!



NOTE:

Most languages do not allow you to assign real numbers to *Integer* variables because *Integer* variables cannot hold fractional amounts. In many languages, however, you are allowed to assign an integer value to a *Real* variable without causing an error. Here is an example:

```
Declare Real r  
Set r = 77
```

Even though the numeric literal 77 is treated as an *Integer*, it can be assigned to a *Real* variable without the loss of data.

Even though the numeric literal 3.0 does not have a fractional value (it is mathematically the same as the integer 3), it is still treated as a real number by the computer because it is written with a decimal point.

Integer Division

Be careful when dividing an integer by another integer. In many programming languages, when an integer is divided by an integer the result will also be an integer. This behavior is known as *integer division*. For example, look at the following pseudocode:

```
Set number = 3 / 2
```

This statement divides 3 by 2 and stores the result in the *number* variable. What will be stored in *number*? You would probably assume that 1.5 would be stored in *number* because that's the result your calculator shows when you divide 3 by 2. However, that's not what will happen in many programming languages. Because the numbers 3 and 2 are both treated as integers, the programming language that you are using might throw away the fractional part of the answer. (Throwing away the fractional part of a number is called *truncation*.) As a result, the statement will store 1 in the *number* variable, not 1.5.

If you are using a language that behaves this way and you want to make sure that a division operation yields a real number, at least one of the operands must be a real number or a *Real* variable.



NOTE:

In Java, C++, and C, the / operator throws away the fractional part of the result when both operands are integers. In these languages the result of the expression $3/2$ would be 1. In Visual Basic, the / operator does not throw away the fractional part of the answer. In Visual Basic, the result of the expression $3/2$ would be 1.5.



Checkpoint

- 2.25 What two items do you usually specify with a variable declaration?
- 2.26 Does it matter where you write the variable declarations in a program?
- 2.27 What is variable initialization?
- 2.28 Do uninitialized variables pose any danger in a program?
- 2.29 What is an uninitialized variable?

2.5 Named Constants

CONCEPT: A named constant is a name that represents a value that cannot be changed during the program's execution.

Assume that the following statement appears in a banking program that calculates data pertaining to loans:

```
Set amount = balance * 0.069
```

In such a program, two potential problems arise. First, it is not clear to anyone other than the original programmer what 0.069 is. It appears to be an interest rate, but in some situations there are fees associated with loan payments. How can the purpose of this statement be determined without painstakingly checking the rest of the program?

The second problem occurs if this number is used in other calculations throughout the program and must be changed periodically. Assuming the number is an interest rate, what if the rate changes from 6.9 percent to 7.2 percent? The programmer would have to search through the source code for every occurrence of the number. (It is not advisable to use an editor's global search and replace feature to change values in a program. You might change an unintended value in the program, and inadvertently cause a logic error.)

Both of these problems can be addressed by using named constants. A *named constant* is a name that represents a value that cannot be changed during the program's execution. The following is an example of how we will declare named constants in our pseudocode:

```
Constant Real INTEREST_RATE = 0.069
```

This creates a constant named `INTEREST_RATE`. The constant's value is the *Real* number 0.069. Notice that the declaration looks a lot like a variable declaration, except that we use the word *Constant* instead of *Declare*. Also, notice that the name of the constant is written in all uppercase letters. This is a standard practice in most programming languages because it makes named constants easily distinguishable from regular variable names. An initialization value must be given when declaring a named constant.

An advantage of using named constants is that they make programs more self-explanatory. The following statement:

```
Set amount = balance * 0.069
```

can be changed to read

```
Set amount = balance * INTEREST_RATE
```

A new programmer can read the second statement and know what is happening. It is evident that

balance is being multiplied by the interest rate. Another advantage to this approach is that widespread changes can easily be made to the program. Let's say the interest rate appears in a dozen different statements throughout the program. When the rate changes, the initialization value in the declaration of the named constant is the only value that needs to be modified. If the rate increases to 7.2 percent, the declaration can be changed to the following:

```
Constant Real INTEREST_RATE = 0.072
```

The new value of 0.072 will then be used in each statement that uses the *INTEREST_RATE* constant.



NOTE:

A named constant cannot be assigned a value with a *Set* statement. If a statement in a program attempts to change the value of a named constant, an error will occur.

2.6 Hand Tracing a Program

CONCEPT: Hand tracing is a simple debugging process for locating hard-to-find errors in a program.

Hand tracing is a debugging process where you imagine that you are the computer executing a program. (This process is also known as *desk checking*.) You step through each of the program's statements one by one. As you carefully look at a statement, you record the contents that each variable will have after the statement executes. This process is often helpful in finding mathematical mistakes and other logic errors.

To hand trace a program, you construct a chart that has a column for each variable, and a row for each line in the program. For example, [Figure 2-17](#) shows how we would construct a hand trace chart for the program that you saw in the previous section. The chart has a column for each of the four variables: *test1*, *test2*, *test3*, and *average*. The chart also has nine rows, one for each line in the program.

	test1	test2	test3	average
1				
2				
3				
4				
5				
6				
7				
8				
9				

Figure 2-17 A program with a hand trace chart

To hand trace this program, you step through each statement, observing the operation that is taking place, and then record the value that each variable will hold *after* the statement executes. When the process is complete, the chart will appear as shown in [Figure 2-18](#). We have written question marks in the chart to indicate that a variable is uninitialized.

	test1	test2	test3	average
1	?	?	?	?
2	?	?	?	?
3	?	?	?	?
4	?	?	?	?
5	?	?	?	?
6	88	?	?	?
7	88	92.5	?	?
8	88	92.5	?	undefined
9	88	92.5	?	undefined

Figure 2-18 Program with the hand trace chart completed

When we get to line 8 we will carefully do the math. This means we look at the values of each variable in the expression. At that point we discover that one of the variables, *test3*, is uninitialized. Because it is uninitialized, we have no way of knowing the value that it contains. Consequently, the result of the calculation will be undefined. After making this discovery, we can correct the problem by adding a line that assigns a value to *test3*.

Hand tracing is a simple process that focuses your attention on each statement in a program. Often this helps you locate errors that are not obvious.

2.7 Documenting a Program

CONCEPT: A program's external documentation describes aspects of the program for the user. The internal documentation is for the programmer, and explains how parts of the program work.

A program's documentation explains various things about the program. There are usually two types of program documentation: external and internal. *External documentation* is typically designed for the user. It consists of documents such as a reference guide that describes the program's features, and tutorials that teach the user how to operate the program.

Sometimes the programmer is responsible for writing all or part of a program's external documentation. This might be the case in a small organization, or in a company that has a relatively small programming staff. Some organizations, particularly large companies, will employ a staff of technical writers whose job is to produce external documentation. These documents might be in printed manuals, or in files that can be viewed on the computer. In recent years it has become common for software companies to provide all of a program's external documentation in PDF (Portable Document Format) files.

Internal documentation appears as *comments* in a program's code. Comments are short notes placed in different parts of a program, explaining how those parts of the program work. Although comments are a critical part of a program, they are ignored by the compiler or interpreter. Comments are intended for human readers of a program's code, not the computer.

Programming languages provide special symbols or words for writing comments. In several languages, including Java, C, and C++, you begin a comment with two forward slashes (//). Everything that you write on the same line, after the slashes, is ignored by the compiler. Here is an example of a comment in any of those languages:

```
// Get the number of hours worked.
```

Some languages use symbols other than the two forward slashes to indicate the beginning of a comment. For example, Visual Basic uses an apostrophe ('), and Python uses the # symbol. In this book, we will use two forward slashes (//) in pseudocode.

Block Comments and Line Comments

Programmers generally write two types of comments in a program: block comments and line comments. *Block comments* take up several lines and are used when lengthy explanations are required. For example, a block comment often appears at the beginning of a program, explaining what the program does, listing the name of the author, giving the date that the program was last modified, and any other necessary information. The following is an example of a block comment:

```
// This program calculates an employee's gross pay.  
// Written by Matt Hoyle.  
// Last modified on 12/14/2018
```


**NOTE:**

Some programming languages provide special symbols to mark the beginning and ending of a block comment.

Line comments are comments that occupy a single line, and explain a short section of the program. The following statements show an example:

```
// Calculate the interest.  
Set interest = balance * interest_Rate  
// Add the interest to the balance.  
Set balance = balance + interest
```

A line comment does not have to occupy an entire line. Anything appearing after the `//` symbol, to the end of the line, is ignored, so a comment can appear after an executable statement. Here is an example:

```
Input age      // Get the user's age.
```

As a beginning programmer, you might be resistant to the idea of liberally writing comments in your programs. After all, it's a lot more fun to write code that actually does something! It is crucial that you take the extra time to write comments, however. They will almost certainly save you time in the future when you have to modify or debug the program. Even large and complex programs can be made easy to read and understand if they are properly commented.

In the Spotlight:

**Using Named Constants, Style Conventions, and Comments**

Suppose we have been given the following programming problem: Scientists have determined that the world's ocean levels are currently rising at about 1.5 millimeters per year. Write a program to display the following:

- The number of millimeters that the oceans will rise in five years
- The number of millimeters that the oceans will rise in seven years
- The number of millimeters that the oceans will rise in ten years

Here is the algorithm:

1. Calculate the amount that the oceans will rise in five years.
2. Display the result of the calculation in Step 1.
3. Calculate the amount that the oceans will rise in seven years.
4. Display the result of the calculation in Step 3.
5. Calculate the amount that the oceans will rise in ten years.

6. Display the result of the calculation in Step 5.

This program is straightforward. It performs three calculations and displays the results of each. The calculations should give the amount the oceans will rise in five, seven, and ten years. Each of these values can be calculated with the following formula:

$$\text{Amount of yearly rise} \times \text{Number of years}$$

The amount of yearly rise is the same for each calculation, so we will create a constant to represent that value. **Program 2-14** shows the pseudocode for the program.

Program 2-14

```
1  // Declare the variables
2  Declare Real fiveYears
3  Declare Real sevenYears
4  Declare Real tenYears
5
6  // Create a constant for the yearly rise
7  Constant Real YEARLY_RISE = 1.5
8
9  // Display the amount of rise in five years
10 Set fiveYears = YEARLY_RISE * 5
11 Display "The ocean levels will rise ", fiveYears,
12         " millimeters in five years."
13
14 // Display the amount of rise in seven years
15 Set sevenYears = YEARLY_RISE * 7
16 Display "The ocean levels will rise ", sevenYears,
17         " millimeters in seven years."
18
19 // Display the amount of rise in ten years
20 Set tenYears = YEARLY_RISE * 10
21 Display "The ocean levels will rise ", tenYears,
22         " millimeters in ten years."
```

Program Output

```
The ocean levels will rise 7.5 millimeters in five years.
The ocean levels will rise 10.5 millimeters in seven years.
The ocean levels will rise 15 millimeters in ten years.
```

Three variables, *fiveYears*, *sevenYears*, and *tenYears*, are declared in lines 2 through 4. These variables will hold the amount that the ocean levels will rise in five, seven, and ten years.

Line 7 creates a constant, *YEARLY_RISE*, which is set to the value 1.5. This is the amount that the oceans rise per year. This constant will be used in each of the program's calculations.

Lines 10 through 12 calculate and display the amount that the oceans will rise in five years. The same values for seven years and ten years are calculated and displayed in lines 15 through 17

and 20 through 22.

This program illustrates the following programming style conventions:

- Several blank lines appear throughout the program (see lines 5, 8, 13, and 18). These blank lines do not affect the way the program works, but make the pseudocode easier to read.
- Line comments are used in various places to explain what the program is doing.
- Notice that each of the *Display* statements is too long to fit on one line. (See lines 11 and 12, 16 and 17, 21 and 22.) Most programming languages allow you to write long statements across several lines. When we do this in pseudocode, we will indent the second and subsequent lines. This will give a visual indication that the statement spans more than one line.

Figure 2-19 shows a flowchart for the program.

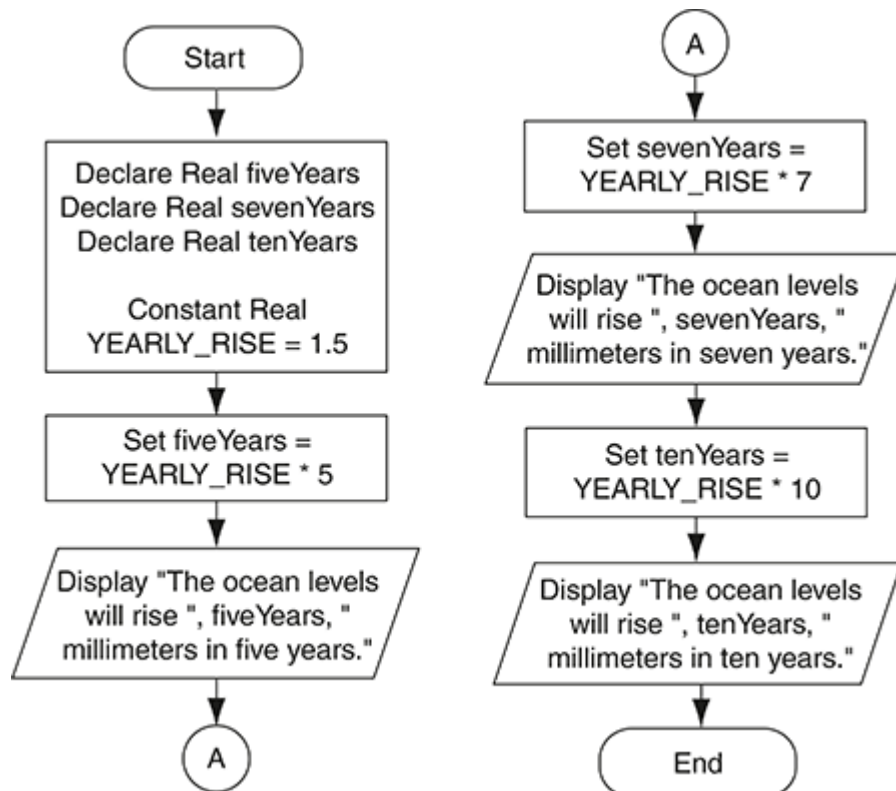


Figure 2-19 Flowchart for **Program 2-14**



Checkpoint

2.30 What is external documentation?

2.31 What is internal documentation?

2.32 What are the two general types of comments that programmers write in a program's code? Describe each.

2.8 Designing Your First Program

Sometimes, as a beginning student, you might have trouble getting started with a programming problem. In this section we will present a simple problem, go through the process of analyzing the program's requirements, and design the algorithm in pseudocode and a flowchart. Here is the programming problem:

Batting Average

In baseball, batting average is commonly used to measure a player's batting ability. You use the following formula to calculate a player's batting average:

$$\text{Batting Average} = \text{Hits} \div \text{Times at Bat}$$

In the formula, *Hits* is the number of successful hits made by the player, and *Times at Bat* is the number of times the player was at bat. For example, if a player is at bat 500 times during a season, and gets 150 hits, that player's batting average is .300. Design a program to calculate any player's batting average.

Recall from [Section 2.2](#) that a program's actions can typically be divided into the following three phases:

1. Input is received.
2. Some process (such as a calculation) is performed on the input.
3. Output is produced.

Your first step is to determine what is required in each phase. Usually these requirements are not stated directly in the problem description. For example, the previously shown batting average problem explains what a batting average is, and merely instructs you to design a program to calculate any player's batting average. It is up to you to brainstorm the problem and determine the requirements of each phase. Let's take a closer look at each phase of the batting average problem.

1. Input is received.

To determine a program's input requirements, you must determine the pieces of data that are required for the program to complete its task. If we look at the batting average formula, we see that two values are needed to perform the calculation:

- The number of hits
- The number of times at bat

Because these values are unknown, the program will have to prompt the user to enter them. Each piece of input will be stored in a variable. You will have to declare those variables when you design the program, so it is helpful in this phase to think about each variable's name and data type. In the batting average program we will use the name *hits* for the variable that stores the number of hits, and the name *atBat* for the variable that holds the number of times at bat. Both of these values will be whole numbers, so these variables will be declared as *Integers*.

2. Some process (such as a calculation) is performed on the input.

Once the program has gathered the required input, it can proceed to use that input in any necessary calculations, or other operations. The batting average program will divide the number of hits by the number of times at bat. The result of the calculation is the player's batting average.

Keep in mind that when a mathematical calculation is performed, you typically want to store the

result of that calculation in a variable. So, you should think about the names and data types of any variables that are needed in this phase. In this example, we will use the name *battingAverage* for the variable that stores the player's batting average. Because this variable will store the result of a division, we will declare it as a *Real*.

3. Output is produced.

A program's output will typically be the result of the process or processes that it has performed. The output of the batting average program will be the result of the calculation, which is stored in a variable named *battingAverage*. The program will display this number in a message that explains what it is.

Now that we have identified the input, processing, and output requirements, we can create the pseudocode and/or flowcharts. First, we will write the pseudocode variable declarations:

```
Declare Integer hits
Declare Integer atBat
Declare Real battingAverage
```

Next we will write the pseudocode for gathering input. Recall that a program typically does two things when it needs the user to enter a piece of input on the keyboard: (1) it displays a message prompting the user for the piece of data, and (2) it reads the user's input and stores that data in a variable. Here is the pseudocode for reading the two pieces of input required by the batting average program:

```
Display "Enter the player's number of hits."
Input hits

Display "Enter the player's number of times at bat."
Input atBat
```

Next we will write the pseudocode for calculating the batting average:

```
Set battingAverage = hits / atBat
```

And finally, we write the pseudocode for displaying the output:

```
Display "The player's batting average is ", battingAverage
```

Now we can put all of these pieces together to form a complete program. **Program 2-15** shows the pseudocode program with comments, and **Figure 2-20** shows the flowchart.

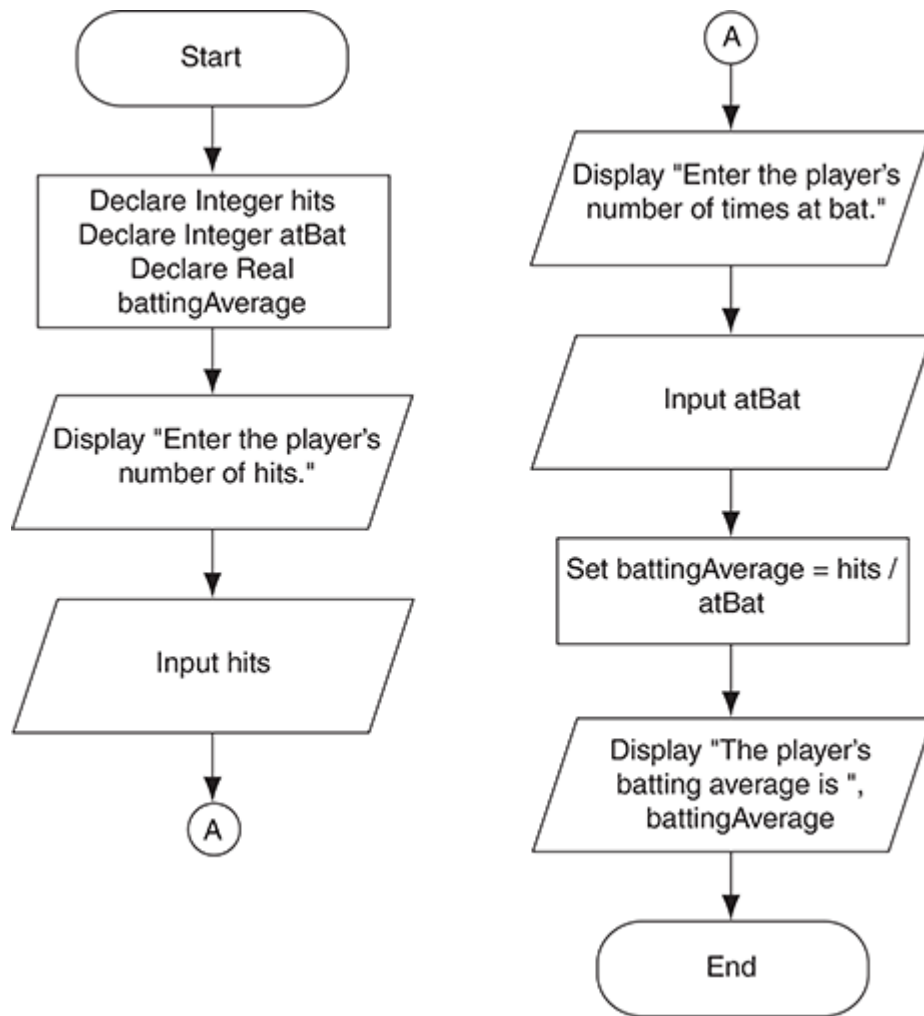


Figure 2-20 Flowchart for **Program 2-15**

Program 2-15

```

1  // Declare the necessary variables.
2  Declare Integer hits
3  Declare Integer atBat
4  Declare Real battingAverage
5
6  // Get the number of hits.
7  Display "Enter the player's number of hits."
8  Input hits
9
10 // Get the number of times at bat.
11 Display "Enter the player's number of times at bat."
12 Input atBat
13
14 // Calculate the batting average.
15 Set battingAverage = hits / atBat
16
17 // Display the batting average.
18 Display "The player's batting average is ", battingAverage

```

Program Output (with Input Shown in Bold)

```
Enter the player's number of hits.  
150 [Enter]  
Enter the player's number of times at bat.  
500 [Enter]  
The player's batting average is 0.3
```

Summary

As a beginning student, whenever you have trouble getting started with a program design, determine the program's requirements as follows:

1. **Input:** Carefully study the problem and identify the pieces of data that the program needs to read as input. Once you know what data is needed as input, decide the names of the variables for those pieces of data, and their data types.
2. **Process:** What must the program do with the input that it will read? Determine the calculations and/or other processes that must be performed. At this time, decide the names and data types of any variables needed to hold the results of calculations.
3. **Output:** What output must the program produce? In most cases, it will be the results of the program's calculations and/or other processes.

Once you have determined these requirements, you will have an understanding of what the program must do. You will also have a list of variables and their data types. The next step is writing the algorithm in pseudocode, or drawing it as a flowchart.

2.9 Focus on Languages: Java, Python, and C++

This section discusses the ways that many of this chapter's topics are implemented in the Java, Python, and C++ programming languages. For a more detailed look at these languages, including several more example programs in each language, you can download the *Java Language Companion*, the *Python Language Companion*, and the *C++ Language Companion* from the publisher's website for this textbook. You can access the website by visiting www.pearson.com/gaddis.

Java

Input, Processing, and Output in Java

Setting Up a Java Program

When you start a new Java program, you must first write a *class declaration* to contain your Java code. For now, simply think of a class declaration as a container for Java code. Here is an example

```
public class Simple
{

}
```

The first line of the class declaration is called the *class header*. In this example, the class header reads:

```
public class Simple
```

The words *public* and *class* are key words in the Java language, and the word *Simple* is the name of the class. Notice that the words *public* and *class* are written in all lowercase letters. In Java, all key words are written in lowercase letters. If you mistakenly write an uppercase letter in a key word, an error will occur when you compile the program.

The class name, which in this case is *Simple*, does not have to be written in all lowercase letters because it is not a key word in the Java language. This is just a word that the programmer uses as the name of the class. Notice that the first character of the class name is written in uppercase. It is not required, but it is a standard practice in Java to write the first character of a class name in uppercase. Java programmers do this so class names are more easily distinguishable from the names of other items in a program.

Another important thing to remember about the class name is that it must be the same as the name of the file that the class is stored in. For example, if you create a class named *Simple* (as shown previously), that class declaration will be stored in a file named *Simple.java*. (All Java source code files must be named with the *.java* extension.)

Notice that a set of curly braces follows the class header. Curly braces are meant to enclose things, and these curly braces will enclose all the code that will be written inside the class. So, the next step is to write some code inside the curly braces.

Inside the class's curly braces you must write the definition of a method named `main`. A *method* is another type of container that holds code. When a Java program executes, it automatically begins running the code that is inside the `main` method. Here is how our `Simple` class will appear after we have added the `main` method declaration:

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

    }
}
```

The first line of the method definition, which is called the *method header*, begins with the words `public static void main` and so forth. At this point, don't be concerned about the meaning of any of these words. Just remember that you have to write the method header *exactly* as it is shown. Notice that a set of curly braces follow the method header. All of the code that you will write inside the method must be written inside these curly braces.

Displaying Screen Output in Java

To display text on the screen in Java, you use the following statements:

- `System.out.println()`
- `System.out.print()`

First, let's look at the `System.out.println()` statement. The purpose of this statement is to display a line of output. Notice that the statement ends with a set of parentheses. The text that you want to display is written as a string inside the parentheses. **Program 2-16** shows an example. (Remember, the line numbers are NOT part of the program! Don't type the line numbers when you are entering program code. The line numbers are shown for reference purposes only.)

Program 2-16 (Output.java)

```
1 public class Output
2 {
3     public static void main(String[] args)
4     {
5         System.out.println("My major is Computer Science.");
6         System.out.println("I plan to be a software developer.");
7         System.out.println("Programming is fun!");
8     }
9 }
```

Program Output

```
My major is Computer Science.
I plan to be a software developer.
Programming is fun!
```

The statements that appear in lines 5 through 7 end with a semicolon. Just as a period marks the end of a sentence in English, a semicolon marks the end of a statement in Java. You'll notice that some lines of code in the program do not end with a semicolon, however. For example, class headers and method headers do not end with a semicolon because they are not considered statements. Also, the curly braces are not followed by a semicolon because they are not considered statements. (If this is confusing, don't despair! As you practice writing Java programs more and more, you will develop an intuitive understanding of the difference between statements and lines of code that are not considered statements.)

Notice that the output of the `System.out.println()` statements appears on separate lines. When the `System.out.println()` statement displays output, it advances the output cursor (the location where the next item of output will appear) to the next line. That means the `System.out.println()` statement displays its output, and then the next thing that is displayed will appear on the following line.

The `System.out.print()` statement displays output, but it does not advance the output cursor to the next line. **Program 2-17** shows an example.

Program 2-17 (Output2.java)

```
1 public class Output2
2 {
3     public static void main(String[] args)
4     {
5         System.out.print("Programming");
6         System.out.print("is");
7         System.out.print("fun.");
8     }
9 }
```

Program Output

```
Programmingisfun.
```

Notice in the program output that all the words are jammed together into one long series of characters. If you want spaces to appear between the words, you have to explicitly display them. **Program 2-18** shows how we have to insert spaces into the strings that we are displaying, if we want the words to be separated on the screen. Notice that in line 5, we have inserted a space in the string, after the letter `g`, and in line 6, we have inserted a space in the string after the letter `s`.

Program 2-18 (Output3.java)

```
1 public class Output3
2 {
3     public static void main(String[] args)
4     {
5         System.out.print("Programming ");
6         System.out.print("is ");
7         System.out.print("fun.");
8     }
9 }
```

```
6      System.out.print("is ");
7      System.out.print("fun.");
8  }
9 }
```

Program Output

```
Programming is fun.
```

Variables in Java

In Java, variables must be declared before they can be used in a program. A variable declaration statement is written in the following general format:

```
DataType VariableName;
```

In the general format, *DataType* is the name of a Java data type, and *VariableName* is the name of the variable that you are declaring. The declaration statement ends with a semicolon. For example, the key word *int* is the name of the integer data type in Java, so the following statement declares a variable named *number*.

```
int number;
```

Table 2-6 lists the Java data types, gives their memory size in bytes, and describes the type of data that each can hold. Note that in this book, we will primarily use the *int*, *double*, and *String* data types.²

² Notice that *String* is written with an initial uppercase letter. To be correct, *String* is not a data type in Java, it is a class. We use it as a data type, though.

Table 2-6 Java Data Types

Data Type	What It Can Hold
<i>byte</i>	Integers in the range of –128 to +127
<i>short</i>	Integers in the range of –32,768 to +32,767
<i>int</i>	Integers in the range of –2,147,483,648 to +2,147,483,647
<i>long</i>	Integers in the range of –9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

<i>float</i>	Floating-point numbers in the range of $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$, with 7 digits of accuracy
<i>double</i>	Floating-point numbers in the range of $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$, with 15 digits of accuracy
<i>String</i>	Strings of text

Here are some other examples of Java variable declarations:

```
int speed;
double distance;
String name;
```

Several variables of the same data type can be declared with the same declaration statement. For example, the following statement declares three *int* variables named *width*, *height*, and *length*.

```
int width, height, length;
```

You can also initialize variables with starting values when you declare them. The following statement declares an *int* variable named *hours*, initialized with the starting value 40:

```
int hours = 40;
```

Variable Names in Java

You may choose your own variable names (and class names) in Java, as long as you do not use any of the Java key words. The key words make up the core of the language and each has a specific purpose. See **Table 2-2** in the *Java Language Companion* for a complete list of Java key words.

The following are some specific rules that must be followed when naming variables in Java:

- The first character must be one of the letters a–z, A–Z, an underscore (`_`), or a dollar sign (`$`).
- After the first character, you may use the letters a–z or A–Z, the digits 0–9, underscores (`_`), or dollar signs (`$`).
- Uppercase and lowercase characters are distinct. This means *itemsOrdered* is not the same as *itemsordered*.
- Variable names cannot include spaces.

Program 2-19 shows an example with three variable declarations. Line 5 declares a *String* variable named *name*, initialized with the string “Jeremy Smith.” Line 6 declares an *int* variable named *hours* initialized with the value 40. Line 7 declares a *double* variable named *pay*, initialized with the value 852.99. Notice that in lines 9 through 11, we use *System.out.println* to display the contents of each variable.

Program 2-19 (VariableDemo.java)

```
1 public class VariableDemo
2 {
3     public static void main(String[] args)
4     {
5         String name = "Jeremy Smith";
6         int hours = 40;
7         double pay = 852.99;
8
9         System.out.println(name);
10        System.out.println(hours);
11        System.out.println(pay);
12    }
13 }
```

Program Output

```
Jeremy Smith
40
852.99
```

Reading Keyboard Input in Java

To read keyboard input in Java, you have to create a type of object in memory known as a *Scanner* object. You can then use the *Scanner* object to read values from the keyboard, and assign those values to variables. **Program 2-20** shows an example of how this is done. (This is the Java version of pseudocode **Program 2-2** shown earlier in this chapter.) Let's take a closer look at the code:

- Line 1 has the following statement: `import java.util.Scanner;`

This statement is necessary to tell the Java compiler that we are going to create a *Scanner* object in the program.

- Line 7 creates a *Scanner* object and gives it the name `keyboard`.
- Line 8 declares an `int` variable named `age`.
- Line 10 displays the string `"What is your age?"`
- Line 11 reads an integer value from the keyboard and assigns that value to the `age` variable.
- Line 12 displays the string `"Here is the value that you entered:"`
- Line 13 displays the value of the `age` variable.

Program 2-20 (GetAge.java)

```
1 import java.util.Scanner;
2
3 public class GetAge
4 {
5     public static void main(String[] args)
6     {
```

```

7      Scanner keyboard = new Scanner(System.in);
8      int age;
9
10     System.out.println("What is your age?");
11     age = keyboard.nextInt();
12     System.out.println("Here is the value that you entered:");
13     System.out.println(age);
14 }
15 }

```

Program Output (with Input Shown in Bold)

```

What is your age?
24 [Enter]
Here is the value that you entered:
24

```

Notice that in line 11, we used the expression `keyboard.nextInt()` to read an integer from the keyboard. If we wanted to read a double from the keyboard, we would use the expression `keyboard.nextDouble()`. Moreover, if we want to read a string from the keyboard, we would use the expression `keyboard.nextLine()`.

Program 2-21 shows how a `Scanner` object can be used to read not only integers, but also `doubles` and strings:

- Line 7 creates a `Scanner` object and gives it the name `keyboard`.
- Line 8 declares a `String` variable named `name`, line 9 declares a `double` variable named `payRate`, and line 10 declares an `int` variable named `hours`.
- Line 13 uses the expression `keyboard.nextLine()` to read a string from the keyboard, and assigns the string to the `name` variable.
- Line 16 uses the expression `keyboard.nextDouble()` to read a `double` from the keyboard, and assigns it to the `payRate` variable.
- Line 19 uses the expression `keyboard.nextInt()` to read an integer from the keyboard, and assigns it to the `hours` variable.

Program 2-21 (GetInput.java)

```

1  import java.util.Scanner;
2
3  public class GetInput
4  {
5      public static void main(String[] args)
6      {
7          Scanner keyboard = new Scanner(System.in);
8          String name;
9          double payRate;
10         int hours;
11
12         System.out.print("Enter your name: ");

```

```
13     name = keyboard.nextLine();
14
15     System.out.print("Enter your hourly pay rate: ");
16     payRate = keyboard.nextDouble();
17
18     System.out.print("Enter the number of hours worked: ");
19     hours = keyboard.nextInt();
20
21     System.out.println("Here are the values that you entered:");
22     System.out.println(name);
23     System.out.println(payRate);
24     System.out.println(hours);
25 }
26 }
```

Program Output (with Input Shown in Bold)

```
Enter your name:  Connie Maroney [Enter]
Enter your hourly pay rate:  55.25 [Enter]
Enter the number of hours worked:  40 [Enter]
Here are the values that you entered:
Connie Maroney
55.25
40
```

Displaying Multiple Items with the + Operator in Java

When the + operator is used with strings, it is known as the *string concatenation operator*. To concatenate means to append, so the string concatenation operator appends one string to another. For example, look at the following statement:

```
System.out.println("This is " + "one string.");
```

This statement will display:

```
This is one string.
```

The + operator produces a string that is the combination of the two strings used as its operands. You can also use the + operator to concatenate the contents of a variable to a string. The following code shows an example:

```
number = 5;
System.out.println("The value is " + number);
```

The second line uses the `+` operator to concatenate the contents of the `number` variable with the string `"The value is "`. Although `number` is not a string, the `+` operator converts its value to a string and then concatenates that value with the first string. The output that will be displayed is:

```
The value is 5
```

Performing Calculations in Java

The Java arithmetic operators are nearly the same as those presented in [Table 2-1](#) earlier in this chapter:

<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>/</code>	Division
<code>%</code>	Modulus

Here are some examples of Java statements that use an arithmetic operator to calculate a value, and assign that value to a variable:

```
total = price + tax;
sale = price - discount;
population = population * 2;
half = number / 2;
leftOver = 17 % 3;
```

Java does not have an exponent operator, but it does provide a method named `Math.pow` for this purpose. Here is an example of how the `Math.pow` method is used:

```
result = Math.pow(4.0, 2.0);
```

The method takes two `double` arguments (the numbers shown inside the parentheses). It raises the first argument to the power of the second argument, and returns the result as a `double`. In this example, 4.0 is raised to the power of 2.0. This statement is equivalent to the following algebraic statement:

$$result = 4^2$$

Named Constants in Java

You create named constants in Java using the `final` key word in a variable declaration. The word `final` is written just before the data type. Here is an example:

```
final double INTEREST_RATE = 0.069;
```

This statement looks just like a regular variable declaration except that the word `final` appears before the data type, and the variable name is written in all uppercase letters. It is not required that the variable name appear in all uppercase letters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names.

Documenting a Java Program with Comments

To write a line comment in Java, you simply place two forward slashes (`//`) where you want the comment to begin. The compiler ignores everything from that point to the end of the line. Here is an example:

```
// This program calculates an employee's gross pay.
```

Multiline comments, or block comments, start with `/*` (a forward slash followed by an asterisk) and end with `*/` (an asterisk followed by a forward slash). Everything between these markers is ignored. Here is an example:

```
/*  
    This program calculates an employee's gross pay  
    including any overtime wages the employee has worked.  
*/
```

Python

Input, Processing, and Output in Python

Displaying Screen Output in Python

To display output on the computer's screen in Python, you use the `print` function. Here is an example of a statement that uses the `print` function to display the message *Hello world*.

```
print('Hello world')
```

To use the `print` function, you type the word `print`, followed by a set of parentheses. Inside the parentheses, you type an *argument*, which is the data that you want displayed on the screen. In this

example, the argument is `'Hello world'`. The quote marks will not be displayed when the statement executes, however. The quote marks simply specify the beginning and the end of the text that we wish to display. **Program 2-22** shows an example of a complete program that displays output on the screen. (Remember, the line numbers are NOT part of the program! Don't type the line numbers when you are entering program code. The line numbers are shown for reference purposes only.)

Program 2-22 (output.py)

```
1 print('My major is Computer Science.')
2 print('I plan to be a software developer.')
3 print('Programming is fun!')
```

Program Output

```
My major is Computer Science.
I plan to be a software developer.
Programming is fun!
```

In Python, you can enclose string literals in a set of single-quote marks (`'`) or a set of double-quote marks (`"`). The string literals in **Program 2-22** are enclosed in single-quote marks, but the program could also be written as shown here:

```
print("My major is Computer Science.")
print("I plan to be a software developer.")
print("Programming is fun!")
```

Python Variables

You do not declare variables in Python. Instead, you use an *assignment statement* to create a variable. Here is an example of an assignment statement:

```
age = 25
```

After this statement executes, a variable named `age` will be created and it will be assigned the integer value 25. Here is another example:

```
title = 'Vice President'
```

After this statement executes, a variable named `title` will be created and it will be assigned the string `'Vice President'`.

Variable Names in Python

You may choose your own variable names in Python, as long as you do not use any of the Python key words. (See [Table 2-1](#) in the *Python Language Companion* for a complete list of Python key words.) Additionally, you must follow these rules when naming variables in Python:

- A variable name cannot contain spaces.
- The first character must be one of the letters a through z, A through Z, or an underscore character (`_`).
- After the first character you may use the letters a through z or A through Z, the digits 0 through 9, or underscores.
- Uppercase and lowercase characters are distinct. This means the variable name `ItemsOrdered` is not the same as `itemsordered`.

Displaying Multiple Items with the `print` Function in Python

You can pass multiple arguments to the `print` function, and Python will print each argument's value on the screen, separated by a space. [Program 2-23](#) shows an example.

Program 2-23 (`print_multiple.py`)

```
1 room = 503
2 print('I am staying in room number', room)
```

Program Output

```
I am staying in room number 503
```

The statement in line 1 creates a variable named `room`, and assigns it the integer value 503. The statement in line 2 displays two items: a string literal followed by the value of the `room` variable. Notice that Python automatically displayed a space between these two items.

Reading Input from the Keyboard in Python

You can use Python's built-in `input` function to read input from the keyboard. The `input` function reads a piece of data that has been entered at the keyboard and returns that piece of data, as a string, back to the program. You normally use the `input` function in an assignment statement that follows this general format:

```
variable = input(prompt)
```

In the general format, `prompt` is a string that is displayed on the screen. The string's purpose is to instruct the user to enter a value. `variable` is the name of a variable that will reference the data that was entered on the keyboard. Here is an example of a statement that uses the `input` function to read data from the keyboard:

```
name = input('What is your name? ')
```

When this statement executes, the following things happen:

- The string `'What is your name? '` is displayed on the screen.
- The program pauses and waits for the user to type something on the keyboard, and then press the Enter key.
- When the Enter key is pressed, the data that was typed is returned as a string, and assigned to the `name` variable.

Program 2-24 shows a complete program that uses the `input` function to read two strings as input from the keyboard.

Program 2-24 (string_input.py)

```
1 first_name = input('Enter your first name: ')
2 last_name = input('Enter your last name: ')
3 print('Hello', first_name, last_name)
```

Program Output (with Input Shown in Bold)

```
Enter your first name:  Vinny [Enter]
Enter your last name:  Brown [Enter]
Hello Vinny Brown
```

Reading Numbers with the `input` Function in Python

The `input` function always returns the user’s input as a string, even if the user enters numeric data. For example, suppose you call the `input` function and the user types the number 72 and pressed the Enter key. The value that is returned from the `input` function is the string `'72'`. This can be a problem if you want to use the value in a math operation. Math operations can be performed only on numeric values, not strings.

Fortunately, Python has built-in functions that you can use to convert a string to a numeric type. **Table 2-7** summarizes two of these functions.

Table 2-7 Python Data Conversion Functions

Function	Description
<code>int(item)</code>	You pass an argument to the <code>int()</code> function and it returns the argument’s value converted to an integer.
<code>float(item)</code>	You pass an argument to the <code>float()</code> function and it returns the argument’s value converted to a floating-point number.

For example, the following code gets an integer and a floating-point number from the user:

```
hours = int(input('How many hours did you work? '))
pay_rate = float(input('What is your hourly pay rate? '))
```

Performing Calculations in Python

The Python arithmetic operators are nearly the same as those presented in **Table 2-1** earlier in this chapter:

+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
**	Exponent

Here are some examples of Python statements that use an arithmetic operator to calculate a value, and assign that value to a variable:

```
total = price + tax
sale = price - discount
population = population * 2
half = number / 2
leftOver = 17 % 3
result = 4**2
```

Program 2-25 shows an example program that performs mathematical calculations (This program is the Python version of pseudocode **Program 2-8** in your textbook.)

Program 2-25 (sale_price.py)

```
1 original_price = float(input("Enter the item's original price: "))
2 discount = original_price * 0.2
3 sale_price = original_price - discount
4 print('The sale price is', sale_price)
```

Program Output (with Input Shown in Bold)

```
Enter the item's original price: 100.00 [Enter]
The sale price is 80.0
```

In Python, the order of operations and the use of parentheses as grouping symbols work just as described earlier in this chapter.

Comments in Python

To write a line comment in Python, you simply place the `#` symbol where you want the comment to begin. The Python interpreter ignores everything from that point to the end of the line. Here is an example:

```
# This program calculates an employee's gross pay.
```

C++

Input, Processing, and Output in C++

The Parts of a C++ Program

The typical C++ program contains the following code:

```
#include <iostream>
using namespace std;

int main()
{
    return 0;
}
```

You can think of this as a skeleton program. As it is, it does absolutely nothing. However, you can add additional code to this program to make it perform an operation. When you write your first C++ programs, you will write other statements inside the curly braces, as indicated in [Figure 2-21](#).

```
#include <iostream>
using namespace std;

int main()
{
    ← You will write other C++
    return 0;      statements in this area.
}
```

Figure 2-21 The Skeleton C++ Program

Semicolons in C++

In C++, a complete programming statement ends with a semicolon. Just as a period marks the end of a sentence in English, a semicolon marks the end of a statement in C++. You'll notice that some lines of code in our example programs do not end with a semicolon, however. That's because not every line of code is a statement. If this is confusing, don't despair! As you practice writing C++ programs, you will develop an intuitive understanding of the difference between statements and lines of code that are not considered statements.

Displaying Screen Output in C++

To display output on the screen in C++, you write a `cout` statement (pronounced *see out*). A `cout` statement begins with the word `cout`, followed by the `<<` operator, followed by an item of data that is to be displayed. The statement ends with a semicolon. **Program 2-26** demonstrates. (Remember, the line numbers are NOT part of the program! They are shown for reference purposes only. Don't type the line numbers when you are entering program code.)

Program 2-26 (DisplayOutput.cpp)

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Hello world";
7     return 0;
8 }
```

Program Output

```
Hello world
```

The `<<` operator is known as the *stream insertion operator*. It always appears on the left side of the item of data that you want to display. Notice that in line 6, the `<<` operator appears to the left of the string `"Hello world"`. When the program runs, *Hello world* is displayed.

You can display multiple items with a single `cout` statement, as long as a `<<` operator appears to the left of each item. **Program 2-27** shows an example. In line 6, three items of data are being displayed: the string `"Programming "`, the string `"is "`, and the string `"fun."`. Notice that the `<<` operator appears to the left of each item.

Program 2-27 (DisplayMultipleItems.cpp)

```
1 #include <iostream>
2 using namespace std;
3
```

```
4 int main()
5 {
6     cout << "Programming " << "is " << "fun.";
7     return 0;
8 }
```

Program Output

```
Programming is fun.
```

When you display output with `cout`, the output is displayed as one continuous line on the screen, for example, look at **Program 2-28**. Even though the program has three `cout` statements, its output appears on one line.

Program 2-28 (DisplayOneLine.cpp)

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Programming ";
7     cout << "is ";
8     cout << "fun.";
9     return 0;
10 }
```

Program Output

```
Programming is fun.
```

The output comes out as one long line is because the `cout` statement does not start a new line unless told to do so. You can use the `endl` manipulator to instruct `cout` to start a new line. **Program 2-29** shows an example.

Program 2-29 (EndlManipulator.cpp)

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     cout << "Programming" << endl;
7     cout << "is" << endl;
8     cout << "fun." << endl;
```



```
9     return 0;
10 }
```

Program Output

```
Programming
is
fun.
```

Variables in C++

In C++, variables must be declared before they can be used in a program. A variable declaration statement is written in the following general format:

```
DataType VariableName;
```

In the general format, *DataType* is the name of a C++ data type, and *VariableName* is the name of the variable that you are declaring. The declaration statement ends with a semicolon. For example, the key word *int* is the name of the integer data type in C++, so the following statement declares a variable named *number*.

```
int number;
```

Table 2-8 lists some of the C++ data types, gives their memory size in bytes, and describes the type of data that each can hold. Note that in this book, we will primarily use the *int*, *double*, and *string* data types.³

³ To use the *string* data type, you must write the directive `#include <string>` at the top of your program. To be correct, *string* is not a data type in C++, it is a class. We use it as a data type, though.

Table 2-8 C++ Data Types

Data Type	What It Can Hold
<i>short</i>	Integers in the range of −32,768 to +32,767
<i>int</i>	Integers in the range of −2,147,483,648 to +2,147,483,647
<i>long</i>	Integers in the range of −2,147,483,648 to +2,147,483,647
<i>float</i>	Floating-point numbers in the range of $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{38}$, with 7 digits of

	accuracy
<i>double</i>	Floating-point numbers in the range of $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{308}$, with 15 digits of accuracy
<i>char</i>	Can store integers in the range of -128 to $+127$. Typically used to store characters.
<i>string</i>	Strings of text
<i>bool</i>	Stores the values <i>true</i> or <i>false</i>

Here are some other examples of variable declarations:

```
int speed;
double distance;
String name;
```

Several variables of the same data type can be declared with the same declaration statement. For example, the following statement declares three *int* variables named *width*, *height*, and *length*.

```
int width, height, length;
```

You can also initialize variables with starting values when you declare them. The following statement declares an *int* variable named *hours*, initialized with the starting value 40:

```
int hours = 40;
```

Variable Names in C++

You may choose your own variable names in C++, as long as you do not use any of the C++ key words. (See [Table 1-1](#) in the *C++ Language Companion* for a complete list of C++ key words.) The key words make up the core of the language and each has a specific purpose. The following are some additional rules that must be followed when naming variables:

- The first character must be one of the letters a–z, A–Z, or an underscore (`_`).
- After the first character, you may use the letters a–z or A–Z, the digits 0–9, underscores (`_`).
- Uppercase and lowercase characters are distinct. This means *itemsOrdered* is not the same as *itemsordered*.
- Variable names cannot include spaces.

Program 2-30 shows an example with three variable declarations. Notice that, because we are using a *string* variable, we have the `#include <string>` directive in line 2. Line 7 declares a *string* variable named *name*, initialized with the string “Jeremy Smith”. Line 8 declares an *int* variable named *hours* initialized with the value 40. Line 9 declares a *double* variable named *pay*,

initialized with the value 852.99. Notice that in lines 11 through 13, we use `cout` to display the contents of each variable.

Program 2-30 (Variables.cpp)

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main()
6 {
7     string name = "Jeremy Smith";
8     int hours = 40;
9     double pay = 852.99;
10
11     cout << name << endl;
12     cout << hours << endl;
13     cout << pay << endl;
14     return 0;
15 }
```

Program Output

```
Jeremy Smith
40
852.99
```

Reading Keyboard Input in C++

To read keyboard input in C++, you write a `cin` statement (pronounced *see in*). A `cin` statement begins with the word `cin`, followed by the `>>` operator, followed by the name of a variable. The statement ends with a semicolon. When the statement executes, the program will wait for the user to enter input at the keyboard, and press the Enter key. When the user presses Enter, the input will be assigned to the variable that is listed after the `>>` operator. (The `>>` operator is known as the *stream extraction operator*.) **Program 2-31** demonstrates.

Program 2-31 (KeyboardInput1.cpp)

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int age;
7
8     cout << "What is your age?" << endl;
9     cin >> age;
```

```
10     cout << "Here is the value that you entered:" << endl;
11     cout << age;
12     return 0;
13 }
```

Program Output (with Input Shown in Bold)

```
What is your age?
24 [Enter]
Here is the value that you entered:
24
```

The program shown in **Program 2-32** uses `cin` statements to read a string, an integer, and a `double`.

Program 2-32 (KeyboardInput2.cpp)

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 int main()
6 {
7     string name;
8     double payRate;
9     int hours;
10
11
12     cout << "Enter your first name." << endl;
13     cin >> name;
14     cout << "Enter your hourly pay rate." << endl;
15     cin >> payRate;
16     cout << "Enter the number of hours worked." << endl;
17     cin >> hours;
18
19     cout << "Here are the values that you entered:" << endl;
20     cout << name << endl;
21     cout << payRate << endl;
22     cout << hours << endl;
23     return 0;
24 }
```

Program Output (with Input Shown in Bold)

```
Enter your first name.
Connie [Enter]
Enter your hourly pay rate.
55.25 [Enter]
```

```
Enter the number of hours worked.  
40 [Enter]  
Here are the values that you entered:  
Connie  
55.25  
40
```

Special Case: Reading String Input Containing Spaces

A `cin` statement can read one-word string input, as previously shown in [Program 2-32](#), but it does not behave as you would expect when the user's input is a string containing multiple words, separated by spaces. If you want to read a string that contains multiple words, you must use the `getline` function. The `getline` function reads an entire line of input, including embedded spaces, and stores it in a `string` variable. The `getline` function looks like the following, where `inputLine` is the name of the `string` variable receiving the input.

```
getline(cin, inputLine);
```

Program 2-33 shows an example of how the `getline` function is used.

Program 2-33 (KeyboardInput3.cpp)

```
1 // This program demonstrates using the getline function  
2 // to read input into a string variable.  
3 #include <iostream>  
4 #include <string>  
5 using namespace std;  
6  
7 int main()  
8 {  
9     string name;  
10    string city;  
11  
12    cout << "Please enter your name." << endl;  
13    getline(cin, name);  
14  
15    cout << "Enter the city you live in." << endl;  
16    getline(cin, city);  
17  
18    cout << "Hello, " << name << endl;  
19    cout << "You live in " << city << endl;  
20    return 0;  
21 }
```

Program Output (with Input Shown in Bold)

```
Please enter your name.  
Kate Smith [Enter]  
Enter the city you live in.  
West Jefferson [Enter]  
Hello, Kate Smith  
You live in West Jefferson
```

Performing Calculations in C++

The C++ arithmetic operators are nearly the same as those presented in [Table 2-1](#) in your textbook:

+	Addition
−	Subtraction
*	Multiplication
/	Division
⋅	Modulus

Here are some examples of C++ statements that use an arithmetic operator to calculate a value, and assign that value to a variable:

```
total = price + tax;  
sale = price - discount;  
population = population * 2;  
half = number / 2;  
leftOver = 17 ⋅ 3;
```

C++ does not have an exponent operator, but it does provide a function named `pow` for this purpose. Here is an example of how the `pow` function is used:

```
result = pow(4.0, 2.0);
```

The function takes two `double` arguments (the numbers shown inside the parentheses). It raises the first argument to the power of the second argument, and returns the result as a `double`. In this example, 4.0 is raised to the power of 2.0. This statement is equivalent to the following algebraic statement:

$$result = 4^2$$

Named Constants in C++

You create named constants in C++ using the `const` key word in a variable declaration. The word `const` is written just before the data type. Here is an example:

```
const double INTEREST_RATE = 0.069;
```

It is not required that the variable name appear in all uppercase letters, but many programmers prefer to write them this way so they are easily distinguishable from regular variable names.

An initialization value must be given when declaring a variable with the `const` modifier, or an error will result when the program is compiled. A compiler error will also result if there are any statements in the program that attempt to change the value of a `const` variable.

Documenting a Program with Comments in C++

To write a line comment in C++, you simply place two forward slashes (`//`) where you want the comment to begin. The compiler ignores everything from that point to the end of the line. Here is an example:

```
// This program calculates an employee's gross pay.
```

Multiline comments, or block comments, start with `/*` (a forward slash followed by an asterisk) and end with `*/` (an asterisk followed by a forward slash). Everything between these markers is ignored. Here is an example:

```
/*  
    This program calculates an employee's gross pay  
    including any overtime wages the employee has worked.  
*/
```