12 ADVANCED MODULARIZATION TECHNIQUES

After studying Chapter 12, you should be able to:

- Understand local and global variables and encapsulation
- Pass a single value to a module
- Pass multiple values to a module
- Return a value from a module
- Use prewritten, built-in modules
- Create an IPO chart
- Understand the advantages of encapsulation
- Reduce coupling and increase cohesion in your modules

UNDERSTANDING LOCAL AND GLOBAL VARIABLES AND ENCAPSULATION

Throughout most of computer programming history, which now totals about 60 years, the majority of programs were written procedurally. A **procedural program** consists of a series of steps or procedures that take place one after another. The programmer determines the exact conditions under which a procedure takes place, how often it takes place, and when the program stops. The logic for every program you have developed so far using this book has been procedural.

TIP You first learned the term *procedural program* in Chapter 4.

It is possible to write procedural programs as one long series of steps. However, by now you should appreciate the benefits of **modularization**, or breaking down programs into reasonable units called modules, subroutines, functions, or methods. The following are benefits of modularization:

- It provides **abstraction**; in other words, it makes it easier to see the "big picture."
- It allows multiple programmers to work on a problem, each contributing one or more modules that later can be combined into a whole program.
- It allows you to reuse your work; you can call the same module from multiple locations within a program.
- It allows you to identify structures more easily.

TIP You first learned the term *modular* in Chapter 2; you learned about *abstraction* in Chapter 3.

TIP Languages that use only global variables are most likely to call their modules "subroutines." Languages that allow local variables and the passing of values are more likely to call their modules "procedures," "methods," or "functions."

Modularization provides many benefits, but using modules and methods in the way you have used them throughout this book also has two major drawbacks:

- Although the modules you have used allow multiple programmers to work on a problem, each programmer must know the names of all the variables used in other modules within the program.
- Although the modules you have used enable you to reuse your work by allowing you to call them from multiple locations within a program, you can't use the modules in other programs unless these programs use the same variable names.

These two limitations have not caused significant problems for you in the programs you have designed so far, for several reasons:

- You most likely have designed every program alone, without using others' modules, so your variable names did not need to agree with anyone else's.
- You most likely have designed each program from beginning to end yourself, either at a single sitting or at least within a relatively short time period, and so you knew and easily remembered all the variable names you declared.

- Your programs have been relatively small, seldom with more than a few variable names to remember.

However, when you become a professional programmer in the business world, many of your programming assignments will involve large applications that will require dozens or even hundreds of modules written by many people. Some modules that you need to use might have been written by others years ago; some modules you write might not be used by other programmers until years from now. Some modules might even be purchased from programmers who work outside your organization, perhaps in another country. It would be virtually impossible to create such programs without some conflicting variable names in the various modules.

In addition, suppose that you have a well-written module that you want to reuse. Consider a module that formats a name and address to fit on a mailing label. You would want to use this module in programs that mail letters to stockholders, invoices to current customers, orders to suppliers, and so on. It would be inconvenient and inefficient to write separate modules containing statements such as `print stockholderName`, `print customerName`, and `print supplierName`. Instead, you would want to create a single module containing a statement such as `print name`, and allow that module to handle data stored in any variable that represents a name.

So that you could more easily understand how to write computer programs, the programs you have written so far have been relatively small, and all the variables you have used throughout this book have been global variables. A **global variable** is one that is available to every module in a program. That is, every module has access to the variable, can use its value, and can change its value. When you declare a variable named `grandTotal` in a program's `housekeeping()` module, add a value to it in a `mainLoop()` module, and print it in a `finish()` module, then `grandTotal` is a global variable within that program. If you tried to reuse the `mainLoop()` or `finish()` module in another program in which the grand total value was named `finalTotal`—or any name other than `grandTotal`—then the new program would fail.

With many older computer programming languages, all variables are global variables. Newer, more modularized languages allow you to use local variables as well. A **local variable** is one whose name and value are known only to its own module. A local variable is declared within a module and ceases to exist when the module ends. Within a module, a variable is said to be **in scope**—that is, existing and usable—from the moment it is declared until it ceases to exist; then it is **out of scope**.

TIP A locally declared variable always goes out of scope when its module ends. In some programming languages, you can purposely destroy variables earlier.

When you declare local variables within modules, you usually do so as the first step within a module, but some languages allow you to declare variables at any point within a module. Sometimes, you declare a local variable because the value is needed only within one module. At other times, the variable is needed in other modules within a program, but you still choose to declare local variables to gain some of the advantages they provide. When you use local variables:

- Programmers of other modules do not need to know the names of your variables.
- Each module becomes an enclosed unit, declaring all the variables it needs. Using this approach, you can more easily reuse your modules in other programs, regardless of the names of the other variables declared in those programs.

As an example of a program that uses local variables because they are needed within only one module, consider a very simple program that asks a student just one arithmetic question. For simplicity, this example won't loop; it provides a single user with a single question. A program such as this one could be contained in a single main module, but you can divide it into three separate modules, as shown in Figure 12-1. The program contains three steps: `housekeeping()`, `askQuestion()`, and `finish()`.

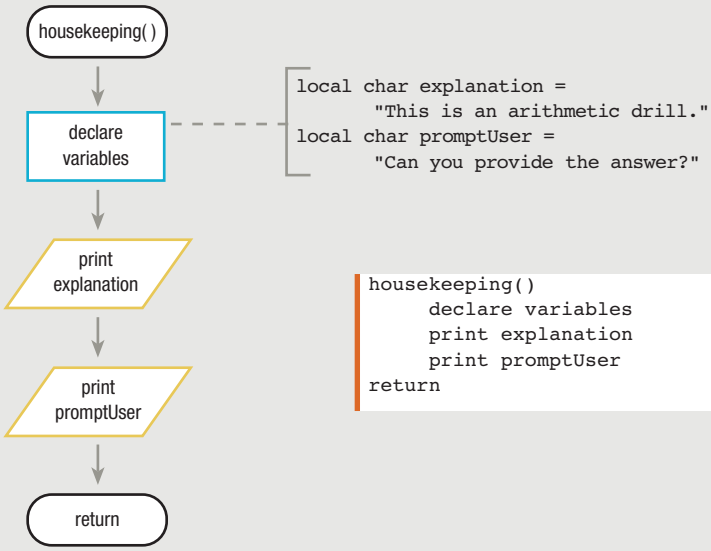
FIGURE 12-1: MAINLINE LOGIC FOR A PROGRAM THAT USES ONLY LOCAL VARIABLES


```
start
     perform housekeeping()
     perform askQuestion()
     perform finish()
stop
```

Figure 12-2 shows the `housekeeping()` module, in which directions are displayed on the screen. Within `housekeeping()`, you can declare variables named `explanation` and `promptUser`; these hold the characters that the user will see as directions. The `explanation` and `promptUser` variables can be local to the `housekeeping()` module because the `askQuestion()` and `finish()` modules never need access to these variables; `housekeeping()` uses the variables, but the other modules do not need to use the two variables or alter them in any way.

TIP Programming languages that use local variables do not require you to modify the variable declaration with the term *local*, as shown in Figure 12-2. The term *local* is used in Figure 12-2 just for emphasis.

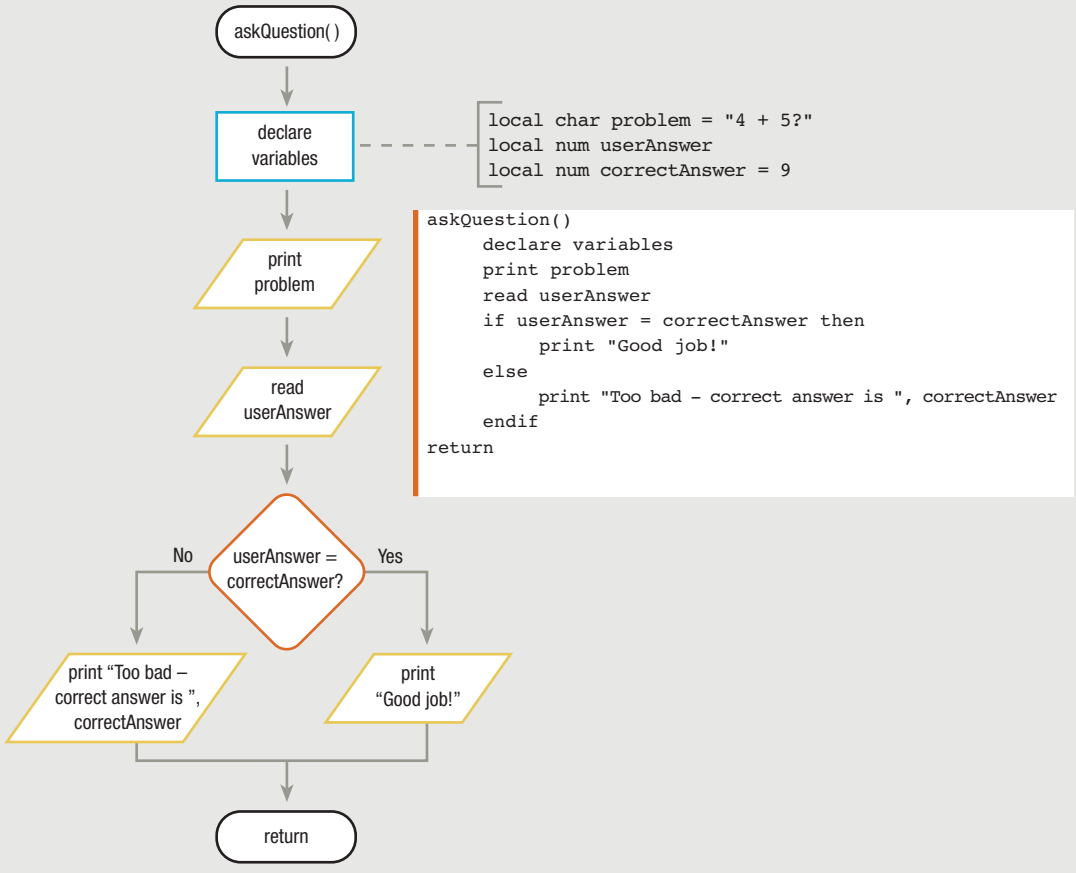
FIGURE 12-2: THE `housekeeping()` MODULE FOR A PROGRAM THAT USES ONLY LOCAL VARIABLES



Within the `askQuestion()` module of this arithmetic program, shown in Figure 12-3, you display an arithmetic problem, accept an answer, determine whether the answer is correct, and write a message. The `askQuestion()` module does not need to know about the `explanation` and `promptUser` variables, but the `askQuestion()` module does need a `problem` variable to hold the arithmetic problem to present to the user, a `userAnswer` variable in which to store the user's answer to the arithmetic problem, and a `correctAnswer` variable to hold the value to which the user's answer will be compared. Within the `askQuestion()` module, you declare these variables and use them. By the time you reach the end of the `askQuestion()` module, the three variables have served their purposes; there is no reason for any other module to have access to their values.

Figure 12-4 shows the `finish()` module for the arithmetic drill program. This module does not need to know the values of any of the variables in the first two modules called by the program; instead, it needs only its own locally declared `endingMessage`.

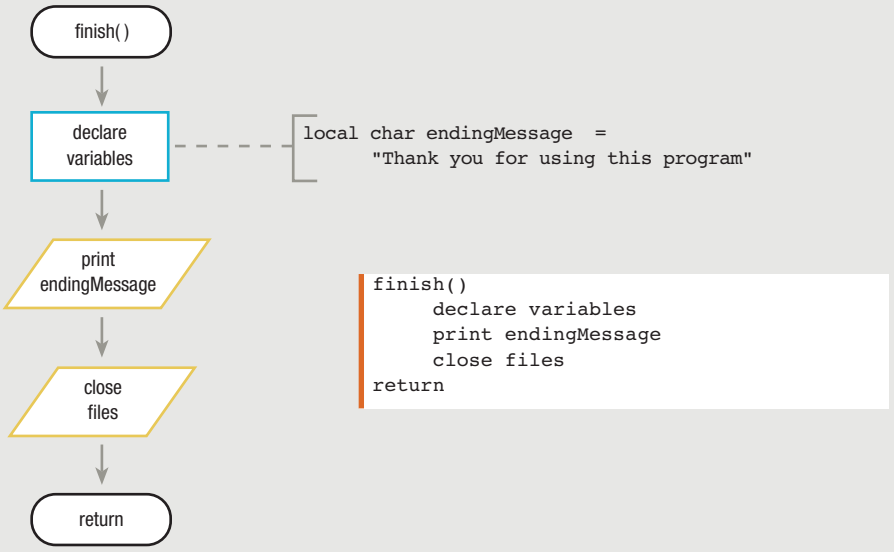
FIGURE 12-3: THE `askQuestion()` MODULE FOR A PROGRAM THAT USES ONLY LOCAL VARIABLES



This arithmetic drill program employs a principle known as **encapsulation**, which means that program components are bundled together. Encapsulation provides a means for **information hiding**, or **data hiding**, which means that the data or variables you use are completely contained within—and accessible only to—the module in which they are declared. In other words, the data and variables are "hidden from" the other program modules. Using encapsulation provides you with several advantages:

- Because each module contains all its own variable names, each module can be inserted easily into another program as a self-contained unit, requiring no changes within any modules in the new program.
- Because each module needs to use only the variable names declared within it, multiple programmers can create the individual modules without knowing the data names used by the other modules.
- Because the variable names in each module are hidden from all other modules, programmers can even use the same variable names as those used in other modules, and no conflict will arise.

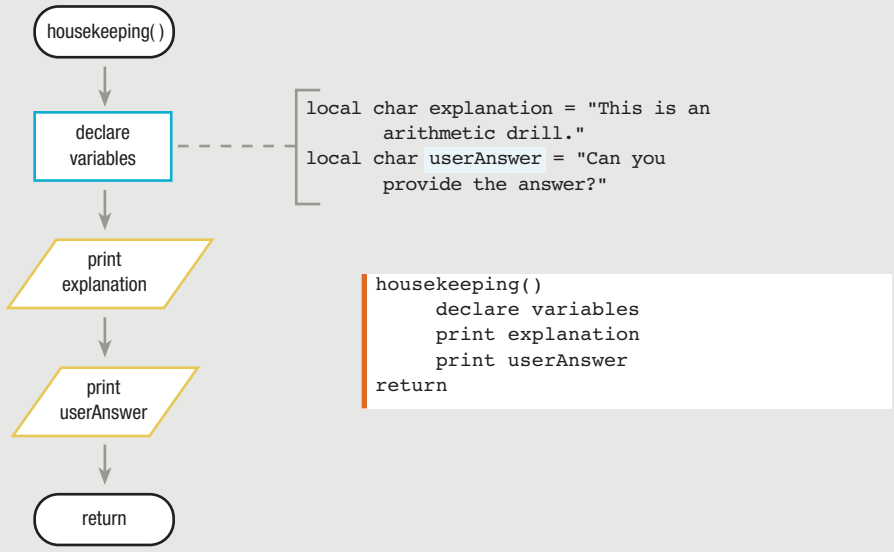
FIGURE 12-4: THE `finish()` MODULE FOR A PROGRAM THAT USES ONLY LOCAL VARIABLES



TIP The terms "encapsulation" and "information hiding" are often used synonymously, but encapsulation only facilitates (not guarantees) the hiding of information.

To illustrate this last point, consider the `housekeeping()` module for the arithmetic drill program shown in Figure 12-2. Programmers who work on this module can give the local variables any name they want. For example, a programmer could decide to call the `promptUser` variable `userAnswer`, as highlighted in Figure 12-5. In a program that employs local variables, giving the `housekeeping()` module's variable this name would have no effect on the usefulness of the identically named variable defined in the `askQuestion()` module. The two `userAnswer` variables are completely separate variables with unique memory addresses. One holds the character prompt "Can you provide the answer?" and the other holds a numeric user answer. Changing the value of `userAnswer` in one module (which is what happens when the user enters an arithmetic problem answer in the `askQuestion()` module) has no effect whatsoever on the separate `userAnswer` variable in the other module. A large program might contain dozens of modules, and each module might contain dozens of variable names. As programs grow in size and complexity, it is a great convenience for a programmer who is working on one module not to have to worry about conflicting with all the other variable names used in the program.

FIGURE 12-5: THE MODIFIED `housekeeping()` MODULE CONTAINING `userAnswer` VARIABLE



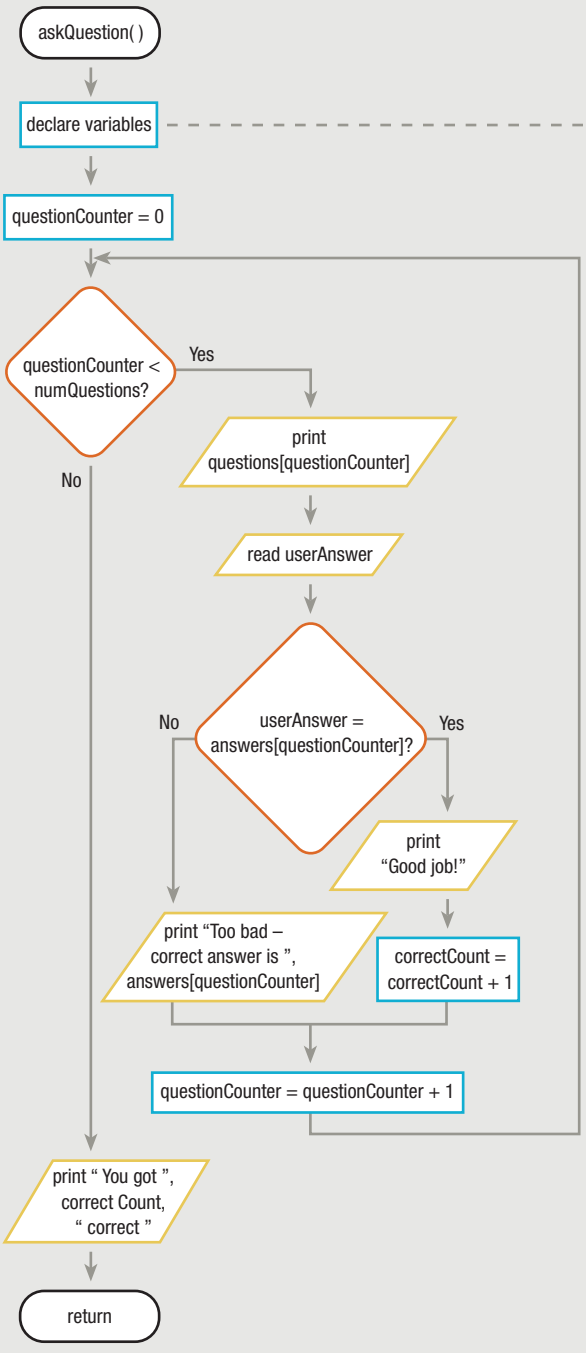
TIP

As an analogy, think of modules as households. Referring to a person named "Edward" in the Johnson household causes no confusion with a person named "Edward" in another household. The same name can be used locally within different households without conflict, in the same way that identical variable names can be used in multiple modules. Conversely, although two men can legally be named "Edward" within the same household, families almost always apply a nickname or qualification to at least one of the like-named people to avoid confusion, perhaps referring to "Ed" and "Eddie," or "Big Edward" and "Little Edward." Similarly, within any module, variable names must differ, if only by a single character.

PASSING A SINGLE VALUE TO A MODULE

It may be convenient for a programmer to use local variables without worrying about naming conflicts, but using local variables produces a problem in many programs. By definition, a local variable is accessible to one module only; however, sometimes more than one module needs access to the same variable value. Consider a new arithmetic drill program. Instead of a single arithmetic problem, it is more reasonable to expect such a program to ask the user a series of problems and keep score. Figure 12-6 shows a revised `askQuestion()` module that accesses an array to provide a series of five questions for the arithmetic drill. The module compares the user's answer to the correct answer that is stored in the corresponding position in a parallel array, and adds 1 to a `correctCount` variable when the answer is correct. After the user completes all five problems, but before the module ends, the value of `correctCount` is displayed.

FIGURE 12-6: THE MODIFIED `askQuestion()` MODULE THAT PROVIDES FIVE PROBLEMS AND KEEPS SCORE




```
local num userAnswer
local num questionCounter
local num numQuestions = 5
local array questions[5]
      questions[0] = "4 + 5?"
      questions[1] = "3 + 3?"
      questions[2] = "2 + 7?"
      questions[3] = "1 + 2?"
      questions[4] = "4 + 1?"
local array answers[5]
      answers[0] = 9
      answers[1] = 6
      answers[2] = 9
      answers[3] = 3
      answers[4] = 5
local num correctCount = 0
```

FIGURE 12-6: THE MODIFIED `askQuestion()` MODULE THAT PROVIDES FIVE PROBLEMS AND KEEPS SCORE (CONTINUED)

```
askQuestion()
     declare variables
     questionCounter = 0
     while questionCounter < numQuestions
          print questions[questionCounter]
          read userAnswer
          if userAnswer = answers[questionCounter] then
               print "Good job!"
               correctCount = correctCount + 1
          else
               print "Too bad - correct answer is ", answers[questionCounter]
          endif
          questionCounter = questionCounter + 1
     endwhile
     print " You got " , correctCount,  " correct"
return
```

The module shown in Figure 12-6 correctly counts and displays the number of correct answers for the user. However, suppose when the user completes the arithmetic drill, you want to print not only the count of correct answers, but also the percentage of correct answers along with one of two messages based on the user's performance. With these additions to the post-problem-solving process, there are enough steps involved that you decide to place these steps in their own module, named `finalStatistics()`. In other words, you want to be able to add to the user's `correctCount` value in the `askQuestion()` module, but you want to be able to determine the `correctCount` percentage and display it from within the `finalStatistics()` module. You must declare a `correctCount` variable, but where?

- If `correctCount` is declared locally within the `askQuestion()` module so you can add to it when the user answers correctly, then the `finalStatistics()` module does not have access to it.
- If `correctCount` is declared locally in the `finalStatistics()` module so you can compute the correct percentage and display it, then the `askQuestion()` module cannot add to it.
- If you attempt to solve the dilemma by declaring a local `correctCount` variable in each module, they are not the same variable; that is, they do not have the same memory address. Therefore, adding to the `correctCount` variable in one module does not alter the value of the unique `correctCount` variable in the other module.
- If you decide not to use local variables but declare `correctCount` as a global variable, the program will work, but you will have avoided using the principle of encapsulation and will have lost the advantages it provides.

The solution to using a locally declared variable within another module lies in a program's ability to pass the value of a local variable from one module to the other. **Passing a value** means sending a copy of data in one module of a program to another module for use. Exactly how you accomplish this differs slightly among languages, but it usually involves including the name of the variable that holds the value you want to pass within parentheses in the call to the module that needs to receive a copy of the value. Figure 12-7 provides an overview of the process. A value is sent from the method call in the `askQuestion()` module into the `finalStatistics()` module. When the `finalStatistics()` module is complete, program control returns to the `askQuestion()` module, where it would proceed with any additional statements in the module; in this case, the next task is to return from the `askQuestion()` module to the mainline program logic. Figure 12-8 shows the flowchart and pseudocode that modify the `askQuestion()` module to pass a copy of the `correctCount` value to the `finalStatistics()` module.

FIGURE 12-7: PASSING `correctCount` TO THE `finalStatistics()` MODULE

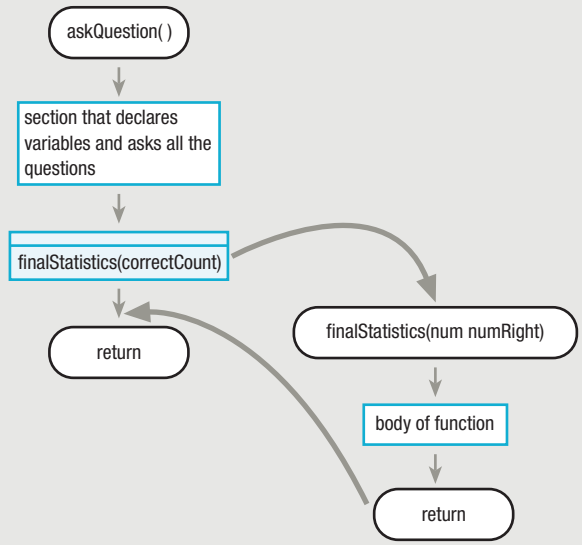


FIGURE 12-8: THE MODIFIED `askQuestion()` MODULE THAT PASSES `correctCount` TO A `finalStatistics()` MODULE

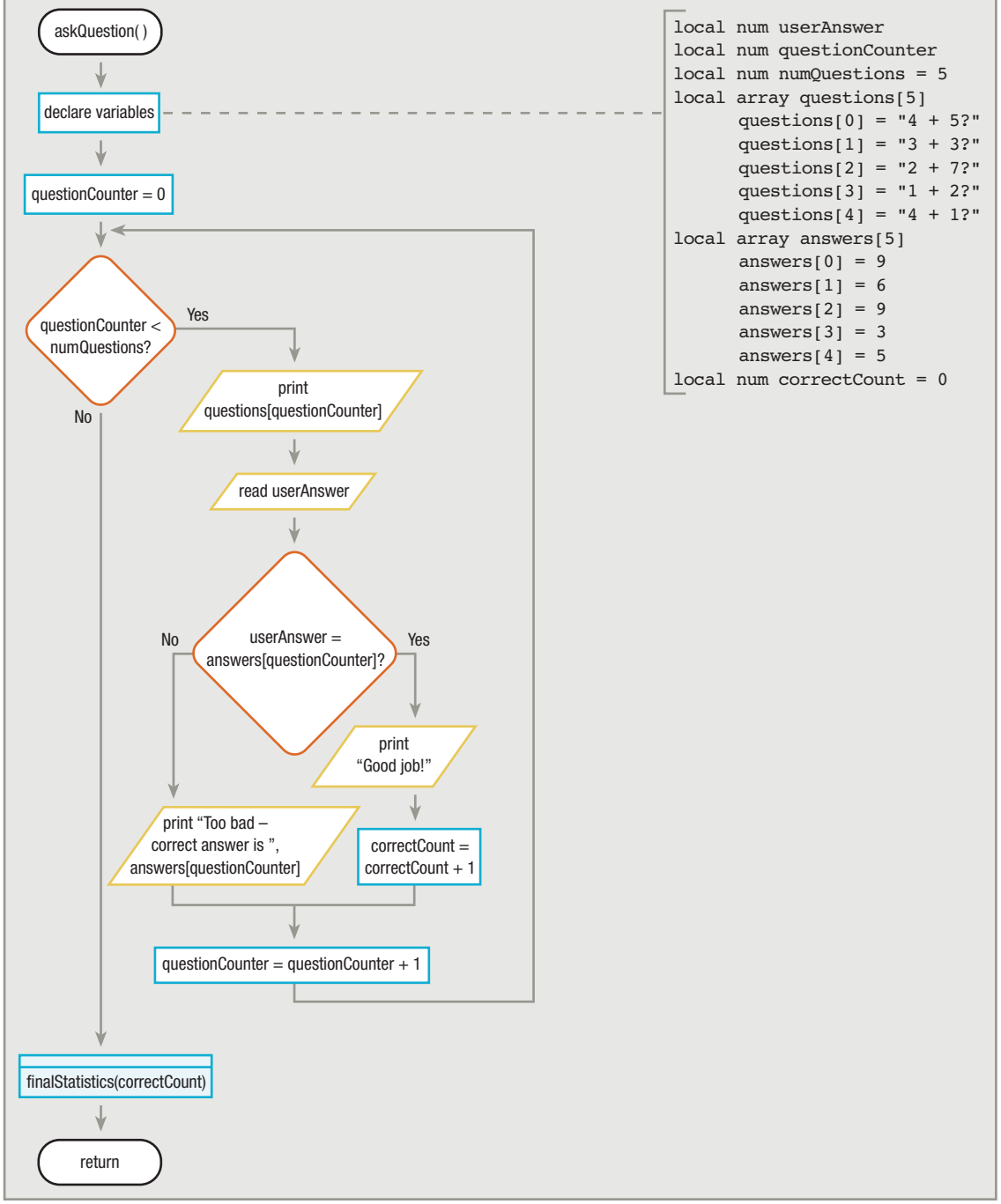


FIGURE 12-8: THE MODIFIED `askQuestion()` MODULE THAT PASSES `correctCount` TO A `finalStatistics()` MODULE (CONTINUED)

```
askQuestion()
     declare variables
     questionCounter = 0
     while questionCounter < numQuestions
          print questions[questionCounter]
          read userAnswer
          if userAnswer = answers[questionCounter] then
               print "Good job!"
               correctCount = correctCount + 1
          else
               print "Too bad - correct answer is ", answers[questionCounter]
          endif
          questionCounter = questionCounter + 1
     endwhile
     perform finalStatistics(correctCount)
return
```

Figure 12-9 shows the `finalStatistics()` module for the program. To prepare this module to receive a copy of the `correctCount` value, you declare a name for the passed value within parentheses in the **module header**, or introductory title statement of the module. The passed variable named within the module header is a parameter to the function.

TIP

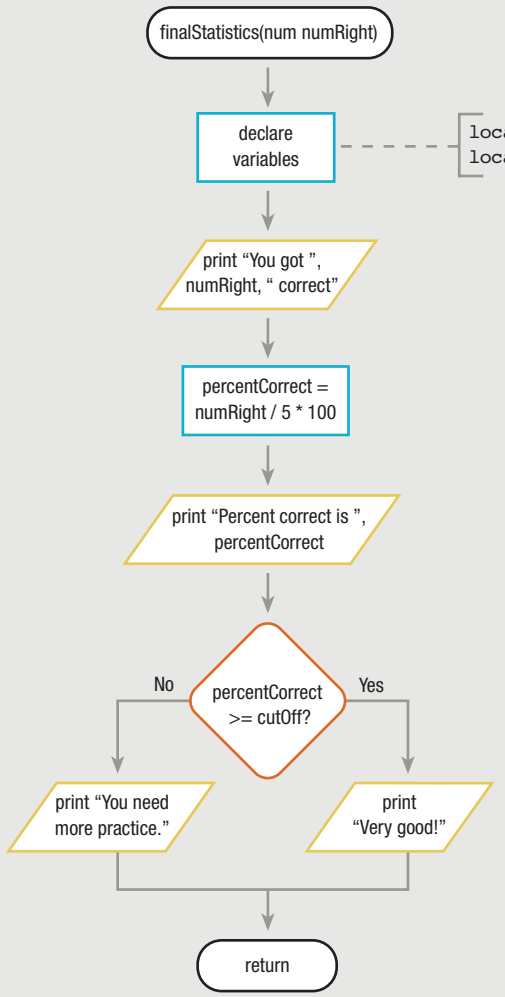
The words "argument" and "parameter" are often used interchangeably, although many programmers make a clear distinction between the two. An **argument** is the expression in the comma-separated list in a function call, while a **parameter** is an object or reference that is declared in a function prototype (declaration) or definition (header).

In Figure 12-9, the `finalStatistics()` module declares a numeric variable named `numRight` in its header statement. Declaring a variable within the parentheses of the module header indicates that this variable is not a regular variable declared locally within the module; it is a special variable that is local to the module but receives its value from the outside. In Figure 12-9, `numRight` receives its value when the `askQuestion()` module in Figure 12-8 calls the `finalStatistics()` module. The `askQuestion()` module passes the value of `correctCount` to `finalStatistics()`; then, within the `finalStatistics()` module, `numRight` takes on the value of `correctCount`, and the percentage of correct answers is calculated using `numRight`.

TIP

Passing a copy of a value to a module sometimes is called *passing by value*. Some languages allow you to pass the actual memory address of a variable to a module; this is called *passing by reference*. When you pass by reference, you lose some of the advantages of information hiding because the module has access to the address of the passed variable, not just to a copy of the value of the passed variable. However, program performance improves because the computer doesn't have to make a copy of the value, thereby saving time (and in the case of very large passed objects, saving significant memory). Both the ability to pass by reference and the syntax to do so vary by programming language.

FIGURE 12-9: THE `finalStatistics()` MODULE THAT RECEIVES `correctCount` VALUE AND CALLS IT `numRight`




```
finalStatistics(num numRight)
     declare variables
     print "You got ", numRight, " correct"
     percentCorrect = numRight / 5 * 100
     print "Percent correct is ",
          percentCorrect
     if percentCorrect >= cutOff then
          print "Very good!"
     else
          print "You need more practice."
     endif
return
```

The `finalStatistics()` module contains its own local variables, `percentCorrect` and `cutOff`. The `percentCorrect` variable is used to hold a calculated percentage of correct answers based on five arithmetic questions, and `cutOff` is used to compare the user's final percentage score with a minimum acceptable percentage. Any module can contain some values that are passed in, like `numRight`, and some that are stored in locally declared variables, such as `percentCorrect` and `cutOff`. The only restriction is that within any module, each variable must have a unique name; however, those names might or might not match any other variable names in other modules. For example, within the `finalStatistics()` module of the arithmetic drill program, you could choose to name the passed local value `correctCount` instead of `numRight`, giving it the same name as its counterpart in the `askQuestion()` module. Whether the name of the variable that holds the count in `finalStatistics()` is the same as that of the corresponding value in the `askQuestion()` module is irrelevant. The `correctCount`

variable used as an argument to `finalStatistics()` within the `askQuestion()` module and the numeric parameter in the `finalStatistics()` module represent two unique memory locations, no matter what name you decide to give the variable that holds the count of right answers within the `finalStatistics()` module.

PASSING MULTIPLE VALUES TO A MODULE

In the `finalStatistics()` module of the arithmetic drill program in Figure 12-9, `numRight`, `percentCorrect`, and `cutOff` all remain in scope from the point at which they are declared until the end of the module. After the `finalStatistics()` module receives its parameter, it contains everything it needs to calculate the percentage of addition problems that the user answered correctly and to determine which message to display. You could easily insert this self-contained module into different programs that calculate correct percentages—for example, programs that display subtraction or multiplication problems, or even those that ask history or grammar questions. As long as those programs pass a value indicating the number of correct answers, no matter what the name of the counter variable is in those programs, the `finalStatistics()` module works correctly—with one major flaw. The `finalStatistics()` module calculates the correct percentage only when the calling program contains exactly five problems—it divides the `numRight` variable by a constant, 5. A more useful `finalStatistics()` module accepts two parameters—one containing the number of correct user answers, and another containing the total number of possible correct answers. Then, whether you call `finalStatistics()` from a program containing a three-question quiz or a 200-item exam, the percentage will be correct.

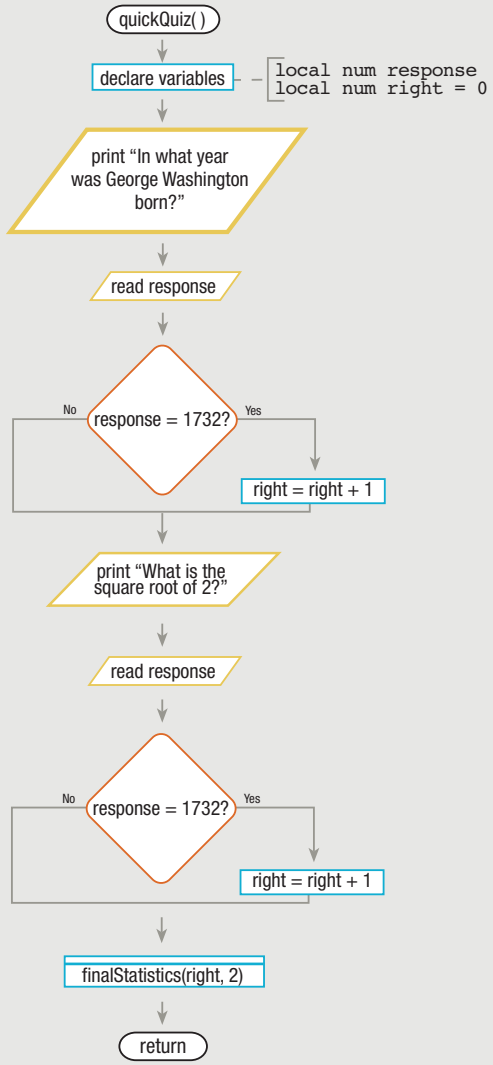
Figure 12-10 shows a module named `quickQuiz()` that includes a call to a `finalStatistics()` module that accepts two parameters, `numRight` and `numPossible`. This version of the `finalStatistics()` module differs from the version of the module in Figure 12-9 in only the highlighted areas. The addition of the `numPossible` parameter makes the module more flexible. The series of parameters that appears in the module header is called a **parameter list**. You could call this module from the `askQuestion()` module shown in Figure 12-8, replacing the current highlighted call to `finalStatistics()` with the statement `perform finalStatistics(correctCount, numQuestions)`. With this call, the `finalStatistics()` module would accept the value of `correctCount` and place it in `numRight`, and then accept the value of `numQuestions` and place it in `numPossible`. Alternatively, as shown in Figure 12-10, you could use the same module with a two-question quiz, passing a variable (`right`) and a constant (2), and the percentage correct would still be accurate.

Notice several important facts about the `finalStatistics()` method header in Figure 12-10:

- The two parameters in the header are separated by a comma. This is the convention in most modern languages such as Java, C++, and C#—any number of parameters is acceptable, but each must be separated from the others using a comma.
- Each parameter consists of both a data type and an identifier. This also is the convention in most modern programming languages. The parameter list might contain any combination of numeric and character items, but the data type of each must be explicitly mentioned with the identifier.
- Each identifier adheres to the rules for naming variables you have used throughout this book—specifically, no spaces are allowed in the identifier names, but each identifier might contain letters and digits.

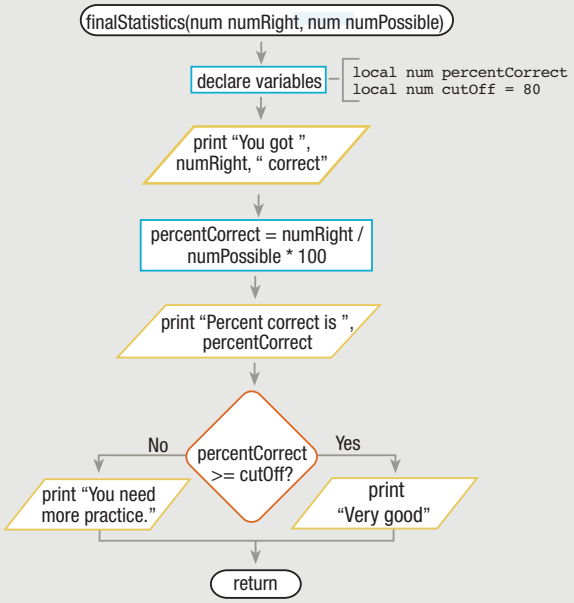
- The order of the parameters is very important. The only way that a module assigns values to the variables named in its parameter list is based on the order in which the arguments are passed from the calling module; the values passed into a module are assigned sequentially as they are received. If, for example, you pass variables containing the values 5 and 100 to the `finalStatistics()` module in that order, the module will display a 5 percent correct result. However, if you reverse the order of the passed values, sending 100 and 5, the module produces a very different result—2,000 percent correct.

FIGURE 12-10: THE `quickQuiz()` MODULE THAT PASSES TWO ARGUMENTS TO THE `finalStatistics()` MODULE




```
quickQuiz()
     declare variables
     print "In what year was George
        Washington born?"
     read response
     if response = 1732 then
          right = right + 1
     endif
     print "What is the square root of 2?"
     read response
     if response = 1.732 then
          right = right + 1
     endif
     perform finalStatistics(right, 2)
return
```

FIGURE 12-10: THE `quickQuiz()` MODULE THAT PASSES TWO ARGUMENTS TO THE `finalStatistics()` MODULE (CONTINUED)




```
finalStatistics(num numRight, num numPossible)
      declare variables
      print "You got ", numRight, " correct"
      percentCorrect = numRight / numPossible
         * 100
      print "Percent correct is ",
            percentCorrect
      if percentCorrect >= cutOff then
            print "Very good!"
      else
            print "You need more practice."
      endif
return
```

TIP

If you send arguments that are the wrong data types to a module, or you send too many or too few, you receive a syntax error message and the program will not compile or execute. However, if you send arguments that are the correct data types but that represent the wrong values, then you create a logical error that produces incorrect output. For example, if a module expects two numeric parameters representing a retail price and a discount in that order, and you pass 100.00 and 20.00, the module might correctly bill a customer a discounted price of $80.00. If you pass 20.00 and 100.00 by mistake, the module will perform a subtraction and produce a bill that indicates the customer has an $80.00 credit.

You can call a module from other modules in a variety of ways—you can use any combination of variables and constants as arguments in a module call. For example, consider the module shown in Table 12-1. The `printCustOrder()` module header shows that it accepts four parameters; the module uses the argument information to calculate a price and print a customer order. To use that module, you could call it in any of the ways shown in the middle section of the table, using any combination of variables and constants as arguments. The bottom part of the table shows some illegal calls and explains why each is unacceptable.

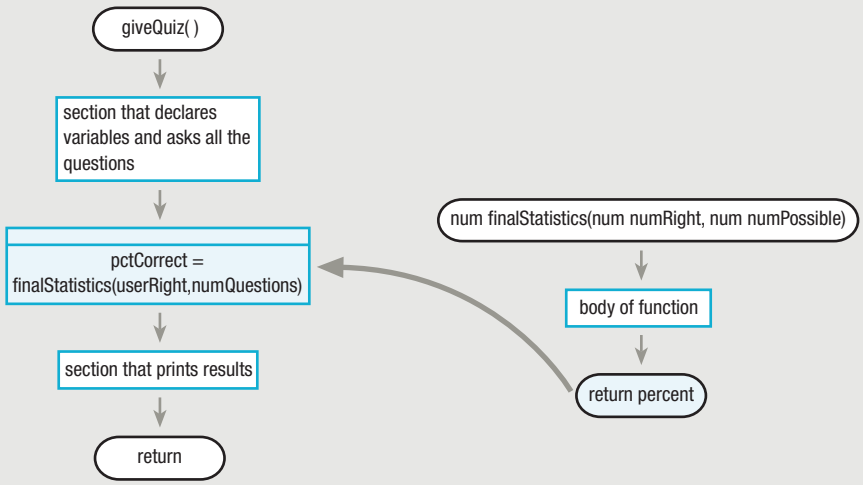
TABLE 12-1: THE `printCustOrder()` MODULE

Module header	
`printCustOrder(num itemNum, num quantity, char custName, char custAddress)`	
Legal calls from a module that contains declared variables named `stockNumber`, `amount`, `lastName`, `firstName`, and `streetAddress`	
Call using four variables	`printCustOrder(stockNumber, amount, lastName, streetAddress)`
Call using three variables and a constant	`printCustOrder(stockNumber, amount, lastName, "Will pick up")`
Call using two variables and two constants	`printCustOrder(stockNumber, 1, "Resident", streetAddress)`
Call using four constants	`printCustOrder(1342, 12, "Lewis", "900 Evergreen Avenue")`
Illegal calls from a module that contains declared variables named `stockNumber`, `amount`, `lastName`, and `streetAddress`	
Not enough arguments	`printCustOrder(stockNumber, amount, lastName)`
Too many arguments	`printCustOrder(stockNumber, amount, lastName, firstName, streetAddress)`
Arguments do not exist as variables in the calling function	`printCustOrder(itemNum, quantity, custName, custAddress)`
Arguments do not match order in module header	`printCustOrder(stockNumber, lastName, amount, streetAddress)`

RETURNING A VALUE FROM A MODULE

Suppose you decide to organize the arithmetic drill program from Figure 12-10 so that the `finalStatistics()` module still computes the user's correct percentage, but the calling module handles the printing of the final statistics. In this case, you pass the values of the count of correct questions and the total number of questions available to the `finalStatistics()` module as before, but the `finalStatistics()` module must **return the value** of the calculated correct percentage back to the calling module. Just as you can pass a value into a module, you can pass back, or return a value to a calling module. Usually, this is accomplished within the `return` statement of the called module. For example, Figure 12-11 shows an overview of how a value is passed back from the `finalStatistics()` module and stored in the `giveQuiz()` module. Figure 12-12 shows the flowchart and pseudocode for a `giveQuiz()` module that calls a rewritten `finalStatistics()` module.

FIGURE 12-11: RETURNING `percent` FROM THE `finalStatistics()` MODULE TO `pctCorrect` IN THE `giveQuiz()` MODULE



The `giveQuiz()` module in Figure 12-12 declares an array of questions and a parallel array holding correct answers. The module displays each question in sequence, compares the user's answer to the correct answer, and determines whether to add 1 to a variable used to keep track of the number of correct responses. Notice that near the end of the `giveQuiz()` module, you call the `finalStatistics()` module and pass in the `userRight` and `numQuestions` values. In the same statement, you assign the return value of the `finalStatistics()` module to the variable named `pctCorrect`. The `pctCorrect` variable is declared locally within the `giveQuiz()` module; its purpose is to receive the value returned by the `finalStatistics()` module. Then, you can use the value of the `pctCorrect` variable within the remainder of the `giveQuiz()` module.

TIP

Within the `giveQuiz()` module in Figure 12-12, the `finalStatistics()` module is called without using the word `perform`. The module actually is performed in the same way as all other modules you have performed using this book. In a flowchart or pseudocode, it would be perfectly acceptable to write `pctCorrect = perform finalStatistics(userRight,numQuestions)`. However, in this book, the word `perform` is eliminated in module examples that return a value, for two reasons: for simplicity in an already-complicated statement, and because the resulting syntax (eliminating `perform`) closely resembles that of popular modern languages such as C++, C#, and Java.

Notice that the variable type name `num` is used as the first word in the header of the `finalStatistics()` module in Figure 12-12. The use of a data type preceding the method header indicates the type of data that will be returned by the module; this use follows the format for methods that return values in many programming languages, such as C++, Java, and C#. The return type of a method is also called the **method's type** or **method's return type**.

FIGURE 12-12: A `giveQuiz()` MODULE THAT SENDS VALUES TO AND RECEIVES A VALUE RETURNED FROM A `finalStatistics()` MODULE

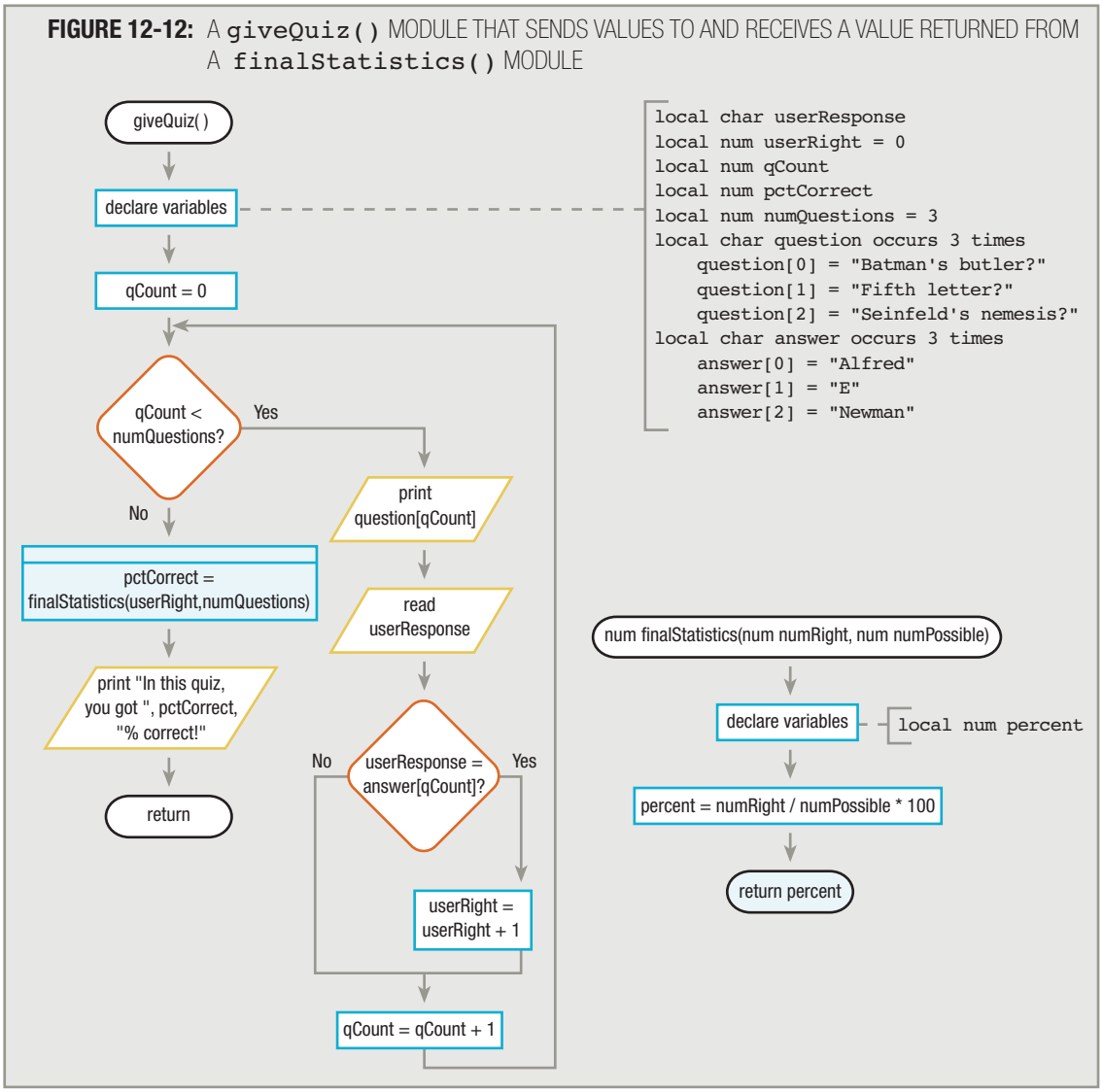


FIGURE 12-12: A `giveQuiz()` MODULE THAT SENDS VALUES TO AND RECEIVES A VALUE RETURNED FROM A `finalStatistics()` MODULE (CONTINUED)

```
giveQuiz()
     declare variables
     qCount = 0
     while qCount < numQuestions
          print question[qCount]
          read userResponse
          if userResponse = answer[qCount] then
               userRight = userRight + 1
          endif
          qCount = qCount + 1
     endwhile
     pctCorrect = finalStatistics(userRight,numQuestions)
     print "In this quiz, you got ", pctCorrect, "% correct!"
return
num finalStatistics(num numRight, num numPossible)
     declare variables
     percent = numRight / numPossible * 100
return percent
```

TIP A function or module should have, at most, one return value, and the `return` statement should always be the last statement in the module. Following these rules complies with the principles of structured programming you have used throughout this book. Recall from Chapter 2 that a structure can have only one entry point and one exit point; because a `return` statement provides a module's exit point, if the module is structured, it will have only one `return` statement, and hence, one return value.

TIP In several programming languages, such as Java, C++, and C#, if a module does not return a value, then the return type you list in the header is void, and the method is referred to as a void method. The word **void** means "empty" or "nothing." You have seen many modules that do not return values throughout this book; their `return` statement simply contains `return`.

TIP If you place statements within a module after the `return` statement, those statements will never execute. They are examples of **unreachable code**, or **dead code**.

USING PREWRITTEN, BUILT-IN MODULES

Many programming languages contain **built-in methods**, or **built-in functions**—prewritten modules that perform frequently needed tasks. For example, many languages contain a module that calculates the square root of a number. Within a program, you could perform the necessary calculations yourself, but it's a tedious process, and one that should not have to be rewritten by every programmer who needs it. The creators of many language compilers include a square root module so that programmers can use their valuable time to solve problems that are more unique to their business.

The only way you can discover whether the language you are using contains a built-in module such as a square root method is to examine the program language documentation; for example, you can read the manual that accompanies a language compiler, search through online help, or examine language-specific textbooks. When you find an available module that suits your needs, you need to discover three facts:

1. The name of the method
2. The arguments you need to pass to the method, if any
3. The type of value returned from the method, if any

As a matter of fact, these are the same three facts that another programmer needs to know to be able to use any method you write. For example, you might discover that in a particular language, the documentation indicates that the format of its square root function is `num sqrt(num)`, indicating that the function name is `sqrt`, that it requires a single numeric variable or constant as an argument, and that it returns a numeric value. A statement such as `num sqrt(num)` fully describes how you can use the method. To use the method, you might create statements similar to the following:

```
num myValue = 16
num squareRootAnswer
squareRootAnswer = sqrt(myValue)
```

You use the method name `sqrt`, pass the value stored in `myValue` to it, and receive an answer back; the answer can then be used in the same way you would use any other value of the same type—you can print it, assign it to a variable, use it as part of a more complex arithmetic calculation, and so on. After the three preceding statements execute, for example, the variable `squareRootAnswer` would hold 4, the square root of 16.

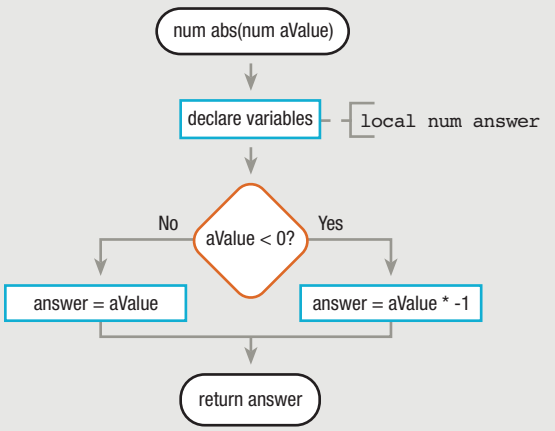
TIP In some languages, such as C++ and Java, a function's name and list of parameters are called its **signature**. When the signature is combined with the method's return type (and other punctuation), it is called the method's **prototype**.

As a programmer who uses a built-in method or function, you do not need to understand how the square root is calculated within the function; the method acts as a **black box**, or a device you can use without understanding its internal processes. In real life, you use many black box items—for example, most of us can use a telephone very well without having any idea how our voice is transported to friends and relatives around the world. Similarly, you can operate a television set without any knowledge of how the images appear there. What you understand about black box devices is their interface—the means of interaction, such as the buttons and speakers. Well-written program methods have the same features—the user understands how to use them through the interface (the signature) but does not need to understand their internal workings.

Consider the three `abs()` functions shown in Figure 12-13. Each version of the `abs()` method returns the absolute value of a number. A number's **absolute value** is the positive value of the number; as examples, the absolute value of –5 is 5, and the absolute value of 17 is 17. In other words, taking the absolute value of a number removes the negative sign, if the number has one. The set of statements within each of the three modules is different, yet a programmer would use each of the modules in an identical fashion. To store the absolute value of `myNumber` in a variable named `answer`, you write `answer = abs(myNumber)`. Just as you do not really care whether your television

operates using household AC current or a battery, or is powered by a hamster on a wheel, as a programmer using a black box method, you don't really care which of the three versions of the `abs()` function shown in Figure 12-13 exists in the programming language you are using. Like the built-in methods, the internal operations of the methods you write should be invisible to the user.

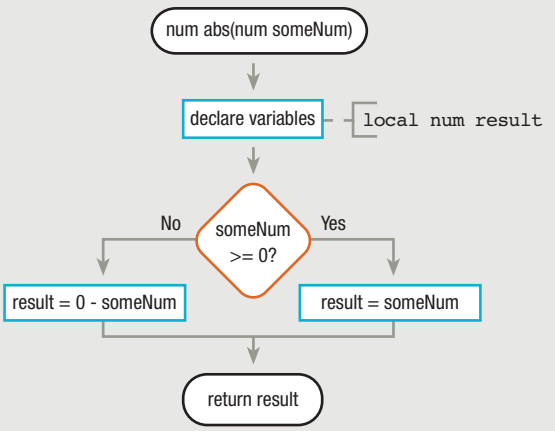
FIGURE 12-13: THREE VERSIONS OF AN `abs()` FUNCTION

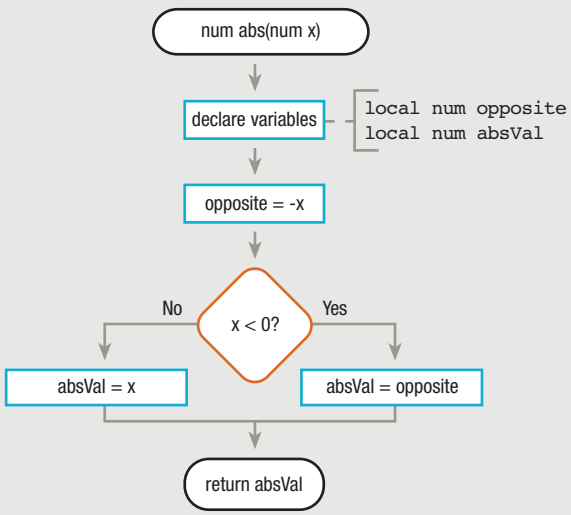



```
num abs(num aValue)
     declare variables
     if aValue < 0 then
          answer = aValue * -1
     else
          answer = aValue
return answer

num abs(num someNum)
     declare variables
     if someNum >= 0 then
          result = someNum
     else
          result = 0 - someNum
return result

num abs(num x)
     declare variables
     opposite = -x
     if x < 0
          absVal = opposite
     else
          absVal = x
return absVal
```





TIP In Figure 12-13, notice how the name of the numeric parameter in the function header can be different in each version of the `abs()` method. The variable's local name in no way affects how the method is called from another module; any numeric value passed into a method takes on the local name provided in the method header.

USING AN IPO CHART

When designing modules to use within larger programs, some programmers find it helpful to use an **IPO chart**, a tool that identifies and categorizes each item needed within the module as pertaining to input, processing, or output. For example, when you design the `finalStatistics()` module in the arithmetic drill program, you can start by placing each of the module's components in one of the three processing categories, as shown in Figure 12-14.

FIGURE 12-14: IPO CHART FOR THE `finalStatistics()` MODULE

Input	Processing	Output
Correct count	Divide correct count by total number of problems and multiply by 100, producing percentage correct	Percentage correct

The IPO chart in Figure 12-14 provides you with an overview of the processing steps involved in the `finalStatistics()` module. Like a flowchart or pseudocode, an IPO chart is just another tool to help you plan the logic of your programs. Many programmers create an IPO chart only for specific modules in their programs and as an alternative to flowcharting or writing pseudocode. IPO charts provide an overview of input to the module, the processing steps that must occur, and the result.

TIP This book emphasizes creating flowcharts and pseudocode. You can find many more examples of IPO charts on the Web.

UNDERSTANDING THE ADVANTAGES OF ENCAPSULATION

When writing a module that receives a variable, you can give the variable any name you like. This feature is especially beneficial if you consider that a well-written module may be used in dozens of programs, each supporting its own unique variable names. To beginning programmers, using only global variables seems like a far simpler option than declaring local variables and being required to pass them from one module to another. If a variable holds a count of correct responses, why not create a single variable, call it `correctCount`, and let every module in the program have access to the data stored there?

As an example of why this is a limiting idea, consider this: The `finalStatistics()` module of the arithmetic drill program might be useful in other programs within the organization—maybe the company creates drills in subjects other than arithmetic, but all drills require final statistics. If it is well-written, the `finalStatistics()` module can be used by other programs in the company for years to come. If the variables that `finalStatistics()` uses are

not declared to be local, then every programmer working on every application within the organization will have to know the names of those variables, to avoid conflict. If `correctCount` is global, then all programmers who use the module must be aware of the name and purpose of the variable and avoid using it in any other context.

If the `finalStatistics()` module is so useful that you sell it to other companies, and if `correctCount` is global, all programmers in those organizations will need to know its name and will have to avoid using it for any other purpose in their programs. The name `correctCount` represents just one variable. Multiply the limitations on global variable name usage by all the variable names used in the programs all over the world, and you can see that using global variable names correctly will soon become impossible. The logistics would be similar to providing a unique first name to every person at birth; you could do it, but you would end up using awkward, cryptic names.

Even if you could provide unique variable names for every program, there are other benefits to using local variables that are passed to modules. Passing values to a module helps facilitate encapsulation. A programmer can write a program (or module) and use procedures developed by others without knowing the details of those procedures; the programmer can use the modules as black boxes. You don't need to know—maybe you don't even care—how a procedure uses the data you send, as long as the results are what you want.

When procedures use local variables, the procedures become miniprograms that are relatively autonomous. Modules that contain their own sets of instructions and their own variables are not dependent on the program that calls them. The details within a module are hidden and contained, or encapsulated, which helps to make the module reusable.

Many real-world examples of encapsulation exist. When you build a house, you don't invent plumbing and heating systems. You incorporate systems that have already been designed. You don't need to know all the fine details of how the systems work; they are self-contained units you attach to your house. This certainly reduces the time and effort it takes to build a house. Assuming the plumbing and electrical systems you choose are already in use in other houses, choosing existing systems also improves your house's reliability. **Reliability** is a feature of programs or modules that have been tested and proven to work correctly. Not only is a prefabricated furnace reliable, but it is unnecessary to know how your furnace works, and if you replace one model with another, you don't care if the internal operations of the new model are different. Whether heat is created from electricity, natural gas, or solar power, only the result—a warm house—is important to you.

Similarly, software that is reusable saves time and money and is more reliable. If the `finalStatistics()` module has been tested previously, you can be confident that it will work correctly when you use it within a different program. If another programmer creates a new and improved `finalStatistics()` module, you don't care how it works, as long as it correctly calculates and prints using the data you send to it.

The concept of passing variables to modules allows programmers to create variable names locally in a module without changing the value of similarly named variables in other modules. The ability to pass values to modules makes programming much more flexible, because independently created modules can exchange information efficiently. However, there are limitations to the ways procedural programs use modules. Any procedural program that uses a module must not reuse its name for any other module within the same program. With procedural programs, you also must know exactly what type of data to pass to a module, and if you have use for a similar module that works on a different type of data or a different number of data items, you must create a new module with a different name. These limitations are eliminated in programs that are object-oriented. In Chapter 13, you will learn the principles of object-oriented programming.

REDUCING COUPLING AND INCREASING COHESION

When you begin to design computer programs, it is difficult to decide how much to put into a module or subroutine. For example, a process that requires 40 instructions can be contained in a single module, two 20-instruction modules, 20 two-instruction modules, or any other combination. In most programming languages, any of these combinations is allowed. That is, you can write a program that will execute and produce correct results no matter how you divide the individual steps into modules. However, placing either too many or too few instructions in a single module makes a program harder to follow and reduces flexibility. When deciding how to organize your program steps into modules, you should adhere to two general rules:

- Reduce coupling.
- Increase cohesion.

REDUCING COUPLING

Coupling is a measure of the strength of the connection between two program modules; it is used to express the extent to which information is exchanged by subroutines. Coupling is either tight or loose, depending on how much one module depends on information from another. **Tight coupling**, which occurs when modules excessively depend on each other, makes programs more prone to errors; there are many data paths to keep track of, many chances for bad data to pass from one module to another, and many chances for one module to alter information needed by another module. **Loose coupling** occurs when modules do not depend on others. In general, you want to reduce coupling as much as possible because connections between modules make them more difficult to write, maintain, and reuse.

Imagine four cooks wandering in and out of the kitchen while preparing a stew. If each is allowed to add seasonings at will without the knowledge of the other cooks, you could end up with a culinary disaster. Similarly, if four payroll program modules are allowed to alter your gross pay figure "at will" without the "knowledge" of the other modules, you could end up with a financial disaster. A program in which several modules have access to your gross pay figure has modules that are tightly coupled. A superior program would control access to the payroll figure by limiting its passage to modules that need it.

You can evaluate whether coupling between modules is loose or tight by looking at the intimacy between modules and the number of parameters that are passed between them.

- Tight coupling—The least intimate situation is one in which modules have access to the same globally defined variables; these modules have tight coupling. When one module changes the value stored in a variable, other modules are affected.
- Loose coupling—The most intimate way to share data is to pass a copy of needed variables from one module to another. That way, the sharing of data is always purposeful—variables must be explicitly passed to and from modules that use them. The loosest (best) subroutines and methods pass single arguments rather than many variables or entire records, if possible.

Usually, you can determine that coupling is occurring at one of several levels. **Data coupling** is the loosest type of coupling; therefore, it is the most desirable. Data coupling is also known as **simple data coupling** or **normal coupling**. Data coupling occurs when modules share a data item by passing arguments. For example, a module that determines a student's eligibility for the dean's list might receive a copy of the student's grade point average to use in making the determination.

Data-structured coupling is similar to data coupling, but an entire record is passed from one module to another. For example, consider a module that determines whether a customer applying for a loan is creditworthy. You might write a module that receives the entire customer record and uses many of its fields to determine whether the customer should be granted the loan. If you need many of the customer fields—such as salary, length of time on the job, savings account balance, and so on—then it makes sense to pass a customer's record to a module. Figure 12-15 shows an example of such a module.

FIGURE 12-15: MODULE THAT DETERMINES CUSTOMER CREDITWORTHINESS

```
char checkCredit(Record custRec)
   declare variables ----------------------------
   creditIsOk = YES_CODE                          |
   if custSalary < MIN_SALARY then                |
      creditIsOk = NO_CODE                        |
   endif                                          |
   if custTimeOnJob < MIN_TIME then               |
      creditIsOk = NO_CODE                        |
   endif                                          |
   if custSavingsBal < MIN_SAVINGS then           |
      creditIsOk = NO_CODE                        |
   endif                                          |
return creditIsOk                                 |
                                                  |
local char creditIsOk                             |
local const char YES_CODE = "Y"                   |
local const char NO_CODE  = "N"                   |
local const num MIN_SALARY = 20000.00             |
local const num MIN_TIME = 2                      |
local const num MIN_SAVINGS = 3000.00             |
```

In the `checkCredit()` module in Figure 12-15, an entire record (`custRec`), rather than any single data field, is passed to the module. The coupling could have been made looser by writing three separate modules: one to check salary, one to check time on the job, and one to check savings balance. However, because so many fields in the customer's record are needed, in this case it is very appropriate to pass the entire record to the module.

Control coupling occurs when a main program (or other module) passes an argument to a module, controlling the module's actions or telling it what to do. For example, Figure 12-16 shows a module that receives a user's choice and calls one of several other modules.

FIGURE 12-16: THE `selectMethod()` MODULE

```
selectMethod(num userChoice)
     if userChoice = 1 then
          perform addRecordToFile()
     else
          if userChoice = 2 then
               perform deleteRecordFromFile()
          else
               if userChoice = 3 then
                    perform printRecords()
               else
                    perform invalidChoice()
               endif
          endif
     endif
return
```

Although control coupling is appropriate at times, the implication in the `selectMethod()` module is that any module that calls it is aware of how `selectMethod()` works—after all, an appropriate choice had to be made and passed to `selectMethod()`. The program that uses `selectMethod()` probably prompts the user for a choice and passes that choice to `selectMethod()`. Therefore, the calling program must know how to phrase the prompt correctly to elicit an appropriate `userChoice`. This coupling is relatively tight. This is a problem, because if you make a change to the `selectMethod()` module—for example, by adding a new option or changing the order of the existing options—then all the programs and other modules that use `selectMethod()` will have to know about the change. If they don't, their prompts will offer incorrect choices, and they won't be sending the appropriate `userChoice` value to the module. Once you have to start keeping track of all the modules and programs that might call a module, the opportunity for errors in a system increases dramatically.

External coupling and **common coupling** occur, respectively, when two or more modules access the same global variable or record. When data can be modified by more than one module, programs become harder to write, read, and modify. That's because if you make a change in a single module, many other modules can be affected. For example, if one module increases a field that holds the year from two digits to four, then all other modules that use the year will have to be altered before they can operate correctly. For another example, if one module can increase your gross pay figure by 10 percent based on years of service, and another module can increase your pay by 20 percent based on annual sales, it makes a difference which module operates first. It's possible that a third module won't work when the salary increases over a specified limit. If you avoid external or common coupling and pass variables instead, you can control how and when the modules receive the data.

Pathological coupling occurs when two or more modules change one another's data. An especially confusing case occurs when `moduleOne()` changes data in `moduleTwo()`, `moduleTwo()` changes data in `moduleThree()`, and `moduleThree()` changes data in `moduleOne()`. This makes programs extremely difficult to follow, and you should avoid pathological coupling at all costs.

INCREASING COHESION

Analyzing coupling lets you see how modules connect externally with other modules and programs. You also want to analyze a module's **cohesion**, which refers to how the internal statements of a module or subroutine serve to accomplish the module's purposes. In highly cohesive modules, all the operations are related, or "go together." Such modules are usually more reliable than those that have low cohesion; they are considered stronger, and they make programs easier to write, read, and maintain.

Functional cohesion occurs when all operations in a module contribute to the performance of only one task. Functional cohesion is the highest level of cohesion; you should strive for it in all methods you write. For example, a module that calculates gross pay appears in Figure 12-17. The module receives two parameters, `hours` and `rate`, and computes gross pay, including time-and-a-half for overtime. The functional cohesion of this module is high because each of its instructions contributes to one task—computing gross pay. If you can write a sentence describing what a module does, using only two words—for example, "Compute gross," "Cube value," or "Print record"—the module is probably functionally cohesive.

FIGURE 12-17: THE `computeGrossPay()` MODULE

```
num computeGrossPay(num hours, num rate)
      declare variables - - - - - - - - - - - - - - - - local num gross
                                                        local const num WORK_WEEK = 40
      if hours <= WORK_WEEK then
         gross = hours * rate
      else
         gross = (WORK_WEEK * rate) + (hours - WORK_WEEK) * (rate * 1.5)
      endif
return gross
```

You might work in a programming environment that has a rule such as "No module will be longer than can be printed on one page" or "No module will have more than 30 lines of code." The rule maker is trying to achieve more cohesion, but this is an arbitrary way of going about it. It's possible for a two-line module to have low cohesion and—although less likely—for a 40-line module to have high cohesion. Because good, functionally cohesive modules perform only one task, they tend to be short. However, the issue is not size. If it takes 20 statements to perform one task within a module, then the module is still cohesive.

Two types of cohesion are considered inferior to functional cohesion, but still acceptable. **Sequential cohesion** takes place when a module performs operations that must be carried out in a specific order on the same data. Sequential cohesion is a slightly weaker type of cohesion than functional cohesion—even though the module might perform a variety of tasks, the tasks are linked because they use the same data, often transforming the data in a series of steps. **Communicational cohesion** occurs in modules that perform tasks that share data. The tasks are not related; only the data items are. If the tasks must be performed in order, the module is sequentially cohesive. If the tasks are not performed in any sequential order but only share the same data, the module is communicationally cohesive; this is considered a weaker form of cohesion than functional or sequential cohesion.

Modules with several other types of cohesion are considered generally inferior to modules that are functionally, sequentially, or communicationally cohesive, but there still are occasions when they can be used appropriately. **Temporal cohesion** takes place when the tasks in a module are related by time. That is, the tasks are placed together because of when they must take place—for example, at the beginning of a program. The prime examples of temporally cohesive modules you have seen are `housekeeping()` and `finishUp()` modules. **Procedural cohesion** takes place when, as with sequential cohesion, the tasks of a module are performed in sequence. However, unlike operations in sequentially cohesive methods, the tasks in procedurally cohesive methods do not share data. Main program modules are often procedurally cohesive; they consist of a series of steps that must be performed in sequence, but perform very different tasks, such as `housekeeping()`, `mainLoop()`, and `finishUp()`. A mainline logic module can also be called a **dispatcher module**, because it dispatches messages to a sequence of more cohesive modules. Unless you are examining your main module, if you sense that a module you have written has only procedural cohesion (that is, it consists of a series of steps that use unrelated data), you probably want to turn it into a dispatcher module. You accomplish this by changing the module so that, instead of performing many different types of tasks, it calls other modules in which the diverse tasks take place. Each of the new modules can be functionally, sequentially, or communicationally cohesive. **Logical cohesion** takes place when a member module performs one or more tasks depending on

a decision, whether the decision is in the form of a `case` structure or a series of `if` statements. The actions performed might go together logically (that is, perform the same type of action), but they don't work on the same data. Like a module that has procedural cohesion, a module that has only logical cohesion should probably be turned into a dispatcher.

One type of cohesion is generally considered to be inferior. **Coincidental cohesion**, as the name implies, is based on coincidence—that is, the operations in a module just happen to have been placed together. Obviously, this is the weakest form of cohesion and is not desirable. However, if you modify programs written by others, you might see examples of coincidental cohesion. Perhaps the program designer did not plan well, or perhaps an originally well-designed program was modified to reduce the number of modules, and now a number of unrelated statements are grouped in a single module.

TIP Coincidental cohesion is almost an oxymoron—cohesion that is simply coincidental is really not cohesion at all.

Most programmers do not think about the names of these cohesion types on a day-to-day basis. In other words, they do not tend to say, "My, this program is temporally cohesive." Rather, they develop a "feel" for what types of tasks rightfully belong together, and for which subsets of tasks should be diverted to their own modules.

Additionally, there is a time and a place for shortcuts. If you need a result from spreadsheet data in a hurry, you can type two values and take a sum rather than creating a formula with proper cell references. If a memo must go out in five minutes, you don't have to change fonts or add clip art with your word processor. Similarly, if you need a quick programming result, you might very well use cryptic variable names, tight coupling, and coincidental cohesion. When you create a professional application, however, you should keep professional guidelines in mind.

CHAPTER SUMMARY

- A procedural program consists of a series of steps or procedures that take place one after the other. Breaking programs into reasonable units called modules, subroutines, functions, or methods provides abstraction, allows multiple programmers to work on a problem, allows you to reuse your work, and allows you to identify structures more easily. By using local rather than global variables, you can take advantage of encapsulation, creating modules without knowing the data names used by other programmers.
- When multiple modules need access to the same variable value, you can pass a variable to the module. The passed variable is called a parameter and usually is named within the module header.
- You can pass multiple values to a module. When you do so, you must observe the order and data types of the arguments.
- Just as you can pass a value into a module, you can pass back a value from a called module to a calling module.
- Many programming languages provide built-in methods, which are preprogrammed modules you can use to perform common tasks.
- When designing modules to use within larger programs, some programmers find it helpful to use an IPO chart, which identifies and categorizes each item needed within the module as pertaining to input, processing, or output.
- The concept of passing variables to modules allows programmers to create variable names locally in a module without changing the value of similarly named variables in other modules. Passing values to a module helps facilitate encapsulation; you need to understand only the interface to the procedure. In addition, passing variables helps to make modules reusable and improves their reliability.
- When writing modules, you should strive to achieve loose coupling and high cohesion.

KEY TERMS

A **procedural program** consists of a series of steps or procedures that take place one after another.

Modularization is the process of breaking down programs into reasonable units called modules, subroutines, functions, or methods.

Abstraction is the process of ignoring minor details, making it easier to see the "big picture."

A **global variable** is one that is available to every module in a program.

A **local variable** is one whose name and value are known only to its own module.

A local variable is **in scope**—that is, existing and usable—from the moment it is declared until it ceases to exist.

A variable that is **out of scope** has ceased to exist.

Encapsulation means that program components are bundled together.

Information hiding, or **data hiding**, means that the data or variables you use are completely contained within—and accessible only to—the module in which they are declared.

Passing a value means that you are sending a copy of data in one module of a program to another module for use.

A **module header** is the introductory title statement of a module.

An **argument** is the expression in the comma-separated list in a function call.

A **parameter** is an object or reference that is declared in a function prototype (declaration) or definition (header).

A **parameter list** is the series of parameters, or passed values, that appears in a module header.

A module might **return a value** to a module that calls it, passing back a copy of the value.

A **method's type** or **method's return type** is the data type of the value it returns.

A method that returns no value is a **void** method. The word "void" means "empty" or "nothing."

Unreachable or **dead code** is any set of program statements that will never execute—for example, those statements within a module that follow the `return` statement.

Built-in methods, or **built-in functions**, are prewritten modules that perform frequently needed tasks.

A method's **signature** includes its name and parameter list. In some languages, a signature, along with the return type (and other punctuation), is also called a **prototype**.

A **black box** is a device you can use without understanding its internal processes.

A number's **absolute value** is the positive value of the number.

An **IPO chart** is a tool that identifies and categorizes each item needed within a module as pertaining to input, processing, or output.

Reliability is a feature of modules or programs that have been tested and proven to work correctly.

Coupling is a measure of the strength of the connection between two program modules.

Tight coupling occurs when modules excessively depend on each other; it makes programs more prone to errors.

Loose coupling occurs when modules do not depend on others.

Data coupling is the loosest type of coupling; therefore, it is the most desirable. Data coupling is also known as **simple data coupling** or **normal coupling**. Data coupling occurs when modules share a data item by passing parameters.

Data-structured coupling is similar to data coupling, but an entire record is passed from one module to another.

Control coupling occurs when a main program (or other module) passes an argument to a module, controlling the module's actions or telling it what to do.

External coupling and **common coupling** occur, respectively, when two or more modules access the same global variable or record.

Pathological coupling occurs when two or more modules change one another's data.

Cohesion is a measure of how the internal statements of a module or subroutine serve to accomplish the module's purposes.

Functional cohesion occurs when all operations in a module contribute to the performance of only one task. Functional cohesion is the highest level of cohesion; you should strive for it in all methods you write.

Sequential cohesion takes place when a module performs operations that must be carried out in a specific order on the same data.

Communicational cohesion occurs in modules that perform tasks that share data. The tasks are not related, just the data items.

Temporal cohesion takes place when the tasks in a module are related by time.

Procedural cohesion takes place when, as with sequential cohesion, the tasks of a module are performed in sequence. However, unlike operations in sequential cohesion, the tasks in procedural cohesion do not share data.

A **dispatcher module** dispatches messages to a sequence of more cohesive modules.

Logical cohesion takes place when a member module performs one or more tasks depending on a decision. The actions performed might go together logically (that is, perform the same type of action), but they don't work on the same data.

Coincidental cohesion is based on coincidence—that is, the operations in a module just happen to have been placed together.

REVIEW QUESTIONS

1. **Which of the following is not a synonym for "module"?**
   a. procedure
   b. function
   c. method
   d. program

2. **Which of the following is not a benefit of modularization?**
   a. Modularization provides abstraction.
   b. Modularization allows multiple programmers to work on a problem.
   c. Modularization ensures the elimination of logical errors.
   d. Modularization allows programmers to reuse their work more easily.

3. **A variable that is available to every module in a program is a(n) _____ variable.**
   a. global
   b. international
   c. local
   d. neighborhood

4. **A local variable is usable when it is _____.**
   a. in view
   b. in scope
   c. in range
   d. limitless

5. **When variables located in a module are hidden from other modules, the program is using _____.**
   a. encapsulation
   b. secret coding
   c. condensation
   d. functional composition

6. **When you pass a value to a module, within the module, the passed variable _____ as the original.**
   a. has the same memory address
   b. has the same name
   c. has the same value
   d. all of the above

7. **The introductory title statement of a module is its _____.**
   a. banner
   b. label
   c. header
   d. caption

8. **A variable passed to a module is a(n) _____.**
   a. argument
   b. quarrel
   c. claim
   d. return type

9. **The series of parameters received by a module is the module's _____.**
   a. return type
   b. directory
   c. sequence
   d. parameter list

10. **Assume you have written a module with the following header: `myModule(char name, num age)`. Which of the following module calls is correct?**
    a. `myModule("Joan", 32)`
    b. `myModule(19, "Sean")`
    c. Both of these are correct.
    d. Neither a nor b is correct.

11. Assume you have written a module with the following header: `anotherModule(char name, num age, num salary)`. Which of the following module calls is correct?

a. `anotherModule("Jerry", 32)`
b. `anotherModule("Elaine", 39, 20000)`
c. both of the above
d. neither a nor b

12. A module that sends a value back to a module that calls it _____ the value.

a. exports
b. imports
c. returns
d. delivers

13. The return type of a method is also called the _____.

a. exact value
b. method's type
c. secondary type
d. parameter list

14. Built-in functions are _____.

a. methods without a return type
b. methods without parameter lists
c. prewritten, automatically available methods
d. customized methods used by a particular type of business or industry

15. To use a method written by another programmer, you must know all of the following except _____.

a. the name of the method
b. the types of arguments passed to the method
c. the number of statements within the method
d. the return type of the method

16. To programmers, a black box is a module that _____.

a. you use without knowing the arguments or return type
b. is built into a programming language
c. you use without knowing how it works internally
d. records instructions as they are executed

17. A tool that identifies input, processing, and output steps for a program or module is a(n) _____.

a. IPO chart
b. hierarchy chart
c. flowchart
d. debugger

18. Programmers should strive to _____.

a. increase coupling
b. increase cohesion
c. both of the above
d. neither a nor b

19. When several modules have access to the same variables and the ability to alter them, the modules are _____ coupled.

a. loosely
b. tightly
c. pathologically
d. morbidly

20. The most desirable level of cohesion is _____ cohesion.

a. coincidental
b. temporal
c. procedural
d. functional

FIND THE BUGS

The following pseudocode contains one or more bugs that you must find and correct.

1. The main program calls a method that prompts the user for an initial and returns it to the main module.

```
start
     declare variables
       char usersInitial
       askUserForInitial()
     print "Your initial is ", usersInitial
stop

char askUserForInitial()
    declare variables
       char letter
    print "Please type your initial"
    read letter
return usersInitial
```

2. **The main program passes a user's entry to a function that displays a multiplication table using the entry multiplied by every value from 2 through 10.**

```
start
   declare variables
      num usersChoice
   print "Enter a number"
   read choice
   multiplicationTable(usersChoice)
stop

multiplicationTable(num value)
   declare variables
      const num LOW = 2
      const num HIGH = 10
      num x;
   while num <= HIGH
     answer = choice * x
     print value, " times ", x, " is ", answer
     num = num + 1
   endwhile
return
```

3. **The main program prompts a user for a Social Security number, name, and income, and then computes tax. The tax calculation and the printing of the taxpayer's report are in separate modules. Tax rates are based on the following table:**

Income	Percent tax rate
0–14,999	0
15,000–21,999	15
22,000–29,999	18
30,000–44,999	22
45,000–59,999	28
60,000 and up	30

```
start
   declare variables
         num socSecNum
         char name
         num income
         num taxDue
   print "Enter socSecNum"
```

```
   read socSecNum
   while socSecNum not = 0
      print "Enter name"
      read name
      print "Enter annual income"
      read name
      taxCalculations()
      print taxReport(socSecNum, taxDue)
      print "Enter socSecNum"
      read socSecNum
   endwhile
stop

num taxCalculations(num income)
   num tax
   const num NUMBRKTS = 6
   num brackets[2] = 0, 15000, 22000, 30000,
       45000, 60000
   num rates[NUMBRKTS - 1] = 0.0, 0.15, 0.18, 0.22, 0.28, 0.30
   num count = NUMBRKTS
   while count >= 0
     if income = brackets[count]
        count = count - 1
     endif
   endwhile
   tax = income * rates[count]
return tax

taxReport(num socSecNum, num name, num taxDue)
   print socSecNum, name, taxDue
return
```

EXERCISES

1. **Create an IPO chart for each of the following modules:**
   a. The module that produces your paycheck
   b. The module that calculates your semester tuition bill
   c. The module that calculates your monthly car payment

2. **Plan the logic for a program that contains two modules. The first module prompts the user for a grade on an exam. Pass the grade to a second module that prints "Pass" if the grade is 60 percent or higher and "Fail" if it is not.**

3. **Complete the following tasks:**
   a. Plan the logic for a program that contains two modules. The first module asks for your employee ID number. Pass the ID number to a second module that prints a message indicating whether the ID number is valid or invalid. A valid employee ID number falls between 100 and 799, inclusive.
   b. Plan the logic for a program that contains two modules. The first module asks for your employee ID number. Pass the ID number to a second module that returns a code to the first module indicating whether the ID number is valid or invalid. A valid employee ID number falls between 100 and 799, inclusive. The first module prints an appropriate message.

4. **Complete the following tasks:**
   a. Plan the logic for an insurance company's premium-determining program that contains three modules. The first module prompts the user for the type of policy needed—health or auto. Pass the user's response to the second module, where the premium is set—$250 for a health policy or $175 for an auto policy. Pass the premium amount to the last module for printing.
   b. Modify Exercise 4a so that the second module calls one of two additional modules—one that determines the health premium or one that determines the auto premium. The health insurance module asks users whether they smoke; the premium is $250 for smokers and $190 for nonsmokers. The auto insurance module asks users to enter the number of traffic tickets they have received in the last three years. The premium is $175 for those with three or more tickets, $140 for those with one or two tickets, and $95 for those with no tickets. Each of these two modules returns the premium amount to the second module, which sends the premium amount to the printing module.

5. **Plan the logic for a program that reads inventory records from a file that contains the following fields: item number, item name, quantity in stock, and price each. In turn, pass each item number, quantity, and price to a module named `printDiscountInfo()`.**

   This is a prewritten module that calculates a new price for each item, taking one of 10 discount percentages, depending on the quantity of the item remaining in stock. The module's signature is `printDiscountInfo(num itemNo, num quantityInStock, num priceEach)`. You do not need to write this module—just call it to display each item's discount amount.

6. **Plan the logic for a program that prompts a user for a customer number, stock number of item being ordered, and quantity ordered.**

   If the customer number is not between 1000 and 7999, inclusive, continue to prompt until a valid customer number is entered. If the stock number of the item is not between 201 and 850, inclusive, continue to prompt for the stock number. Pass the stock number to a method that a colleague at your organization has written; the module's signature is `num getPrice(num stockNumber)`. The `getPrice()` module accepts a stock number and returns the price of the item. Multiply the price by the quantity ordered, giving the total due. Pass the customer number and the calculated price to an already written method whose signature is `printBill(num custNum, num price)`. This method determines the customer's name and address by using the customer ID number, and calculates the final bill, including tax, using the price. Organize your program using as many modules as you feel are appropriate. You do not need to write the `getPrice()` and `printBill()` modules—assume they have already been written.

7. **Plan the logic for a program that prompts a user for numeric values and continues to read them until the user enters 999.**

   Display the numeric average of the values; then, display each number and a statement of how far away it is from the average. For example, if the user enters 5, 6, and 7, the output is:

```
The average is 6
5 is 1 away from the average
6 is 0 away from the average
7 is 1 away from the average
```

   Assume that you can use a built-in absolute value function whose signature is `num abs(num value)`. The function accepts a numeric value and returns its absolute value.

8. **The Information Services Department at the Springfield Library has created modules with the following signatures:**

Signature	Description
`num getNumber(num high, num low)`	Prompts the user for a number. Continues to prompt until the number falls between designated high and low limits. Returns a valid number.
`char getCharacter()`	Prompts the user for a character string and returns the entered string.
`num lookUpISBN(char title)`	Accepts the title of a book and returns the ISBN. Returns a 0 if the book cannot be found.
`char lookUpTitle(num isbn)`	Accepts the ISBN of a book and returns a title. Returns a space character if the book cannot be found.
`char isBookAvailable(num isbn)`	Accepts an ISBN, searches the library database, and returns a "Y" or "N" indicating whether the book is currently available.

   a. Design an interactive program that does the following, using the prewritten modules wherever they are appropriate.

   - ☐ Prompt the user for and read a library card number, which must be between 1000 and 9999.
   - ☐ Prompt the user for and read a search option—1 to search for a book by ISBN, 2 to search for a book by title, and 3 to quit. Allow no other values to be entered.
   - ☐ While the user does not enter 3, prompt for an ISBN or title, based on the user's previous selection. If the user enters an ISBN, get and display the book's title and ask for confirmation—a "Y" or "N" as to whether the title is correct.

- If the user has entered a valid ISBN, or a title that matches a valid ISBN, check whether the book is available, and display an appropriate message for the user.
- The user can continue to search for books until he or she enters 3 as the search option.

b. Develop the logic for each of the modules in Exercise 8a.

DETECTIVE WORK

1. **In some programming languages, beginning programmers traditionally write their first module to perform what specific task?**
2. **What is beta testing?**

UP FOR DISCUSSION

1. **Modularized furniture comes with sections that can be assembled in a variety of configurations. What other everyday items are modularized?**
2. **As a professional programmer, you might never write an entire program. Instead, you might be asked to write specific modules that are destined to become part of a larger system. Is this appealing to you?**