10 USING MENUS AND VALIDATING INPUT

After studying Chapter 10, you should be able to:

- ☐ Understand the need for interactive, menu-driven programs
- ☐ Create a program that uses a single-level menu
- ☐ Code modules as black boxes
- ☐ Improve menu programs
- ☐ Use a case structure to manage a menu
- ☐ Create a program that uses a multilevel menu
- ☐ Validate input
- ☐ Understand types of data validation

USING INTERACTIVE PROGRAMS

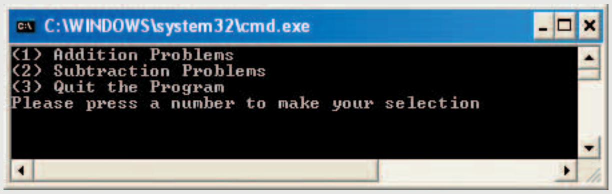
You can divide computer programs into two broad categories based on how they get their data. Programs for which all the data items are gathered prior to running use **batch processing**. Programs that depend on user input while the programs are running use **interactive processing**.

Many computer programs use batch processing with sequential files of data records that have been collected for processing. All standard billing, inventory, payroll, and similar programs work this way, and all the program logic you have developed while working through this text also works like this. Records used for batch processing are gathered over a period of time—hours, days, or even months. Programs that use batch processing typically read an input record, process it according to coded instructions, output the result, and then read another record. Batch processing gets its name because the data records are not processed at the time they are created; instead, they are "saved" and processed in a batch. For example, you do not receive a credit card bill immediately after every purchase, when the record is created. All purchases during a one-month period are gathered and processed at the end of that billing period.

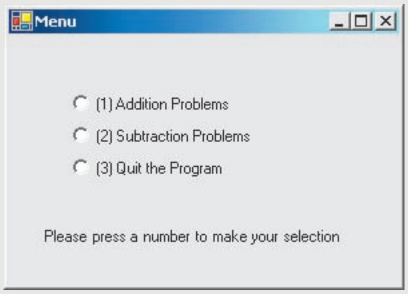
Many computer programs cannot be run in batches. Instead, they must run interactively—that is, they must interact with a user while they are running. Ticket reservation programs for airlines and theaters must select tickets while you are interacting with them, not at the end of the month. A computerized library catalog system must respond to library patrons' requests immediately, while the patrons are searching, not at the end of every week. Interactive computer programs are often called **real-time applications**, because they run while a transaction is taking place, not at some later time. You also can refer to interactive processing as **online processing**, because the user's data or requests are gathered during the execution of the program, while the computer is operating. A batch processing system can be **offline**; that is, you can collect data such as time cards or purchase information well ahead of the actual computer processing of the paychecks or bills.

A **menu program** is a common type of interactive program in which the user sees a number of options on the screen and can select any one of them. For example, an educational program that drills you on elementary arithmetic skills might display three options, as shown in the two menus in Figure 10-1. The menu on the left is used in **console applications**, those that require the user to enter a choice using the keyboard; the menu style on the right is used in **graphical user interface applications**, those that allow the user to use a mouse or other pointing device to make selections. The style you use partly depends on the programming language you choose; with languages that allow either style, the program developer decides on the format based on considerations such as the preferences of users and the amount of time available for development.

FIGURE 10-1: ARITHMETIC DRILL MENUS



```
(1) Addition Problems
(2) Subtraction Problems
(3) Quit the Program
Please press a number to make your selection
```



Please press a number to make your selection

TIP You could include a title or further instructions on the menus shown in Figure 10-1, as well as on the other menus in this chapter. They are eliminated here to keep the examples as simple as possible.

The final option in each menu in Figure 10-1, *Quit the Program*, is very important; without it, there would be no elegant way for the program to terminate. A menu without a *Quit* option is very frustrating to the user.

Some menu programs require the user to enter a number to choose a menu option. For example, the user enters a *2* to perform a subtraction drill from the first menu shown in Figure 10-1. Other menu programs require the user to enter a letter of the alphabet—for example, *S* for a subtraction drill. Still other programs allow the user to use a pointing device such as a mouse to point to a choice on the screen, as with the menu on the right side of Figure 10-1. The most sophisticated programs allow users to employ the selection method that is most convenient at the time.

TIP Many organizations provide an audio menu to callers to handle routing of telephone calls. If you have ever called an organization and heard a message like "Press 1 for the Sales Department," then you have used an interactive menu.

USING A SINGLE-LEVEL MENU

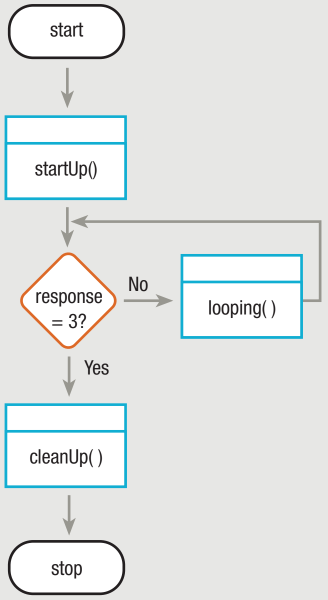
Suppose you want to write a program that displays a menu like the one shown in Figure 10-1. The program drills a student's arithmetic skills—if the student chooses the first option, four addition problems are displayed, and if the student chooses the second option, four subtraction problems are displayed. This program uses a **single-level menu**; that is, the user makes a selection from only one menu before using the program for its ultimate purpose—arithmetic practice. With more complicated programs, a user's choice from an initial menu often leads to other menus from which the user must make several selections before reaching the desired destination.

Suppose you want to write a program that requires the user to enter a digit to make a menu choice. The mainline logic for an interactive menu program is not substantially different from any of the other sequential file programs you've seen so far in this book. You can create `startUp()`, `looping()`, and `cleanUp()` modules, as shown in Figure 10-2.

The only difference between the mainline logic in Figure 10-2 and that of other programs you have worked with lies in the main loop control question. When a program's input data comes from a data file, asking whether the input file is at the end-of-file (`eof`) condition is appropriate. An interactive, menu-driven program is not controlled by an end-of-file condition, but by a user's menu response. The mainline logic, then, is more appropriately controlled by the user's response. For example, Figure 10-2 shows the mainline logic containing the question `response = 3?`.

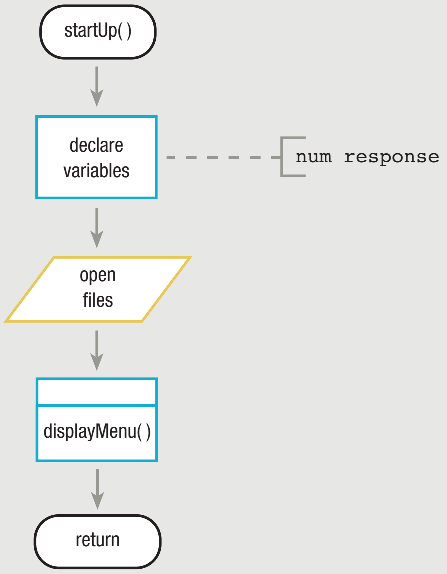
The `startUp()` module in the arithmetic drill program defines variables and opens files. The name of one of the variables is `response`; this is the numeric variable that will hold the user's menu choice. The `startUp()` module also displays the menu for the first time, so that the user can make a choice. See Figure 10-3.

FIGURE 10-2: MAINLINE LOGIC FOR THE ARITHMETIC DRILL MENU PROGRAM



```
start
     perform startUp()
     while response not = 3
          perform looping()
     endwhile
     perform cleanUp()
stop
```

FIGURE 10-3: THE `startUp()` MODULE FOR THE ARITHMETIC DRILL PROGRAM

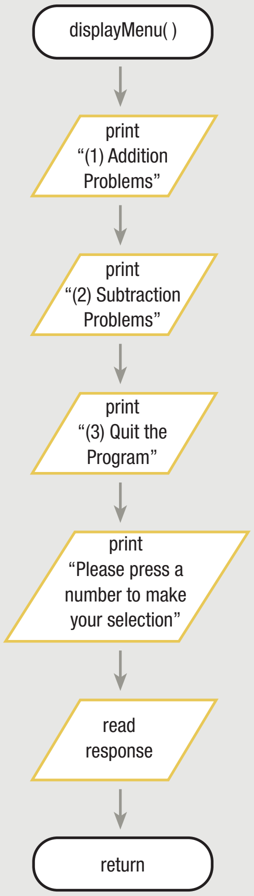


```
startUp()
     declare variables
     open files
     perform displayMenu()
return
```

TIP In many programming languages, if the keyboard is the default input device and the monitor is the default output device for an application, an explicit `open files` statement is frequently not needed.

You can include the set of instructions that displays the user menu directly in the `startUp()` module, or, as shown here, you can place the instructions in their own module. For example, Figure 10-4 shows the `displayMenu()` module that the `startUp()` module in Figure 10-3 calls. The `displayMenu()` module writes four menu lines on the screen, and then a `read response` statement reads the user's numeric choice from the keyboard.

FIGURE 10-4: THE `displayMenu()` MODULE FOR THE ARITHMETIC DRILL PROGRAM


displayMenu( )
print "(1) Addition Problems"
print "(2) Subtraction Problems"
print "(3) Quit the Program"
print "Please press a number to make your selection"
read response
return

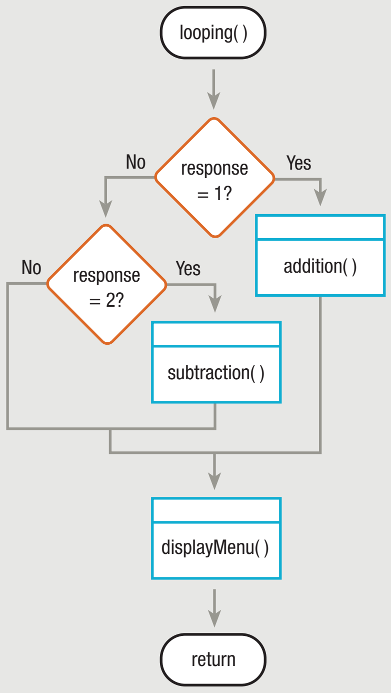
```
displayMenu()
    print "(1) Addition Problems"
    print "(2) Subtraction Problems"
    print "(3) Quit the Program"
    print "Please press a number to make your selection"
    read response
return
```

TIP □□□□ You might choose to add a command to clear the screen before printing any of the menu options. The precise syntax of the command differs from programming language to programming language. When you clear the screen, all previous messages and responses are removed, thus providing a cleaner look to the screen. Often, you clear a screen in the same circumstances when you start a new page in a printed report—at the beginning of the program or after a specified number of lines of output have been displayed.

By the time the logic of the arithmetic drill program leaves the `startUp()` module, the user has entered a value for `response`. In the mainline logic (Figure 10-2), if `response` is not *3* (for the *Quit the Program* option), then the program enters the `looping()` module. The `looping()` module makes decisions about the user's input, and either performs one of two submodules, `addition()` or `subtraction()`; or, if the user has entered a number other than 1, 2, or 3, the module performs no submodule. Following the performance of the chosen arithmetic drill, the program calls the `displayMenu()` module again, and the user has the opportunity to select the same arithmetic drill, a different one, or the *Quit the Program* option. See Figure 10-5.

TIP In the `looping()` module in Figure 10-5, a user who has entered a value such as 4 or 5 receives no explanation, but is shown the menu again. You will improve this module later in this chapter.

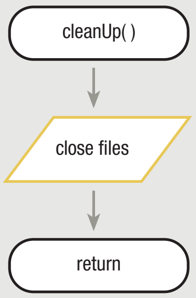
FIGURE 10-5: THE `looping()` MODULE FOR THE ARITHMETIC DRILL PROGRAM



```
looping()
     if response = 1 then
          perform addition()
     else
          if response = 2 then
               perform subtraction()
          endif
     endif
     perform displayMenu()
return
```

When the `looping()` module ends, control passes to the main program. If the user has entered a value of *3* to select the *Quit the Program* option during a `displayMenu()` module, the outcome of the question `response = 3?` sends the program to the `cleanUp()` module. That module simply closes the files, as shown in Figure 10-6.

FIGURE 10-6: THE `cleanUp()` MODULE FOR THE ARITHMETIC DRILL PROGRAM



```
cleanUp()
     close files
return
```

CODING MODULES AS BLACK BOXES

Any steps you want can occur within the `addition()` and `subtraction()` modules in the arithmetic drill program. The contents of these modules should not affect the main structure of the program in any way. You can write an `addition()` module that requires the user to solve simple addition problems, such as `3 + 4`, or you can write an `addition()` module that requires the user to solve more difficult, multidigit problems, such as `9267 + 3488`. You can write the module to contain a single problem for the user to solve, or dozens. As you will recall from Chapter 2, part of the advantage of modular, structured programs lies in your ability to break programs into modules that can be assigned to any number of programmers and then pieced back together at each module's single entry or exit point. Thus, any number of `addition()` or `subtraction()` modules can be used within the arithmetic drill program, and a new one can be substituted at any time.

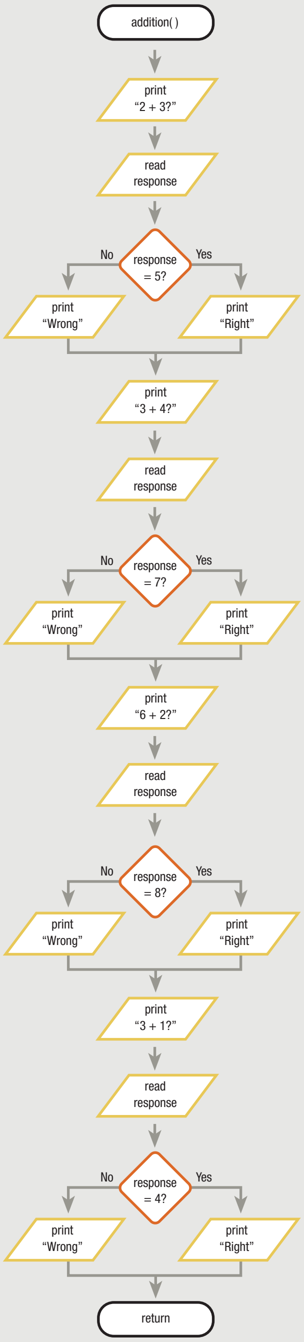
Programmers often refer to the code in modules such as `addition()` and `subtraction()` as existing within a **black box**, meaning that the module statements are encapsulated in a container that makes them "invisible" to the rest of the program. You probably own many real-life objects that are black boxes to you—a television or a stereo, for example. You might not know how these devices work internally, and if someone substituted new internal mechanisms in your devices, you might not know or care, so long as the devices continued to work properly. Similarly, many different `addition()` or `subtraction()` modules could be "plugged into" the arithmetic drill menu program and it would continue to function appropriately.

When first developing a program, programmers frequently don't bother with module details at all, because many versions of a module can substitute for one another. Instead, programmers concentrate on the mainline logic and on understanding what the called modules will do, not on how they will do it. When programmers develop systems containing many modules, they often code "empty" black box procedures, called **stubs**. That way, they can develop the overall project logic without worrying about the minor details. Later, they can code the details in the stub modules.

Figure 10-7 shows a possible `addition()` module. The module displays four addition problems one at a time, waits for the user's response, and displays a message indicating whether the user is correct.

TIP □□□□ You can write a `subtraction()` module using a format that is almost identical to the `addition()` module. The only necessary change is the computation operation used in the actual problems.

FIGURE 10-7: THE `addition()` MODULE, VERSION 1



```
addition()
     print "2 + 3?"
     read response
     if response = 5 then
          print "Right"
     else
          print "Wrong"
     endif
     print "3 + 4?"
     read response
     if response = 7 then
          print "Right"
     else
          print "Wrong"
     endif
     print "6 + 2?"
     read response
     if response = 8 then
          print "Right"
     else
          print "Wrong"
     endif
     print "3 + 1?"
     read response
     if response = 4 then
          print "Right"
     else
          print "Wrong"
     endif
return
```

The addition() module shown in Figure 10-7 works, but it is repetitious; a basic set of statements repeats four times, changing only the actual problem values that the user should add, and the correct answer to which the user's response is compared. A more elegant solution involves storing the problem values in arrays and using a loop. For example, if you declare two arrays, as shown in Figure 10-8, then the loop in Figure 10-9 displays and checks four problems. The power of using an array allows you to alter a subscript in order to display four separate addition problems.

FIGURE 10-8: ARRAYS FOR ADDITION PROBLEMS

```
num probValFirst[0] = 2
num probValFirst[1] = 3
num probValFirst[2] = 6
num probValFirst[3] = 3

num probValSecond[0] = 3
num probValSecond[1] = 4
num probValSecond[2] = 2
num probValSecond[3] = 1
```

TIP To use the addition() module shown in Figure 10-9, besides the problem value arrays, you also have to declare the numeric variable count.

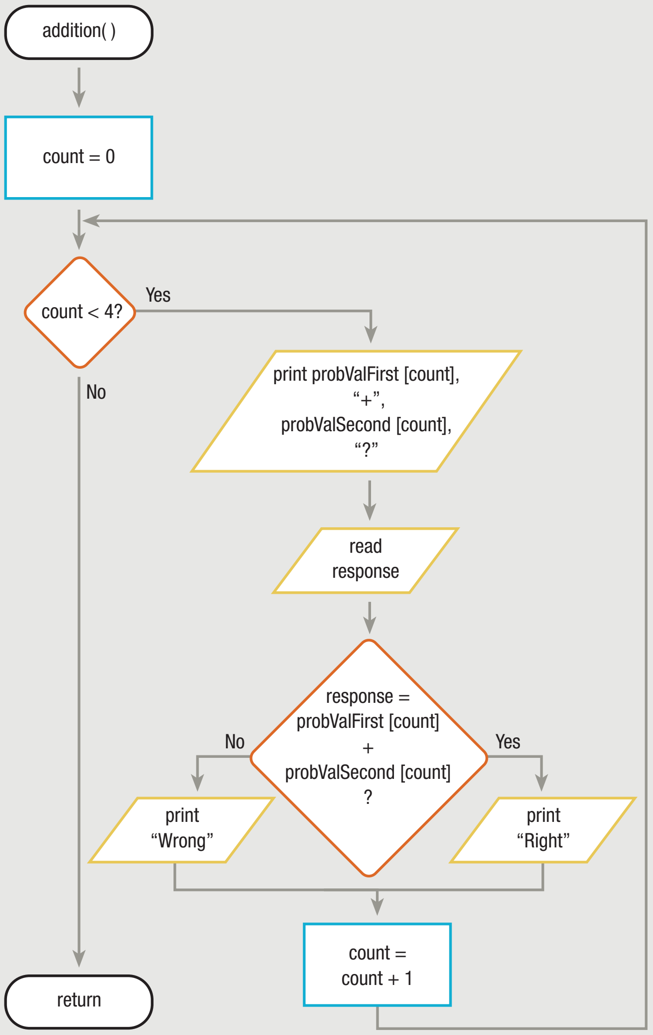
In Figure 10-9, the addition() module sets a count variable to 0. Then, because count remains less than 4, a problem is displayed for the student. The problem display is constructed in four parts—the first probValFirst element, a plus sign, the first probValSecond element, and a question mark. The module reads the user's answer and compares it to the calculated sum of the two operands in the addition problem, printing either "Right" or "Wrong".

Calculating the correct answer is an improvement over the original version of the program for two reasons. First, a hard-coded answer might be typed incorrectly by the programmer, whereas a calculated answer will always be correct. Second, if the programmer decides to alter the values used in the arithmetic problem, the calculated answer will be recomputed automatically.

After the user receives feedback on the arithmetic problem, count is increased, and if it remains in range, the arithmetic drill proceeds with the next addition problem.

The addition() module in Figure 10-9 is more compact and efficient than the module shown in Figure 10-7. However, it still contains flaws. A student will not want to use the addition() module more than two or three times. Every time a user executes the program, the same four addition problems are displayed. Once students have solved all the addition problems, they probably will be able to provide memorized answers without practicing arithmetic skills at all. Fortunately, most programming languages provide you with built-in modules, or **functions**, that automatically provide a mathematical value such as a square root, absolute value, or random number. Functions that generate a random number usually take a form similar to random(x), where x is a value you provide for the maximum random number you want. Different computer systems use different formulas for generating a random number; for example, many use part of the current clock time when the random number function is called. However, a programming language's built-in functions can operate as black boxes, just as your program modules do, so you need not know exactly how the functions do their jobs. You can use the random number function without knowing how it determines the specific random number.

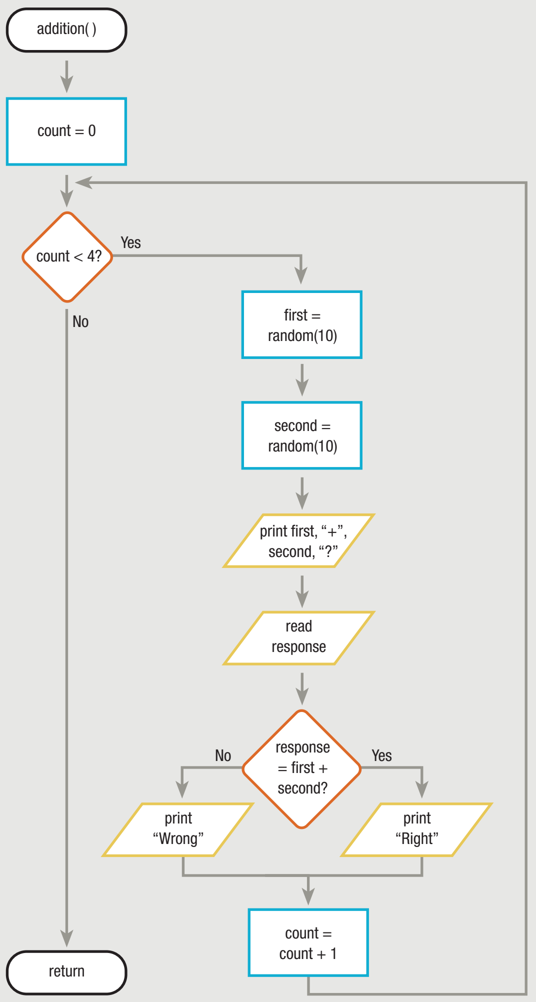
FIGURE 10-9: THE `addition()` MODULE, VERSION 2, USING ARRAYS


addition( )
count = 0
count < 4?
Yes
No
print probValFirst [count], "+", probValSecond [count], "?"
read response
response = probValFirst [count] + probValSecond [count] ?
No
Yes
print "Wrong"
print "Right"
count = count + 1
return

```
addition()
     count = 0
     while count < 4
          print probValFirst[count], "+", probValSecond[count], "?"
          read response
          if response = probValFirst[count] + probValSecond[count] then
               print "Right"
          else
               print "Wrong"
          endif
          count = count + 1
     endwhile
return
```

Figure 10-10 shows an `addition()` module in which two random numbers, each 10 or less, are generated for each of four arithmetic problems. Using this technique, you do not have to store values in an array, and users encounter different addition problems every time they use the program.

FIGURE 10-10: THE `addition()` MODULE, VERSION 3, USING RANDOM VALUES



```
addition()
     count = 0
     while count < 4
          first = random(10)
          second = random(10)
          print first, "+", second, "?"
          read response
          if response = first + second then
               print "Right"
          else
               print "Wrong"
          endif
          count = count + 1
     endwhile
return
```

TIP The module in Figure 10-10 would require two new variable declarations in the `startUp()` module in Figure 10-3: `num first` and `num second`.

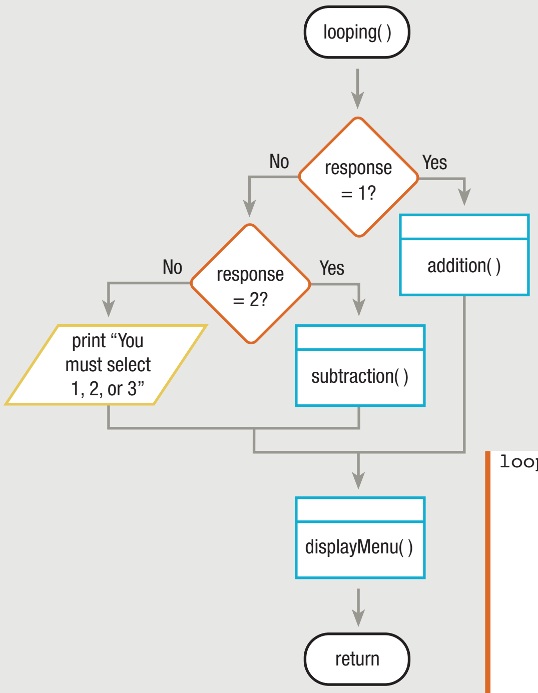
TIP Popular spreadsheet programs also contain functions. As in programming, they are built-in modules that return requested values such as square root or absolute value. Most spreadsheets also contain dozens of specialized functions to support financial applications, such as computing the future value of an investment and calculating a loan payment.

You can make many additional improvements to any of the `addition()` modules shown in Figures 10-7, 10-9, and 10-10. For example, you might want to give the user several chances to calculate the correct answer, or you might want to vary the messages displayed in response to correct and incorrect answers. However you change the `addition()` or `subtraction()` modules in the future, the main structure of the menu program does not have to change; modularization has made your program easily modifiable to meet changing needs and user preferences.

MAKING IMPROVEMENTS TO A MENU PROGRAM

When the menu appears at the end of the `looping()` module of the arithmetic drill program, if the user selects anything other than 3, the `looping()` module is entered again. Note that if the user chooses *4* or *9* or any other invalid menu item, the menu simply reappears. Unfortunately, the repeated display of the menu can confuse the user. Perhaps the user is familiar with another program in which option *9* has always meant *Quit*. When using the arithmetic drill program, the user who does not read the menu carefully might press *9*, get the menu back, press *9*, and get the menu back again. The programmer can assist the user by displaying a message when the selected `response` value is not one of the allowable menu options, as shown in Figure 10-11.

FIGURE 10-11: ADDING AN ERROR MESSAGE TO THE `looping()` MODULE OF THE ARITHMETIC DRILL PROGRAM



```
looping()
     if response = 1 then
          perform addition()
     else
          if response = 2 then
               perform subtraction()
          else
               print "You must select 1, 2, or 3"
          endif
     endif
     perform displayMenu()
return
```

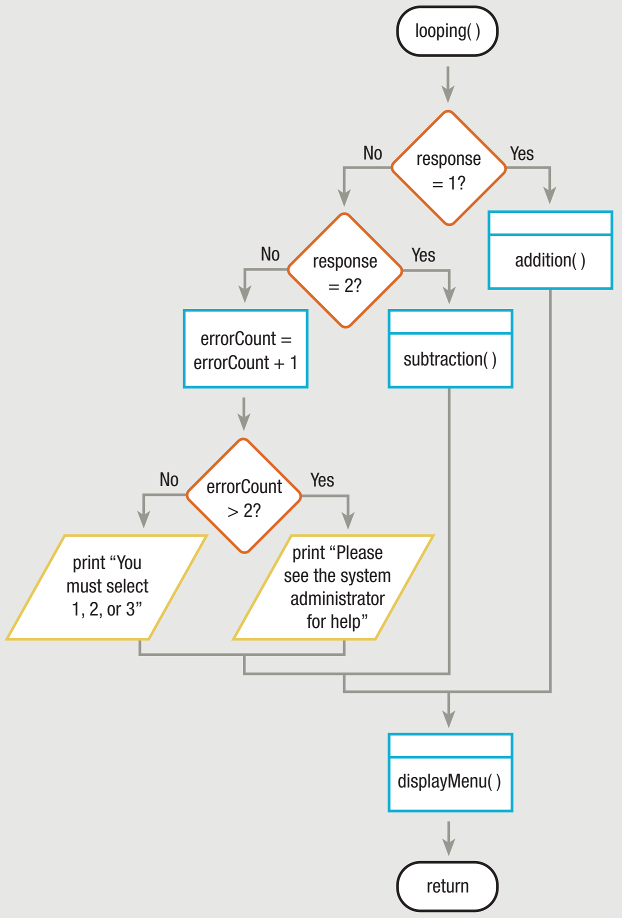
TIP

When you code the displayMenu() module in a programming language, you might choose to write it so that it clears the screen of all old output before showing the menu options. If so, then in the looping() module in Figure 10-11, you would be required to place a statement that pauses the program—for example, requiring the user to press a key or using a built-in function available in many languages that waits for a number of specified seconds before continuing. Without such a pause, the message "You must select 1, 2, or 3" would be displayed on the screen, then be replaced by the menu almost instantaneously, denying the user enough time to read the message.

Among programmers, there is a saying that no program is ever really completed. You always can continue to make improvements. For example, the looping() module in Figure 10-11 shows that a helpful message ("You must select 1, 2, or 3") appears when the user selects an inappropriate option. However, if users do not understand the message, or simply do not stop to read the message, they might keep entering invalid data. As a user-friendly improvement to your program, you can add a counter that keeps track of a user's invalid responses. For example, you can decide that after three invalid entries, you will issue a stronger message, such as "Please see the system administrator for help." Figure 10-12 shows this logic. Of course, to use this module, you must remember to declare errorCount in your variable list in the startUp() module, and initialize it to 0. Then, each time the user chooses an invalid response and you display the message "You must select 1, 2, or 3", you can add 1 to errorCount. When errorCount exceeds 2, you display the stronger message.

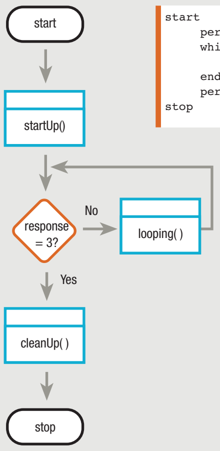
You can make an additional improvement to the looping() module in Figure 10-12. Suppose the user starts the program and enters a *5*. The value of response is not 1, 2, or 3, so you add 1 to errorCount, display the message "You must select 1, 2, or 3", and display the menu. Suppose the user enters a *5* again. Once again the response is not 1, 2, or 3, so you add 1 to errorCount, which is now 2, display the message "You must select 1, 2, or 3", and display the menu. If the user enters a *5* again, errorCount exceeds 2 and the user receives the message "Please see the system administrator for help." Assume the user gets help and figures out that he or she must type 1, 2, or 3. The user then might successfully use the program for several more minutes. However, the next time the user makes a selection error, errorCount will increase to 4 and the stronger "system administrator" message appears immediately, even though this is only the user's first "new" mistake. If you want to give the user three more chances before the stronger message appears again, then you should reset errorCount to 0 every time the user makes a valid choice. This technique allows the user to make three bad selections after any good selection before the stronger message appears. See Figure 10-13 for a flowchart and pseudocode of a complete program containing all the improvements.

FIGURE 10-12: THE `looping()` MODULE WITH A STRONGER ERROR MESSAGE AFTER THREE ERRORS

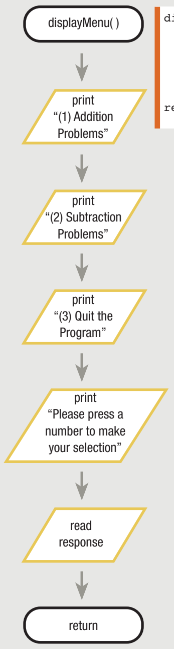

looping( )
response = 1?
No
Yes
addition( )
response = 2?
No
Yes
subtraction( )
errorCount = errorCount + 1
errorCount > 2?
No
Yes
print "You must select 1, 2, or 3"
print "Please see the system administrator for help"
displayMenu( )
return

```
looping()
     if response = 1 then
          perform addition()
     else
          if response = 2 then
               perform subtraction()
          else
               errorCount = errorCount + 1
               if errorCount > 2 then
                    print "Please see the system administrator for help"
               else
                    print "You must select 1, 2, or 3"
               endif
          endif
     endif
     perform displayMenu()
return
```

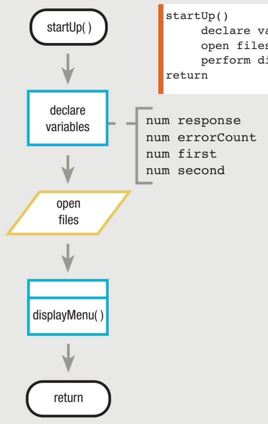
FIGURE 10-13: COMPLETE PROGRAM ALLOWING THREE ATTEMPTS AT SUCCESSFUL MENU SELECTION BEFORE STRONGER MESSAGE APPEARS



```
start
     perform startUp()
     while response not = 3
          perform looping()
     endwhile
     perform cleanUp()
stop
```

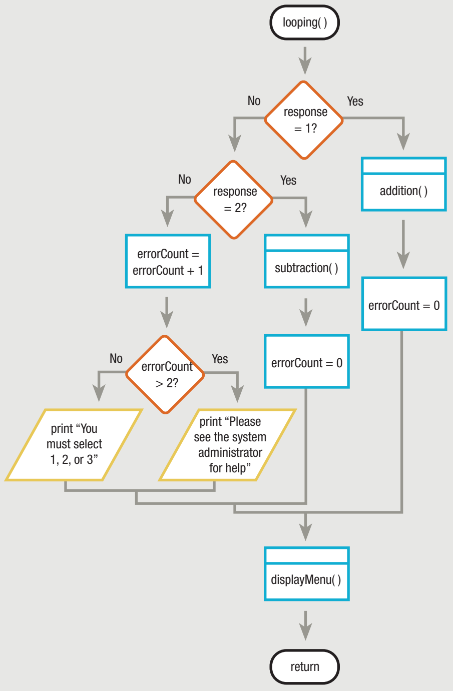

displayMenu( )
print "(1) Addition Problems"
print "(2) Subtraction Problems"
print "(3) Quit the Program"
print "Please press a number to make your selection"
read response
return

```
displayMenu()
     print "(1) Addition Problems"
     print "(2) Subtraction Problems"
     print "(3) Quit the Program"
     print "Please press a number to make your selection"
     read response
return
```



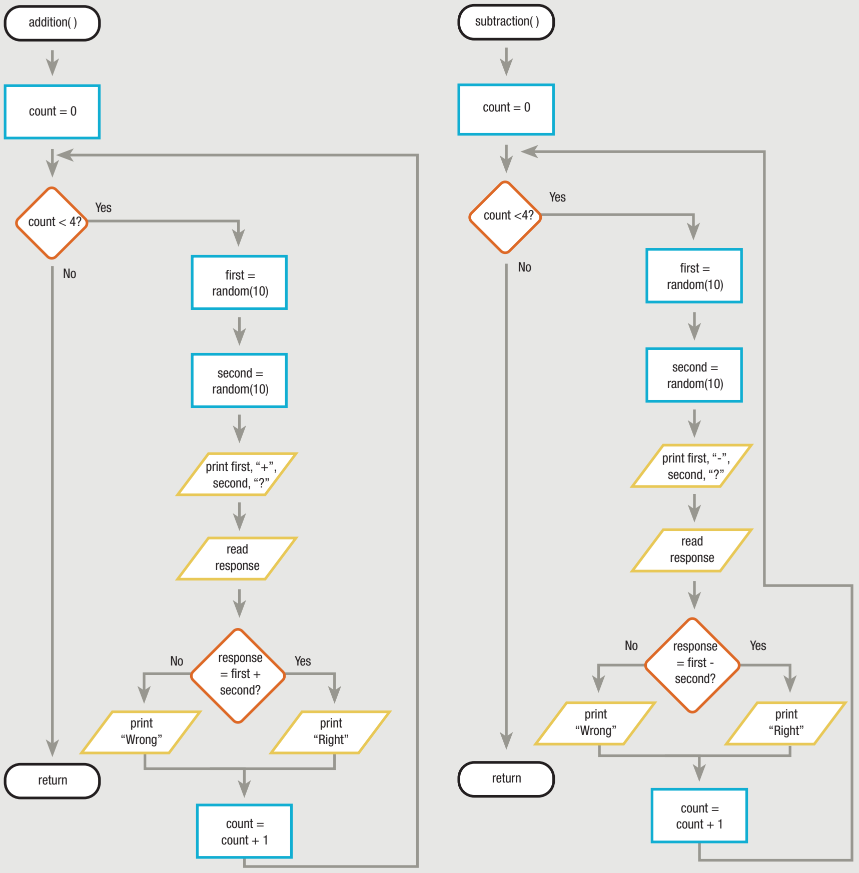
```
num response
num errorCount
num first
num second
```

```
startUp()
     declare variables
     open files
     perform displayMenu()
return
```



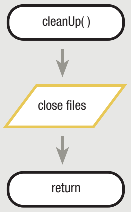
```
looping()
     if response = 1 then
          perform addition()
          errorCount = 0
     else
          if response = 2 then
               perform subtraction()
               errorCount = 0
          else
               errorCount = errorCount + 1
               if errorCount > 2 then
                    print "Please see the system administrator for help"
               else
                    print "You must select 1, 2, or 3"
               endif
          endif
     endif
     perform displayMenu()
return
```

FIGURE 10-13: COMPLETE PROGRAM ALLOWING THREE ATTEMPTS AT SUCCESSFUL MENU SELECTION BEFORE STRONGER MESSAGE APPEARS (CONTINUED)



```
addition()
     count = 0
     while count < 4
          first = random(10)
          second = random(10)
          print first, "+", second, "?"
          read response
          if response = first + second then
              print "Right"
          else
              print "Wrong"
          endif
          count = count + 1
     endwhile
return
```

```
subtraction()
     count = 0
     while count < 4
          first = random(10)
          second = random(10)
          print first, "-", second, "?"
          read response
          if response = first - second then
              print "Right"
          else
              print "Wrong"
          endif
          count = count + 1
     endwhile
return
```



```
cleanUp()
     close files
return
```

USING THE CASE STRUCTURE TO MANAGE A MENU

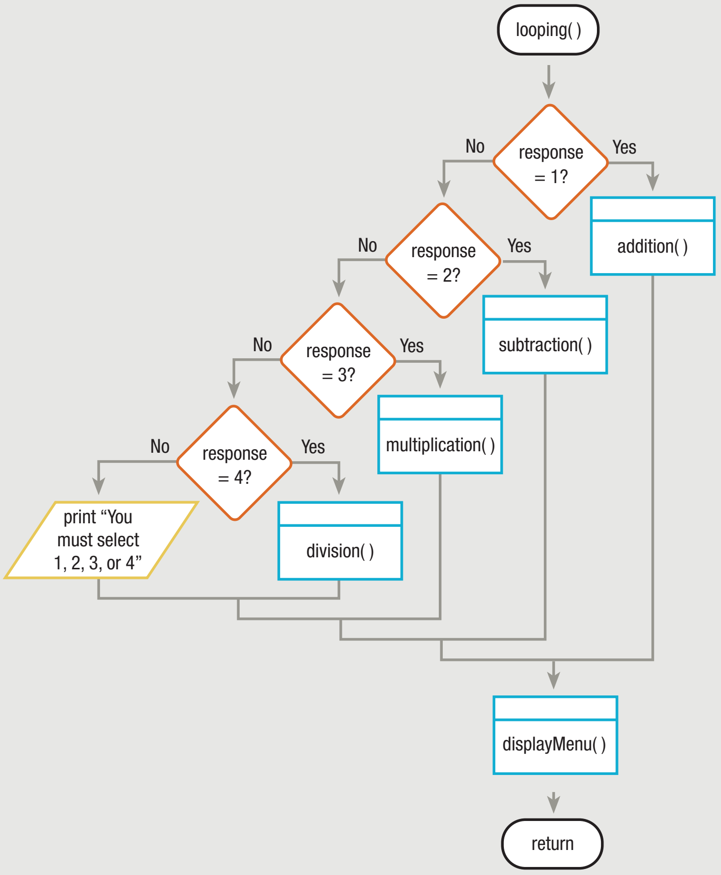
The arithmetic drill program contains just three valid user options: numeric entries that represent addition, subtraction, or quitting the program. Many menus include more than three options, but the main logic of such programs is not substantially different from that in programs with only three. You just include more decisions that lead to additional submodules. For example, Figure 10-14 shows the main logic for a menu program with four optional arithmetic drills.

In Chapter 2 and again in Chapter 5, you learned about the case structure. You can use the case structure to make decisions when you need to test a single variable against several possible values. The case structure is particularly convenient to use in menu-driven programs, because you decide from among several courses of action based on the value in the user's `response` variable. The case structure often is a more convenient way to express a series of individual decisions.

As you have learned, the syntax of case structures in most programming languages allows you to make a series of comparisons, and if none is true, an *Other* or *Default* option executes. Using a default option is a great convenience in a menu-driven program, because a user usually can enter many more invalid responses than valid ones. Figure 10-15 shows the logic of a four-option arithmetic drill program that uses the case structure.

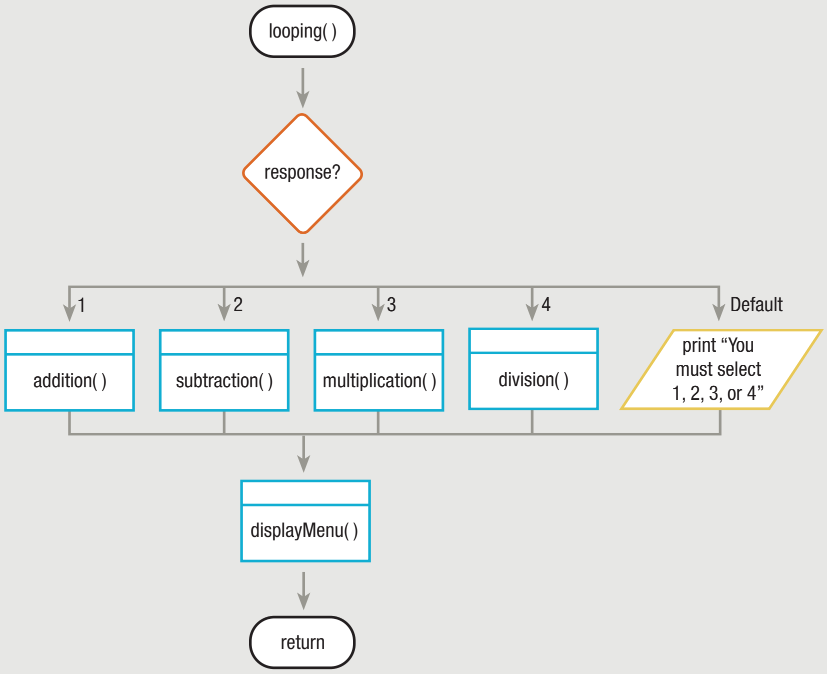
All menu-driven programs should be **user-friendly**, meaning that they should make it easy for the user to make desired choices. Instead of requiring a user to type numbers to select an arithmetic drill, you can improve the menu program by allowing the user the additional option of typing the first letter of the desired option—for example, *A* for addition. To enable the menu program to accept alphabetic characters as a variable named `response`, you must make sure you declare `response` as a character variable in the `startUp()` module. Numeric variables can hold only numbers, but character variables can hold alphabetic characters (such as *A*) as well as numbers.

FIGURE 10-14: MAIN LOGIC OF PROGRAM CONTAINING FOUR OPTIONAL ARITHMETIC DRILLS



```
looping()
     if response = 1 then
          perform addition()
     else
          if response = 2 then
               perform subtraction()
          else
               if response = 3 then
                    perform multiplication()
               else
                    if response = 4 then
                         perform division()
                    else
                         print "You must select 1, 2, 3, or 4"
                    endif
               endif
          endif
     endif
     perform displayMenu()
return
```

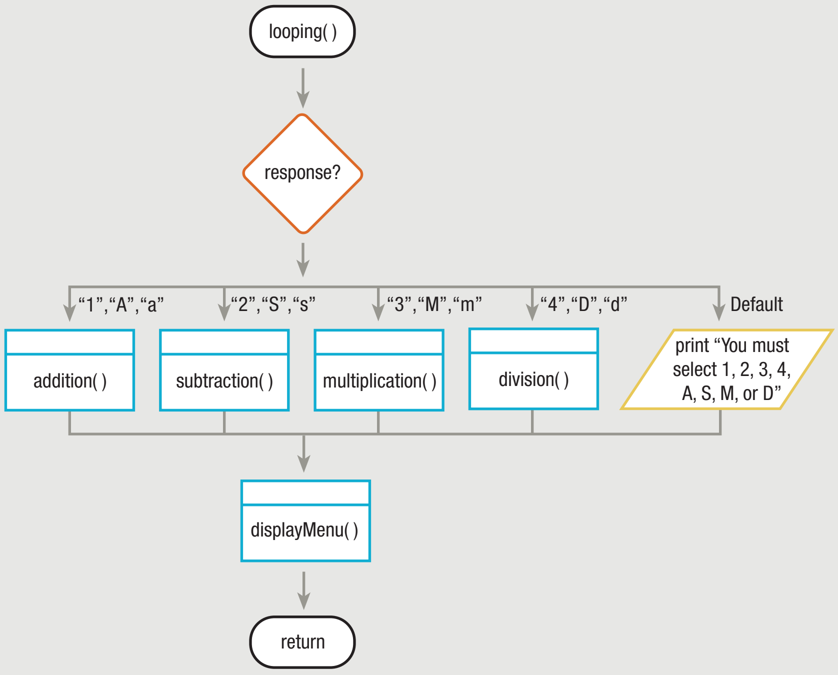
FIGURE 10-15: MENU PROGRAM USING THE CASE STRUCTURE



```
looping()
     case based on response
          case 1
               perform addition()
          case 2
               perform subtraction()
          case 3
               perform multiplication()
          case 4
               perform division()
          default
               print "You must select 1, 2, 3, or 4"
     endcase
     perform displayMenu()
return
```

Programmers sometimes overlook the fact that computers recognize uppercase letters as being different from their lowercase counterparts. Thus, a response of *A* is different from a response of *a*. A good menu-driven program probably would allow any of three responses for the first option of *(1) Addition—1, A,* or *a*. Figure 10-16 shows the case structure that performs the menu option selection when the user can enter a variety of responses for each menu choice.

FIGURE 10-16: MENU PROGRAM USING THE CASE STRUCTURE WITH MULTIPLE ALLOWED RESPONSES



```
looping()
     case based on response
          case "1", "A", "a"
               perform addition()
          case "2", "S", "s"
               perform subtraction()
          case "3", "M", "m"
               perform multiplication()
          case "4", "D", "d"
               perform division()
          default
               print "You must select 1, 2, 3, 4,
                     A, S, M, or D"
     endcase
     perform displayMenu()
return
```

USING MULTILEVEL MENUS

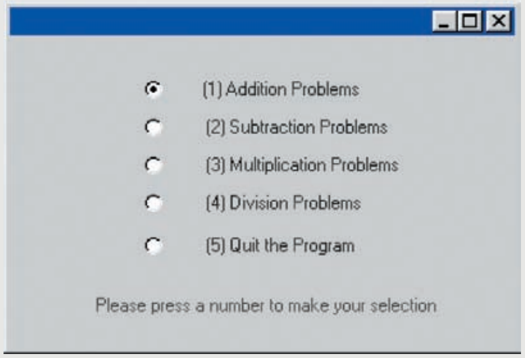
Sometimes, a program requires more options than can easily fit in one menu. When you need to present the user with a large number of options, you invite several potential problems:

- If there are too many options to fit on the display at one time, the user might not realize that additional options are available.
- The screen is too crowded to be visually pleasing when you try to force all the options to fit on the screen.
- Users become confused and frustrated when you present them with too many choices.

When you have many menu options to present, using a multilevel menu might be more effective than using a single-level menu. With a **multilevel menu**, the selection of a menu option leads to another menu from which the user can make further, more refined selections.

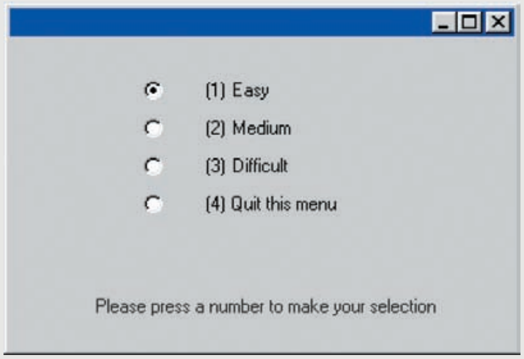
For example, an arithmetic drill program might contain three difficulty levels for each type of problem. After the user sees a menu like the one shown in Figure 10-17, he or she can choose to quit the program immediately, without selecting an arithmetic drill. You refer to a menu that controls whether the program will continue as the **main menu** of a program. Alternatively, the user can choose to continue the program, selecting an Addition, Subtraction, Multiplication, or Division arithmetic drill. No matter which drill the user chooses, you can display a second menu like the one shown in Figure 10-18. A second-level (or later-level) menu is a **submenu**.

FIGURE 10-17: FIRST OR MAIN MENU FOR ARITHMETIC DRILL PROGRAM


[✓ (1) Addition Problems]
(2) Subtraction Problems
(3) Multiplication Problems
(4) Division Problems
(5) Quit the Program

Please press a number to make your selection

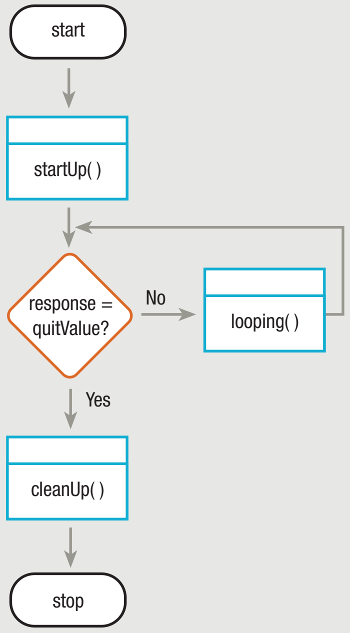
FIGURE 10-18: SECOND OR SUBMENU FOR ARITHMETIC DRILL PROGRAM


(1) Easy
(2) Medium
(3) Difficult
(4) Quit this menu

Please press a number to make your selection

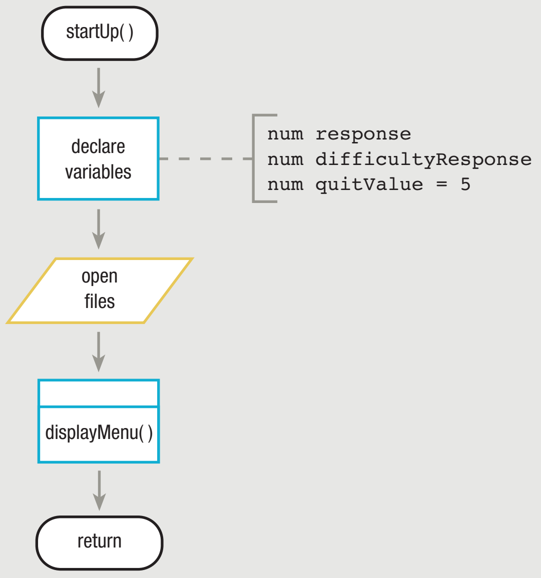
The mainline logic of this multilevel menu arithmetic program calls a `startUp()` module in which the first menu presents options for the four types of arithmetic problems—*Addition, Subtraction, Multiplication,* and *Division*—as well as an option to quit. When the user makes a selection—for example, *Addition*—the mainline logic determines that `response` is not the quit option, so the `looping()` module executes. Figures 10-19, 10-20, and 10-21 show flowcharts and pseudocode for the mainline logic, `startUp()` module, and `displayMenu()` module, respectively.

FIGURE 10-19: FLOWCHART AND PSEUDOCODE FOR MAINLINE LOGIC FOR MULTILEVEL MENU PROGRAM



```
start
     perform startUp()
     while response not equal to quitValue
            perform looping()
     endwhile
     perform cleanUp()
stop
```

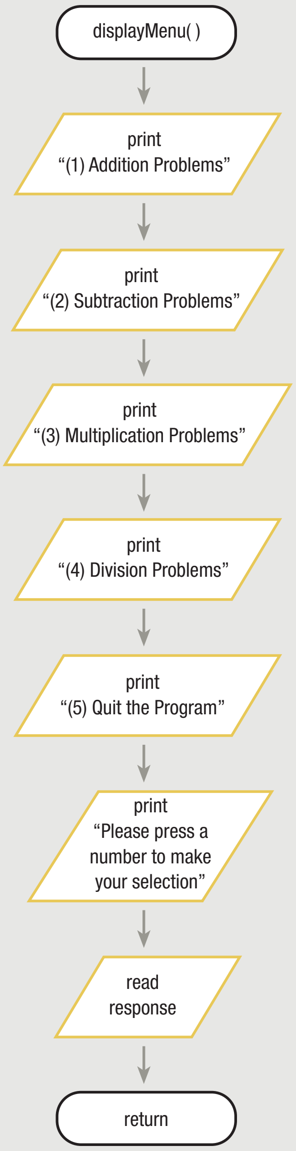
FIGURE 10-20: FLOWCHART AND PSEUDOCODE FOR `startUp()` MODULE FOR MULTILEVEL MENU PROGRAM



```
startUp()
    declare variables
    open files
    perform displayMenu()
return
```

When the program begins, unless the user chooses to quit by entering the `quitValue` (*5*) for the `response` in the `startUp()` module, the `looping()` module executes. The `looping()` module uses a case structure to select one of five actions. Either the user has entered the correct `response` to select addition, subtraction, multiplication, or division problems, or the user has selected an invalid option. If the user selects an invalid option, an error message "Sorry. Invalid entry." appears. Whether or not the user selects an entry that performs one of the four arithmetic drill modules, the final step in the `looping()` module displays the menu again and waits for the next `response`. Back in the mainline logic, the new `response` value is tested, and if the user has entered anything other than the *quitValue*, the `looping()` module executes again. Figure 10-22 shows the flowchart and pseudocode for this version of the `looping()` module.

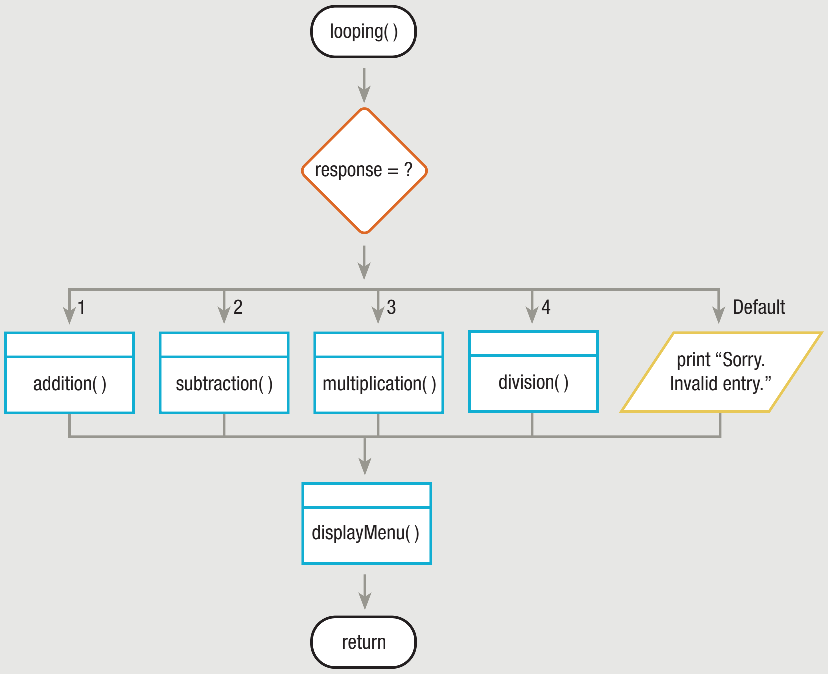
FIGURE 10-21: FLOWCHART AND PSEUDOCODE FOR `displayMenu()` MODULE FOR MULTILEVEL MENU PROGRAM


displayMenu( )
print "(1) Addition Problems"
print "(2) Subtraction Problems"
print "(3) Multiplication Problems"
print "(4) Division Problems"
print "(5) Quit the Program"
print "Please press a number to make your selection"
read response
return

```
displayMenu()
    print "(1) Addition Problems"
    print "(2) Subtraction Problems"
    print "(3) Multiplication Problems"
    print "(4) Division Problems"
    print "(5) Quit the Program"
    print "Please press a number to make your selection"
    read response
return
```

In the `looping()` module in Figure 10-22, if the user selects a valid option, then the module executes one of the four arithmetic drill modules. For example, if the user selects *1* for *Addition Problems*, then the `addition()` module executes.

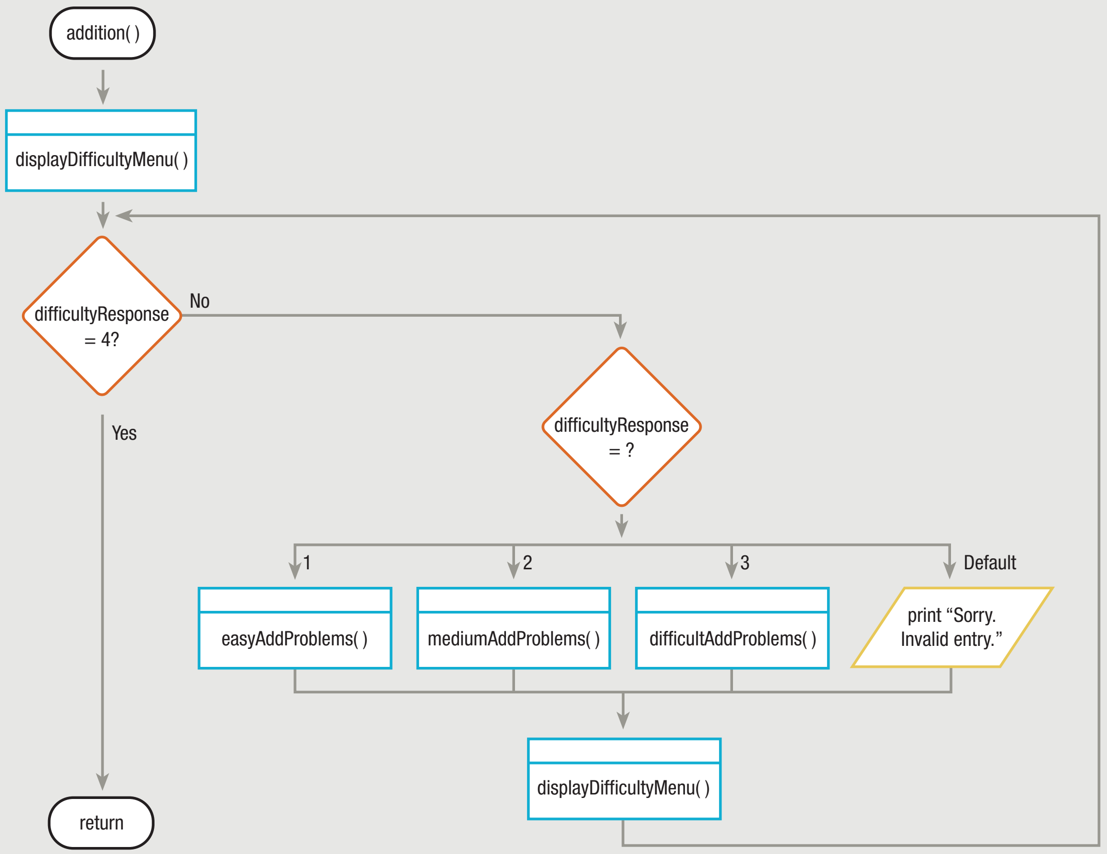
FIGURE 10-22: FLOWCHART AND PSEUDOCODE FOR `looping()` MODULE FOR MULTILEVEL MENU PROGRAM



```
looping()
    case based on response
           case 1
                 perform addition()
           case 2
                 perform subtraction()
           case 3
                 perform multiplication()
           case 4
                 perform division()
           default
                 print "Sorry. Invalid entry."
    endcase
    perform displayMenu()
return
```

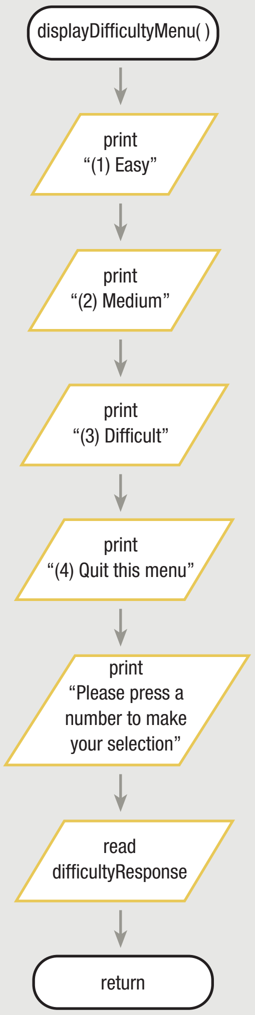
Within the `addition()` module, the first task is to allow the user to select a problem-difficulty level from a submenu like the one shown in Figure 10-18. Figure 10-23 shows the flowchart and pseudocode for the `addition()` module. Within the `addition()` module, you call another module to display the difficulty level. Shown in Figure 10-24, this module allows the user to choose easy, medium, or difficult addition problems. If the user selects to quit this menu by entering a *4*, then the user will leave the `addition()` module and return to the main menu to choose a different type of arithmetic problem, choose addition again, or quit the program. In the `displayDifficultyMenu()` module, if the user makes a selection other than *4*, the case structure in the `addition()` module determines one of four actions: either one of three addition problem modules executes, or the user is informed that the choice is invalid. In any case, the last action of the `addition()` module is to display the difficulty level menu again. As long as users choose options other than *4*, they can continue to select addition problem drills at any of the three difficulty levels.

FIGURE 10-23: FLOWCHART AND PSEUDOCODE FOR `addition()` MODULE FOR MULTILEVEL MENU PROGRAM



```
addition()
    perform displayDifficultyMenu()
    while difficultyResponse is not equal to 4
          case based on difficultyResponse
              case 1
                   perform easyAddProblems()
              case 2
                   perform mediumAddProblems()
              case 3
                   perform difficultAddProblems()
              default
                   print "Sorry. Invalid entry."
          endcase
          perform displayDifficultyMenu()
    endwhile
return
```

FIGURE 10-24: FLOWCHART AND PSEUDOCODE FOR `displayDifficultyMenu()` MODULE FOR MULTILEVEL MENU PROGRAM

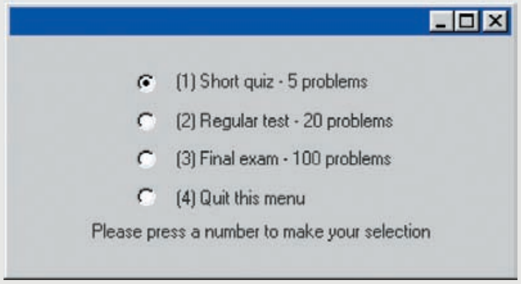


```
displayDifficultyMenu()
    print "(1) Easy"
    print "(2) Medium"
    print "(3) Difficult"
    print "(4) Quit this menu"
    print "Please press a number to make your selection"
    read difficultyResponse
return
```

The `subtraction()`, `multiplication()`, and `division()` modules can contain code similar to that in the `addition()` module. That is, each module can display a submenu of difficulty levels. The actual arithmetic problems do not execute until the user reaches the `easyAddProblems()` module or one of its counterparts.

Many programs have multiple menu levels. For example, you might want the `easyAddProblems()` module to display a new menu asking the user for the number of problems to attempt. Figure 10-25 shows a possible menu.

FIGURE 10-25: THIRD MENU FOR ARITHMETIC DRILL PROGRAM



[● (1) Short quiz - 5 problems]
(2) Regular test - 20 problems
(3) Final exam - 100 problems
(4) Quit this menu
Please press a number to make your selection

You would not need to learn any new techniques to create as many levels of menus as the application warrants. The module that controls each new level can:

- Display a menu.
- Accept a response.
- Perform another module based on the selection (or inform the user of an error) while the user does not select the *Quit* option for the specific menu level.
- Display the menu and accept a response again.

VALIDATING INPUT

Menu programs rely on a user's input to select one of several paths of action. Other types of programs also require a user to enter data. Unfortunately, you cannot count on users to enter valid data, whether they are using a menu or supplying information to a program. Users will make incorrect choices because they don't understand the valid choices, or simply because they make typographical errors. Therefore, the programs you write will be improved if you employ **defensive programming**, which means trying to prepare for all possible errors before they occur. Incorrect user entries are by far the most common source of computer errors.

You can circumvent potential problems caused by a user's invalid data entries by validating the user's input. **Validating input** involves checking the user's responses to ensure they fall within acceptable bounds. Validating input does not eliminate all program errors. For example, if a user can choose option *1* or option *2* from a menu, validating the input means you check to make sure the user response is *1* or *2*. If the user enters a *3*, you can issue an error message. However, if the user enters a *2* when she really wants a *1*, there is no way you can validate the response. Similarly, if a user must enter his birth date, you can validate that the month falls between 1 and 12; you usually cannot verify that the user has typed his true birth date.

TIP Validating input is also called *editing* data.

TIP Programmers employ the acronym GIGO to mean "garbage in, garbage out." It means that if your input is incorrect, your output is worthless.

The correct action to take when you find invalid data depends on the application. Within an interactive program, you might require the user to reenter the data. If your program uses a data file, you might print a message so someone can correct the invalid data. Alternatively, you can force the invalid data to a default value. **Forcing** a field to a value means you override incorrect data by setting the field to a specific value. For example, you might decide that if a month value does not fall between 1 and 12, you will force the field to 0 or 99. This indicates to those who use the data that no valid value exists.

New programmers often make the following two kinds of mistakes when validating data:

- They use incorrect logic to check for valid responses when there is more than one possible correct entry.
- They fail to account for the user making multiple invalid entries.

For example, assume a user is required to respond with a *Y* or *N* to a yes-or-no question. The pseudocode in Figure 10-26 appears to check for valid responses.

FIGURE 10-26: INVALID METHOD FOR VALIDATING USER RESPONSE

```
print "Do you want to continue? Enter Y or N."
read userAnswer
if userAnswer not equal to "Y" OR userAnswer not equal to "N" then
       print "Invalid response. Please type Y or N"
       read userAnswer
endif
```

The logic shown in Figure 10-26 intends to make sure that the user enters a *Y* or an *N*. However, if you use the logic shown in Figure 10-26, all users will see the "Invalid response" error message, no matter what they type. Remember, when you use OR logic, only one of the two expressions used in each half of the OR expression must be true for the whole expression to be true. For example, if the user types a *B*, then `userAnswer` is not equal to *Y*. Therefore, `userAnswer not equal to "Y"` is true, and the "Invalid response" message is displayed. However, if the user types an *N*, `userAnswer` also is not equal to *Y*. Again, the condition in the `if` statement is true, and the "Invalid response" message prints, even though the response is actually valid. Similarly, if the user types a *Y*, `userAnswer not equal to "Y"` is false, but `userAnswer not equal to "N"` is true, so again "Invalid response" prints. Every character that exists is either not Y or not N, even "Y" and "N". The correct logic prints the "Invalid response" message when `userAnswer` is not Y *and it is also not N*. See Figure 10-27.

FIGURE 10-27: IMPROVED METHOD FOR VALIDATING USER RESPONSE

```
print "Do you want to continue? Enter Y or N."
read userAnswer
if userAnswer not equal to "Y" AND userAnswer not equal to "N" then
       print "Invalid response. Please type Y or N"
       read userAnswer
endif
```

TIP You first learned about OR decision logic in Chapter 5.

If you use the logic shown in Figure 10-27, when the user types an invalid response, you will correctly display the error message and get a new `userAnswer`. However, you have not made allowance for the user typing an invalid response a second time. Instead of using a decision statement to check for a valid response, you can use a loop to continue to issue error messages and get new input as long as the user continues to make invalid selections. Figure 10-28 shows the logic for the best method for validating user input.

FIGURE 10-28: BEST METHOD FOR VALIDATING USER RESPONSE

```
print "Do you want to continue? Enter Y or N."
read userAnswer
while userAnswer not equal to "Y" AND userAnswer not equal to "N"
      print "Invalid response. Please type Y or N"
      read userAnswer
endwhile
```

UNDERSTANDING TYPES OF DATA VALIDATION

The data you use within computer programs is varied. It stands to reason that validating data requires a variety of methods. In the last section, you learned to check for an exact match of a user response to the character “Y” or “N”. In addition, some of the techniques you want to master include validating:

- Data type
- Range
- Reasonableness and consistency of data
- Presence of data

VALIDATING A DATA TYPE

Some programming languages allow you to check data items to make sure they are the correct data type. Although this technique varies from language to language, you can often make a statement like the one shown in Figure 10-29. In this program segment, `isNumeric()` represents a method call; it is used to check whether the entered `employeeSalary` falls within the category of numeric data. A method such as `isNumeric()` is most often provided with the language translator you use to write your programs. Such a method operates as a black box; you can use its results without understanding its internal statements.

FIGURE 10-29: METHOD FOR CHECKING DATA FOR CORRECT TYPE

```
print "Enter salary."
read employeeSalary
while employeeSalary not isNumeric()
       print "Salary not numeric. Please reenter."
       read employeeSalary
endwhile
```

TIP Some languages require you to check data against the actual machine codes (such as ASCII or EBCDIC) used to store the data, to determine if the data is the appropriate type.

Besides allowing you to check whether a value is numeric, some languages contain methods with names like `isChar()` (for "is the value a character data type?"), `isWhitespace()` (meaning "is the value a nonprinting character such as a space, a tab, or the Enter key?"), `isUpper()` (meaning "is the value a capital letter?"), and `isLower()` (meaning "is the value a lowercase letter?").

In many languages, you accept all user data as a string of characters, and then use built-in methods to attempt to convert the characters to the correct data type for your application. When the conversion methods succeed, you have useful data; when the conversion methods fail because the user has entered the wrong data type, you can take appropriate action, such as issuing an error message, reprompting the user, or forcing the data to a default value.

VALIDATING A DATA RANGE

Sometimes, a user response or other data must fall within a range of values. For example, when the user enters a month, you typically require it to fall between 1 and 12, inclusive. The method you use to check for a valid range is similar to one you use to check for an exact match; you continue to prompt for and receive responses while the user's response is out of range. See Figure 10-30.

FIGURE 10-30: METHOD FOR VALIDATING USER RESPONSE WITHIN RANGE

```
print "Enter month."
read userAnswer
while userAnswer < 1 OR userAnswer > 12
       print "Invalid response. Please enter month 1 through 12."
       read userAnswer
endwhile
```

VALIDATING REASONABLENESS AND CONSISTENCY OF DATA

Data items can be the correct type and within range, but still be incorrect. You have experienced this phenomenon yourself if anyone has ever misspelled your name or overbilled you. The data might have been the correct type—that is, alphabetic letters were used in your name—but the name itself was incorrect. There are many data items that you cannot check for reasonableness; it is just as reasonable that your name is Catherine as it is that your name is Katherine or Kathryn.

However, there are many data items that you can check for reasonableness. If you make a purchase on May 3, 2007, then the payment cannot possibly be due prior to that date. Perhaps within your organization, if you work in Department 12, you cannot possibly make more than $20.00 per hour. If your zip code is 90201, your state of residence cannot be New York. If your pet's breed is stored as "Great Dane," then its species cannot be "bird." Each of these examples involves comparing two data fields for reasonableness and consistency. You should consider making as many such comparisons as possible when writing your own programs.

TIP ☐☐☐☐ Frequently, testing for reasonableness and consistency involves using additional data files. For example, to check that a user has entered a valid county of residence for a state, you might use a file that contains every county name within every state in the United States, and check the user's county against those contained in the file.

VALIDATING PRESENCE OF DATA

Sometimes, data is missing from a file, either for a reason or by accident. A job applicant might fail to submit an entry for the `salaryAtPreviousJob` field, or a client might have no entry for the `emailAddress` field. A data-entry clerk might accidentally skip a field when typing records. Many programming languages allow you to check for missing data and take appropriate action with a statement similar to `if emailAddress is blank perform noEmailModule()`. You can place any instructions you like within `noEmailModule()`, including forcing the field to a default value or issuing an error message.

Good defensive programs try to foresee all possible inconsistencies and errors. The more accurate your data, the more useful information you will produce as output from your programs.

CHAPTER SUMMARY

- Programs for which all the data items are gathered prior to running use batch processing. Programs that depend on user input while they are running use interactive, real-time, online processing. A menu program is a common type of interactive program in which the user sees a number of options on the screen and can select any one of them.
- When you create a single-level menu, the user makes a selection from only one menu before using the program for its ultimate purpose. The user's response controls the mainline logic of a menu program.
- When you code a module as a black box, the module statements are invisible to the rest of the program. Many versions of a module can substitute for one another. When programmers develop systems containing many modules, they often code "empty" black box procedures, called stubs; later they can code the details in the stub modules. In addition, most programming languages provide you with built-in black box functions.
- A programmer can improve a menu program and assist the user by displaying a message when the selected response is not one of the allowable menu options. Another user-friendly improvement to a program adds a counter that keeps track of a user's invalid responses and issues a stronger message after a specific number of invalid responses.
- You can use the case structure to make decisions when you need to test a single variable against several possible values. The case structure is particularly convenient to use in menu-driven programs, because you decide from among several courses of action based on the value in the user's response variable.
- When a program requires more options than can easily fit in one menu, you can use a multilevel menu. With a multilevel menu, the selection of an option from a main menu leads to a submenu from which the user can make further, more refined selections. With multilevel menus, the module that controls each new level can display a menu, accept a response, and—while the user does not select the quit option for that menu level—perform another module based on the selection (or inform the user of an error). Finally, the module for each menu level displays the menu and accepts a response again.
- You can circumvent potential problems caused by a user's invalid data entries by validating the user's input. Validating input involves checking the user's responses to ensure they fall within acceptable bounds, and taking one of several possible actions. Common mistakes when validating data include using incorrect logic and failing to account for the user making multiple invalid entries.
- Some of the techniques you want to master include validating data type, range, reasonableness and consistency of data, and presence of data.

KEY TERMS

Programs for which all the data items are gathered prior to running use **batch processing**.

Programs that depend on user input while the programs are running use **interactive processing**.

Interactive computer programs are often called **real-time applications**, because they run while a transaction is taking place, not at some later time.

You also can refer to interactive processing as **online processing**, because the user's data or requests are gathered during the execution of the program, while the computer is operating.

A batch processing system can be **offline**; that is, you can collect data such as time cards or purchase information well ahead of the actual computer processing of the paychecks or bills.

A **menu program** is a common type of interactive program in which the user sees a number of options on the screen and can select any one of them.

Console applications are programs that require the user to enter choices using the keyboard.

Graphical user interface applications allow the user to use a mouse or other pointing device to enter choices.

A **single-level menu** is one from which a user makes a selection that results in the program's ultimate purpose, as opposed to displaying additional menus.

When code exists in a **black box**, module statements are "invisible" to the rest of the program.

Stubs are empty procedures, intended to be coded later.

Functions are modules that automatically provide a mathematical value such as a square root, absolute value, or random number.

User-friendly programs are those that make it easy for the user to make desired choices.

With a **multilevel menu**, the selection of a menu option leads to another menu from which the user can make further, more refined selections.

The **main menu** of a program is the menu that determines whether execution of the program will continue.

A second-level, or later-level, menu is a **submenu**.

Defensive programming involves trying to prepare for all possible errors before they occur.

Validating input involves checking the user's responses to ensure they fall within acceptable bounds.

Forcing a field to a value means you override incorrect data by setting the field to a specific value.

REVIEW QUESTIONS

1. **Programs for which all the data items are gathered prior to running use __________ processing.**
    a. batch
    b. interactive
    c. online
    d. real-time
2. **Programs that depend on user input while the programs are running use __________ processing.**
    a. artificial
    b. delayed
    c. batch
    d. interactive
3. **Which of the following means the same as interactive processing?**
    a. query processing
    b. virtual processing
    c. real-time processing
    d. batch processing
4. **A menu program is a common type of __________ program.**
    a. batch
    b. interactive
    c. control break
    d. offline
5. **If a user makes a selection from only one menu before using the program for its ultimate purpose, then the menu is a __________ menu.**
    a. primary
    b. single-level
    c. focal
    d. batch
6. **When module statements are invisible to the rest of a program, they are said to exist within a __________.**
    a. black hole
    b. magic hat
    c. mirror
    d. black box
7. **Modules containing no code that are used as temporary placeholders are called __________.**
    a. black boxes
    b. stubs
    c. fill-ins
    d. padded

8. **Most programming languages provide you with built-in modules called ________ that automatically provide a mathematical value such as a square root, absolute value, or random number.**
   a. functions
   b. formulas
   c. stubs
   d. black boxes

9. **Writing a program that provides a user with increasingly detailed help messages as the user continues to make data-entry errors requires that the program contain a ________.**
   a. loop
   b. counter
   c. both of these
   d. neither a nor b

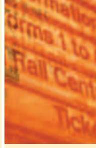
10. **The structure that provides a more convenient way to express a series of decisions that are based on the value of a single variable is the ________ structure.**
    a. loop
    b. do until
    c. case
    d. sequence

11. **A program that makes it easy for a user to accomplish tasks is said to be ________.**
    a. simplex
    b. user-friendly
    c. structured
    d. accommodating

12. **You might need to create a multilevel menu from a single-level one if ________.**
    a. you do not have enough options to fill the screen
    b. users are allowed only true-false choices
    c. the screen appears too crowded
    d. all of the above

13. **Where is the user selection that ends a program most likely to appear?**
    a. in a program's main menu
    b. in a program's first submenu
    c. in a program's last submenu
    d. in every menu in a program

14. **Writing programs that try to prepare for all possible user errors is known as ________ programming.**
    a. proactive
    b. cautious
    c. aggressive
    d. defensive



15. Checking to ensure that data values fall within acceptable bounds is known as ________ data.

a. forcing
b. defending
c. classifying
d. validating

16. Which value for `deptNumber` would be considered valid using the following code?

```
if deptNumber not = 1 OR deptNumber not = 2 then
     print "Invalid number"
else
     print "Valid number"
endif
```

a. 1
b. 2
c. Both 1 and 2 are valid.
d. Neither 1 nor 2 is valid.

17. Which value for `deptNumber` would be considered valid using the following code?

```
if deptNumber not = 5 AND deptNumber not = 6 then
     print "Invalid number"
else
     print "Valid number"
endif
```

a. 5
b. 6
c. Both 5 and 6 are valid.
d. Neither 5 nor 6 is valid.

18. Which of the following student data items could most easily be validated by a program used by a college?

a. The name of the high school the student attended is spelled correctly.
b. The student's middle name is correct.
c. The student's grade point average is between 0.0 and 4.0, inclusive.
d. The student's last tuition payment is for the correct amount.

19. Which of the following data items could least easily be validated by a program used by a grocery store?

a. The Universal Product Code for an item contains the correct number of digits (12).
b. The product name is alphabetic.
c. The date the product was last ordered from the manufacturer is a valid date and no more than two years old.
d. The product price is no more than any other store in the state is charging this week.

20. Good defensive programs ________.

a. catch all errors
b. catch all range errors, but not necessarily other error types
c. catch many errors
d. seldom catch errors until the data is visually verified by clerical employees

FIND THE BUGS

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

1. **Head Gear, Inc. sells customized baseball caps embroidered with your team name or company logo. This application allows a user to enter the phrase to be imprinted on a cap and a quantity. The application then displays a menu from which a user can choose the color for the caps ordered. The total amount due is displayed when the order is complete.**

```
start
     perform firstTasks()
     while phrase not = QUIT
          perform userChoices()
     endwhile
     perform finishUp()
stop

firstTasks()
     declare variables
         char phrase
         num quantity
         num colorChoice
         const num QUIT = "XXX"
         const num LOWAMOUNT = 1
         const num HIGHAMOUNT = 500
         const num DISCOUNTPRICE = 6.99
         const num REGPRICE = 8.99
         const num CUTOFF = 100
         const num CAMO_PREMIUM = 1.50
         num price
     open files
     print "Enter phrase you want embroidered on caps
     print " or enter ", QUIT, " to quit"
     read phrase
return

userChoices()
     display "Enter quantity"
     while quantity < LOWAMOUNT OR quantity > HIGHAMOUNT
        print "Invalid amount. Please re-enter quantity"
        read quantity
     endwhile
     perform displayMenu()
     perform computePrice()
     print "Enter phrase you want embroidered on caps
     print " or enter ", QUIT, " to quit"
return

displayMenu()
    colorChoice = 0
    whileColorChoice < 1 OR colorChoice > 6
      print "Choose a color from the following menu"
      print "(1) Black"
      print "(2) Red"
      print "(3) Blue"
      print "(4) Green"
      print "(5) White"
      print "(6) Camouflage"
    endwhile
return

computePrice()
    if quantity < = CUTOFF then
        price = REGPRICE * quantity
    else
        price = DISCOUNTPRICE
    endif
    if colorChoice = 6 then
         price = price + CAMO_PREMIUM * quantity
    endif
    print "Total is $ ", price
return

finishUp()
    close files
return
```

2. **The Good Thoughts Web site lets users select whether they are in the mood for an inspirational, motivational, or empathetic message. A message is randomly selected from a database of quotes and displayed. (Assume that a random number can be obtained by passing a numeric argument to a built-in `rand()` function that returns a value from 0 through one less than the argument.)**

```
start
     perform getReady()
     while entry not = QUIT
          perform displayMessage()
     endwhile
     perform finishUp()
stop
```

```
getReady()
     declare variables
          num entry
          const num QUIT = 4
     open files
     perform menuSelect()
stop

menuSelect()
    userInput = 1
    whileUserInput < 1 OR userInput > QUIT
      print "Choose a type of message for the day"
      print "(1) Inspirational message"
      print "(2) Motivational message"
      print "(3) Empathetic message"
      print "(4) Quit"
    endwhile
return

displayMessage()
   num SZ = 3
   char inspirationMessages[SZ]
   char motiveMessages[SZ]
   char empathyMessages[SZ]
   inspirationMessages message[0] =
       "This is the first day of the rest of your life"
   inspirationMessages [SZ] =
       "The sun will come out tomorrow"
   inspirationMessages [NUM] =
       "The journey is the destination"
   motiveMessages[SZ] = "Go the extra mile"
   motiveMessages[2] = "Rome wasn't built in a day"
   motiveMessages[3] =
       "If at first you don't succeed, try, try again"
   empathyMess [0] = "Poor baby"
   empathyMess[1] = "I feel your pain"
   empathyMess[2] = "I know where you are coming from"

   randNum = rand(SZ)
        // A prewritten function that returns 0, 1 or 2
   if user = 1 then
     print inspireMessage[SZ]
   else
```

```
     if userInput = 2 then
         print motiveMessage[5]
     else
         print empMess[randNum]
     endif
   endif
   perform menuSelect()
return

finishUp()
    close files
return
```

EXERCISES

1. **Develop the logic for a program that gives you the following options for a trivia quiz:**

   (1) Movies
   (2) Television
   (3) Sports
   (4) Quit

   When the user selects an option, display a question that falls under the category. After the user responds, display whether the answer is correct.

   a. Draw the hierarchy chart.
   b. Draw the flowchart.
   c. Write the pseudocode.

2. **Modify the program in Exercise 1 so that when the user selects a trivia quiz topic option, you display five questions in the category instead of just one.**

3. **Develop the logic for a program that presents you with the following options for a banking machine:**

   (1) Deposit
   (2) Withdrawal
   (3) Quit

   After you select an option, the program asks you for the amount of money to deposit or withdraw, then displays your balance and allows you to make another selection. When the user selects *Quit*, display the final balance.

   a. Draw the hierarchy chart.
   b. Draw the flowchart.
   c. Write the pseudocode.

4. **Develop the logic for a program that gives you the following options:**

(1) Hot dog	1.50
(2) Fries	1.00
(3) Lemonade	.75
(4) End order	

You should be allowed to keep ordering from the menu until you press *4* for *End order*, at which point you should see a total amount due for your entire order.

a. Draw the hierarchy chart.
b. Draw the flowchart.
c. Write the pseudocode.

5. **Develop the logic for a program that gives you the following options when registering for college classes:**

(1) English 101	3
(2) Math 260	5
(3) History 100	3
(4) Sociology 151	4
(5) Quit	

You should be allowed to select as many classes as you want before you choose the *Quit* option, but you should not be allowed to register for the same class twice. The program accumulates the hours for which you have registered and displays your tuition bill at $50 per credit hour.

a. Draw the hierarchy chart.
b. Draw the flowchart.
c. Write the pseudocode.

6. **Suggest two subsequent levels of menus for each of the first two options in this main menu:**

(1) Print records from file
(2) Delete records from file
(3) Quit

7. **Develop the logic for a program that displays the rules for a sport or a game. The user can select from the following menu:**

(1) Sports
(2) Games
(3) Quit

If the user chooses *1* for *Sports*, then display options for four different sports of your choice (for example, soccer or basketball).

If the user chooses *2* for *Games*, display options for:

(1) Card games
(2) Board games
(3) Quit

Display options for at least two card games (for example, Hearts) and two board games (for example, checkers) of your choice. Then display a one- or two-sentence summary of the game rules.

a. Draw the hierarchy chart.
b. Draw the flowchart.
c. Write the pseudocode.

8. **Draw the menus and then develop the logic for a program that displays United States travel and tourism facts. The main menu should allow the user to choose a region of the country. The next level should allow the user to select a state in that region. The final level should allow the user to select a city, at which point the user can view facts such as the city's population and average temperature. Write the complete module for only one region, one state, and one city.**
   a. Draw the hierarchy chart.
   b. Draw the flowchart.
   c. Write the pseudocode.

9. **Design the menus and then develop the logic for an interactive program for a florist. The first screen asks the user to choose indoor plants, outdoor plants, nonplant items, or quit. When the user chooses indoor or outdoor plants, list at least three appropriate plants of your choice. When the user chooses a plant, display its correct price. If the user chooses the nonplant option, offer a choice of gardening tools, gift items, or quit. Depending on the user selection, display at least three gardening tools or gift items. When the user chooses one, display its price.**
   a. Draw the hierarchy chart.
   b. Draw the flowchart.
   c. Write the pseudocode.

10. **Design the menus and then develop the logic for an interactive program for a company's customer database. Store the customers' ID numbers in a 20-element array; store their balances due in a parallel 20-element array. The menu options include: add customers to the database, find a customer in the database, print the database, and quit. If the user chooses to add customers, allow the user to enter a customer ID and balance to the current list, but do not let the list exceed 20 customers. If the user chooses to print, then print all existing IDs and balances; if there are none, issue a message. If the user chooses to find a customer, issue a message if there are none; otherwise, provide a second menu with three options—find by number, find by balance, or quit. Assume that every customer has a unique ID number, but that there might be several customers with the same balance.**
    a. Draw the hierarchy chart.
    b. Draw the flowchart.
    c. Write the pseudocode.

11. **Design the logic for a program that creates job applicant records, including all input data and starting salary. The program asks users for their first name, middle initial, last name, birth date (month, day, and year), current age, date of application (month, day, and year), and the job title for which they are applying. Available jobs and starting salaries appear in the following table:**

JOB TITLE	SALARY
Clerk I	26,000
Clerk II	30,000
Administrative assistant	37,500
Technical writer	39,000
Programmer I	42,500
Programmer II	50,000

Perform as many validation checks as you can think of to make sure that complete and accurate records are created.

a. Draw the hierarchy chart.
b. Draw the flowchart.
c. Write the pseudocode.

12. **Design the logic for a program that creates student records for Creighton Technical College and assigns an advisor and a dormitory to each student. The program asks users for their first name, last name, birth date (month, day, and year), and intended major. Advisors are assigned based on major, as follows:**

MAJOR	ADVISOR LAST NAME
Business	Brown for the first 100 students, then Davis
Computer Information Systems	Cunningham for the first 100 students, then Lee
Heating and Air Conditioning	Parke
Hospitality	Hunter
Undeclared	Ulster

Dormitories are assigned based on both major and age, as follows:

MAJOR	AGE	DORMITORY
Business	under 21	Washington
Business	21 and over	Adams
Computer Information Systems	under 21	Jefferson
Computer Information Systems	21 and over	Lincoln
Heating and Air Conditioning	any	Grant
Hospitality or Undeclared	any	Wilson

Perform as many validation checks as you can think of to make sure that complete and accurate records are created.

a. Draw the hierarchy chart.
b. Draw the flowchart.
c. Write the pseudocode.

DETECTIVE WORK

1. **Many programming languages make a distinction between the terms "function" and "procedure." To most programmers, what is the difference?**
2. **What is black box testing? What are the advantages and disadvantages of this type of testing?**
3. **What is defensive programming? What is Murphy's law? What do the two have to do with each other?**

UP FOR DISCUSSION

1. **Obviously, you use a menu in a restaurant. Where else?**
2. **Have you ever used a telephone menu system that was inconvenient or frustrating? Describe the problems you encountered. Can you develop a set of recommendations for telephone menu systems?**