2 UNDERSTANDING STRUCTURE

After studying Chapter 2, you should be able to:

- Describe the features of unstructured spaghetti code
- Describe the three basic structures—sequence, selection, and loop
- Use a priming read
- Appreciate the need for structure
- Recognize structure
- Describe three special structures—case, do-while, and do-until

UNDERSTANDING UNSTRUCTURED SPAGHETTI CODE

Professional computer programs usually get far more complicated than the number-doubling program from Chapter 1, shown in Figure 2-1.

FIGURE 2-1: NUMBER-DOUBLING PROGRAM

```
get inputNumber
calculatedAnswer = inputNumber * 2
print calculatedAnswer
```

Imagine the number of instructions in the computer program that NASA uses to calculate the launch angle of a space shuttle, or in the program the IRS uses to audit your income tax return. Even the program that produces a paycheck for you on your job contains many, many instructions. Designing the logic for such a program can be a time-consuming task. When you add several thousand instructions to a program, including several hundred decisions, it is easy to create a complicated mess. The popular name for logically snarled program statements is **spaghetti code**. The reason for the name should be obvious—the code is as confusing to read as following one noodle through a plate of spaghetti.

For example, suppose you are in charge of admissions at a college, and you've decided you will admit prospective students based on the following criteria:

- You will admit students who score 90 or better on the admissions test your college gives, as long as they are in the upper 75 percent of their high-school graduating class. (These are smart students who score well on the admissions test. Maybe they didn't do so well in high school because it was a tough school, or maybe they have matured.)
- You will admit students who score at least 80 on the admissions test if they are in the upper 50 percent of their high-school graduating class. (These students score fairly well on the test, and do fairly well in school.)
- You will admit students who score as low as 70 on your test if they are in the top 25 percent of their class. (Maybe these students don't take tests well, but obviously they are achievers.)

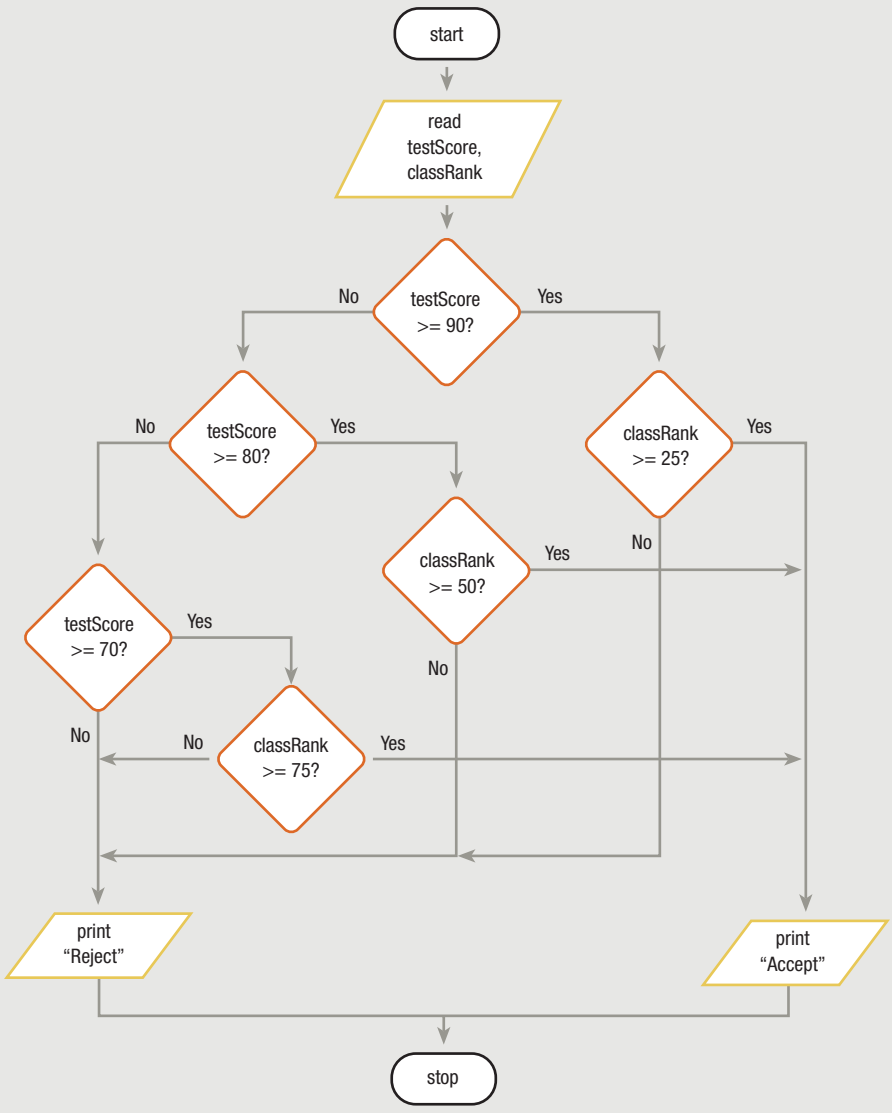
Table 2-1 summarizes the admission requirements.

TABLE 2-1: ADMISSION REQUIREMENTS

Test score	High-school rank
90–100	Upper 75 percent (from 25th to 100th percentile)
80–89	Upper half (from 50th to 100th percentile)
70–79	Upper 25 percent (from 75th to 100th percentile)

The flowchart for this program could look like the one in Figure 2-2. This kind of flowchart is an example of spaghetti code. Many computer programs (especially older computer programs) bear a striking resemblance to the flowchart in Figure 2-2. Such programs might "work"—that is, they might produce correct results—but they are very difficult to read and maintain, and their logic is difficult to follow.

FIGURE 2-2: SPAGHETTI CODE EXAMPLE

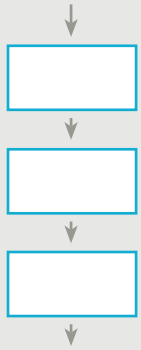


UNDERSTANDING THE THREE BASIC STRUCTURES

In the mid-1960s, mathematicians proved that any program, no matter how complicated, can be constructed using one or more of only three structures. A **structure** is a basic unit of programming logic; each structure is a sequence, selection, or loop. With these three structures alone, you can diagram any task, from doubling a number to performing brain surgery. You can diagram each structure with a specific configuration of flowchart symbols.

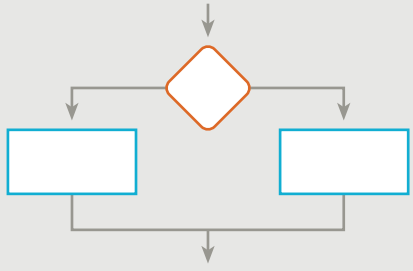
The first of these structures is a sequence, as shown in Figure 2-3. With a **sequence structure**, you perform an action or task, and then you perform the next action, in order. A sequence can contain any number of tasks, but there is no chance to branch off and skip any of the tasks. Once you start a series of actions in a sequence, you must continue step-by-step until the sequence ends.

FIGURE 2-3: SEQUENCE STRUCTURE



The second structure is called a **selection structure** or **decision structure**, as shown in Figure 2-4. With this structure, you ask a question, and, depending on the answer, you take one of two courses of action. Then, no matter which path you follow, you continue with the next task.

FIGURE 2-4: SELECTION STRUCTURE



Some people call the selection structure an **if-then-else** because it fits the following statement:

```
if someCondition is true then
   do oneProcess
else
   do theOtherProcess
```

For example, while cooking you may decide the following:

```
if we have brownSugar then
   use brownSugar
else
   use whiteSugar
```

Similarly, a payroll program might include a statement such as:

```
if hoursWorked is more than 40 then
   calculate regularPay and overtimePay
else
   calculate regularPay
```

The previous examples can also be called **dual-alternative ifs**, because they contain two alternatives—the action taken when the tested condition is true and the action taken when it is false. Note that it is perfectly correct for one branch of the selection to be a "do nothing" branch. For example:

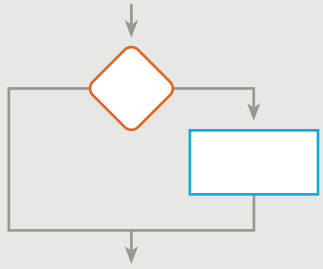
```
if it is raining then
   take anUmbrella
```

or

```
if employee belongs to dentalPlan then
   deduct $40 from employeeGrossPay
```

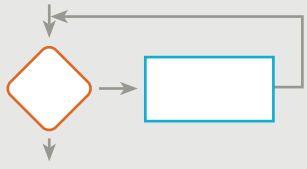
The previous examples are **single-alternative ifs**, and a diagram of their structure is shown in Figure 2-5. In these cases, you don't take any special action if it is not raining or if the employee does not belong to the dental plan. The case where nothing is done is often called the **null case**.

FIGURE 2-5: SINGLE-ALTERNATIVE DECISION STRUCTURE



The third structure, shown in Figure 2-6, is a loop. In a **loop structure**, you continue to repeat actions based on the answer to a question. In the most common type of loop, you first ask a question; if the answer requires an action, you perform the action and ask the original question again. If the answer requires that the action be taken again, you take the action and then ask the original question again. This continues until the answer to the question is such that the action is no longer required; then you exit the structure. You may hear programmers refer to looping as **repetition** or **iteration**.

FIGURE 2-6: LOOP STRUCTURE



Some programmers call this structure a **while...do**, or more simply, a **while** loop, because it fits the following statement:

```
while testCondition continues to be true
   do someProcess
```

You encounter examples of looping every day, as in:

```
while you continue to beHungry
   take anotherBiteOfFood
```

or

```
while unreadPages remain in the readingAssignment
   read another unreadPage
```

In a business program, you might write:

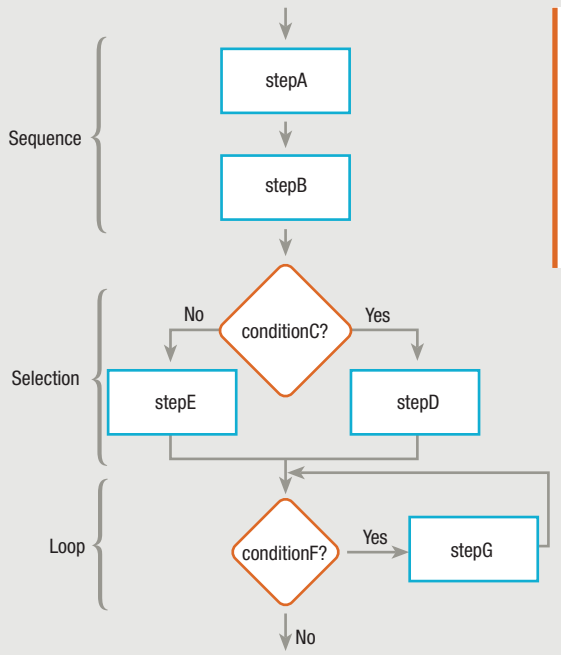
```
while quantityInInventory remains low
   continue to orderItems
```

or

```
while there are more retailPrices to be discounted
   compute a discount
```

All logic problems can be solved using only these three structures—sequence, selection, and loop. The three structures, of course, can be combined in an infinite number of ways. For example, you can have a sequence of tasks followed by a selection, or a loop followed by a sequence. Attaching structures end-to-end is called **stacking** structures. For example, Figure 2-7 shows a structured flowchart achieved by stacking structures, and shows pseudocode that might follow that flowchart logic.

FIGURE 2-7: STRUCTURED FLOWCHART AND PSEUDOCODE




```
do stepA
do stepB
if conditionC is true then
     do stepD
else
     do stepE
endif
while conditionF is true
     do stepG
endwhile
```

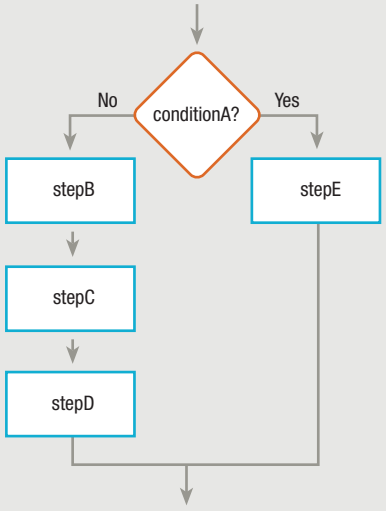
The pseudocode in Figure 2-7 shows two end-structure statements—`endif` and `endwhile`. You can use an `endif` statement to clearly show where the actions that depend on a decision end. The instruction that follows `if` occurs when its tested condition is true, the instruction that follows `else` occurs when the tested condition is false, and the instruction that follows `endif` occurs in either case—it is not dependent on the `if` statement at all. In other words, statements beyond the `endif` statement are "outside" the decision structure. Similarly, you use an `endwhile`

statement to show where a loop structure ends. In Figure 2-7, while `conditionF` continues to be true, `stepG` continues to execute. If any statements followed the `endwhile` statement, they would be outside of, and not a part of, the loop.

TIP Whether you are drawing a flowchart or writing pseudocode, you can use either of the following pairs to represent decision outcomes: yes and no or true and false. This book follows the convention of using yes and no in flowchart diagrams and true and false in pseudocode.

Besides stacking structures, you can replace any individual tasks or steps in a structured flowchart diagram or pseudocode segment with additional structures. In other words, any sequence, selection, or loop can contain other sequences, selections, or loops. For example, you can have a sequence of three tasks on one side of a selection, as shown in Figure 2-8. Placing a structure within another structure is called **nesting** the structures.

FIGURE 2-8: FLOWCHART AND PSEUDOCODE SHOWING A SEQUENCE NESTED WITHIN A SELECTION

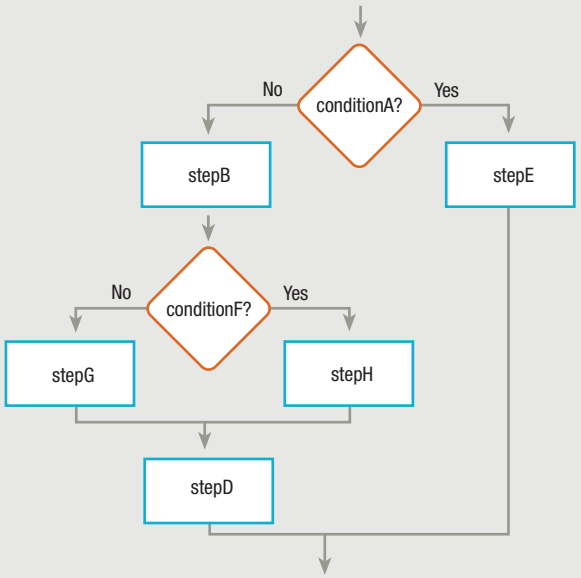



```
if conditionA is true then
     do stepE
else
     do stepB
     do stepC
     do stepD
endif
```

When you write the pseudocode for the logic shown in Figure 2-8, the convention is to indent all statements that depend on one branch of the decision, as shown in the pseudocode. The indentation and the `endif` statement both show that all three statements (`do stepB`, `do stepC`, and `do stepD`) must execute if `conditionA` is not true. The three statements constitute a **block**, or a group of statements that execute as a single unit.

In place of one of the steps in the sequence in Figure 2-8, you can insert a selection. In Figure 2-9, the process named `stepC` has been replaced with a selection structure that begins with a test of the condition named `conditionF`.

FIGURE 2-9: SELECTION IN A SEQUENCE WITHIN A SELECTION



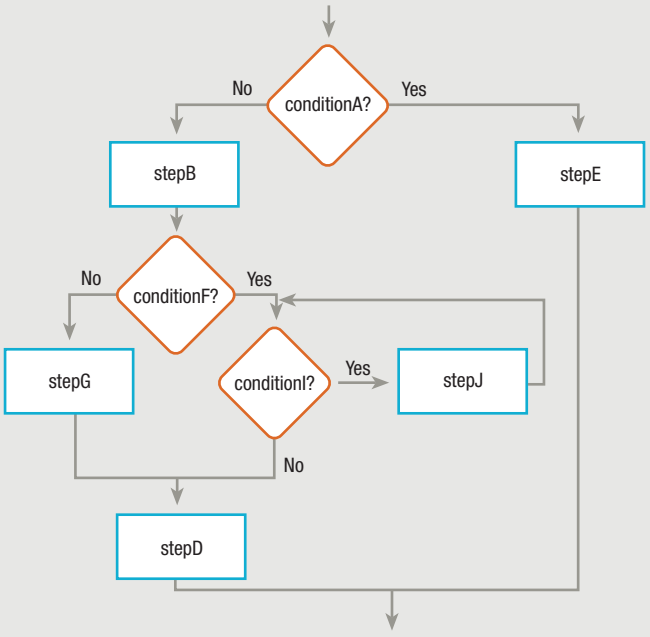

```
if conditionA is true then
     do stepE
else
     do stepB
     if conditionF is true then
          do stepH
     else
          do stepG
     endif
     do stepD
endif
```

In the pseudocode shown in Figure 2-9, notice that `do stepB`, `if conditionF is true then`, `else`, `endif`, and `do stepD` all align vertically with each other. This shows that they are all "on the same level." If you look at the same problem flowcharted in Figure 2-9, you see that you could draw a vertical line through the symbols containing `stepB`, `conditionF`, and `stepD`. The flowchart and the pseudocode represent exactly the same logic. The `stepH` and `stepG` processes, on the other hand, are one level "down"; they are dependent on the answer to the `conditionF` question. Therefore, the `do stepH` and `do stepG` statements are indented one additional level in the pseudocode.

Also notice that the pseudocode in Figure 2-9 has two `endif` statements. Each is aligned to correspond to an `if`. An `endif` always partners with the most recent `if` that does not already have an `endif` partner, and an `endif` should always align vertically with its `if` partner.

In place of `do stepH` on one side of the new selection in Figure 2-9, you can insert a loop. This loop, based on `conditionI`, appears inside the selection that is within the sequence that constitutes the "No" side of the original `conditionA` selection. In the pseudocode in Figure 2-10, notice that the `while` aligns with the `endwhile`, and that the entire `while` structure is indented within the true ("Yes") half of the `if` structure that begins with the decision based on `conditionF`. The indentation used in the pseudocode reflects the logic you can see laid out graphically in the flowchart.

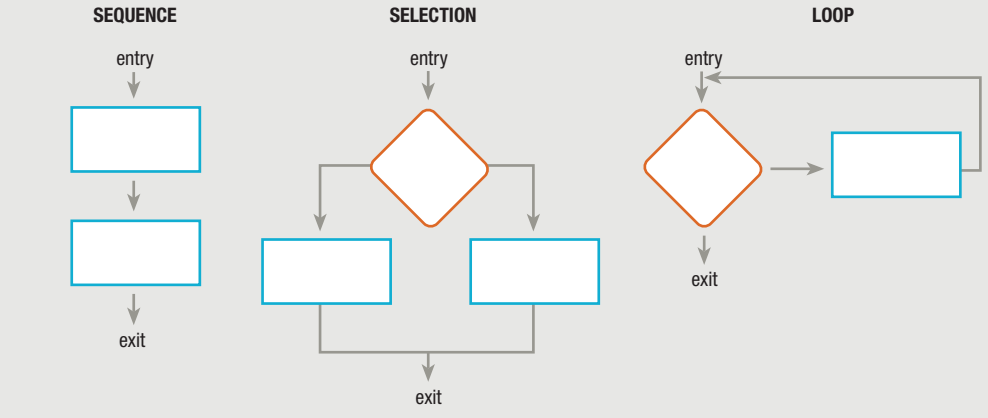
FIGURE 2-10: FLOWCHART AND PSEUDOCODE FOR LOOP WITHIN SELECTION WITHIN SEQUENCE WITHIN SELECTION




```
if conditionA is true then
     do stepE
else
     do stepB
     if conditionF is true then
          while conditionI is true
               do stepJ
          endwhile
     else
          do stepG
     endif
     do stepD
endif
```

The combinations are endless, but each of a structured program's segments is a sequence, a selection, or a loop. The three structures are shown together in Figure 2-11. Notice that each structure has one entry and one exit point. One structure can attach to another only at one of these points.

FIGURE 2-11: THE THREE STRUCTURES



TIP

Try to imagine physically picking up any of the three structures using the "handles" marked entry and exit. These are the spots at which you could connect a structure to any of the others. Similarly, any complete structure, from its entry point to its exit point, can be inserted within the process symbol of any other structure.

In summary, a structured program has the following characteristics:

- A structured program includes only combinations of the three basic structures—sequence, selection, and loop. Any structured program might contain one, two, or all three types of structures.
- Structures can be stacked or connected to one another only at their entry or exit points.
- Any structure can be nested within another structure.

TIP

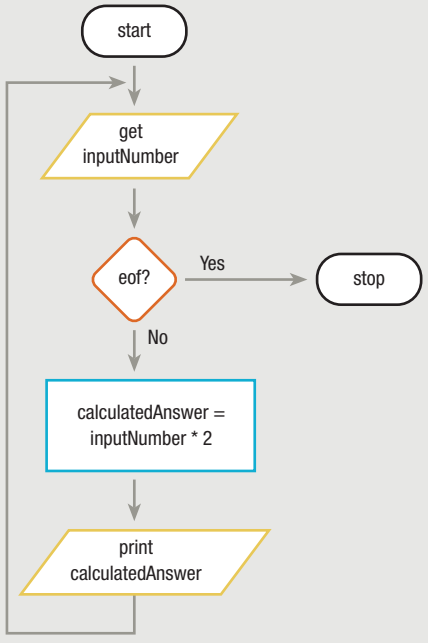
A structured program is never required to contain examples of all three structures; a structured program might contain only one or two of them. For example, many simple programs contain only a sequence of several tasks that execute from start to finish without any needed selections or loops. As another example, a program might display a series of numbers, looping to do so, but never making any decisions about the numbers.

USING THE PRIMING READ

For a program to be structured and work the way you want it to, sometimes you need to add extra steps. The priming read is one kind of added step. A **priming read** or **priming input** is the statement that reads the first input data record. If a program will read 100 data records, you read the first data record in a statement that is separate from the other 99. You must do this to keep the program structured.

At the end of Chapter 1, you read about a program like the one in Figure 2-12. The program gets a number and checks for the end-of-file condition. If it is not the end of file, then the number is doubled, the answer is printed, and the next number is input.

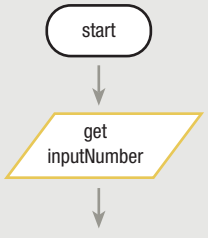
FIGURE 2-12: UNSTRUCTURED FLOWCHART OF A NUMBER-DOUBLING PROGRAM



Is the program represented by Figure 2-12 structured? At first, it might be hard to tell. The three allowed structures were illustrated in Figure 2-11.

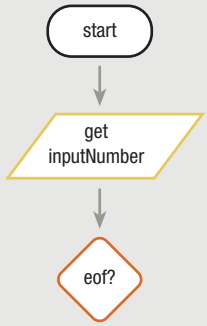
The flowchart in Figure 2-12 does not look exactly like any of the three shapes shown in Figure 2-11. However, because you may stack and nest structures while retaining overall structure, it might be difficult to determine whether a flowchart as a whole is structured. It's easiest to analyze the flowchart in Figure 2-12 one step at a time. The beginning of the flowchart looks like Figure 2-13.

FIGURE 2-13: BEGINNING OF A NUMBER-DOUBLING FLOWCHART



Is this portion of the flowchart structured? Yes, it's a sequence. (Even a single task can be a sequence—it's just a brief sequence.) Adding the next piece of the flowchart looks like Figure 2-14.

FIGURE 2-14: NUMBER-DOUBLING FLOWCHART



The sequence is finished; either a selection or a loop is starting. You might not know which one, but you do know the sequence is not continuing, because sequences can't contain questions. With a sequence, each task or step must follow without any opportunity to branch off. Therefore, which type of structure starts with the question in Figure 2-14? Is it a selection or a loop?

With a selection structure, the logic goes in one of two directions after the question, and then the flow comes back together; the question is not asked a second time. However, in a loop, if the answer to the question results in the loop being entered and the loop statements executing, then the logic returns to the question that started the loop; when the body of a loop executes, the question that controls the loop is always asked again.

In the number-doubling problem in the original Figure 2-12, if it is not `eof` (that is, if the end-of-file condition is not met), then some math is done, an answer is printed, a new number is obtained, and the `eof` question is asked again. In other words, while the answer to the `eof` question continues to be *no*, eventually the logic will return to the `eof` question. (Another way to phrase this is that while it continues to be true that `eof` has not yet been reached, the logic keeps returning to the same question.) Therefore, the number-doubling problem contains a structure beginning with the `eof` question that is more like the beginning of a loop than it is like a selection.

The number-doubling problem *does* contain a loop, but it's not a structured loop. In a structured loop, the rules are:

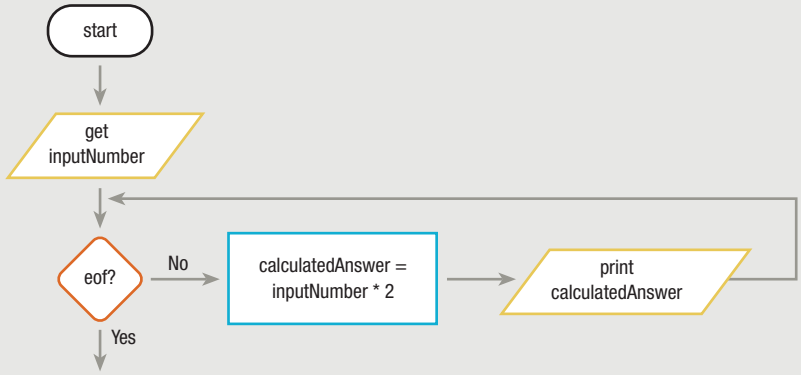
1. You ask a question.
2. If the answer indicates you should take some action or perform a procedure, then you do so.
3. If you perform the procedure, then you must go right back to repeat the question.

The flowchart in Figure 2-12 asks a question; if the answer is *no* (that is, while it is true that the `eof` condition has not been met), then the program performs two tasks: it does the arithmetic and it prints the results. Doing two things is acceptable because two tasks with no possible branching constitute a sequence, and it is fine to nest a structure within another structure. However, when the sequence ends, the logic doesn't flow right back to the question. Instead, it goes

above the question to get another number. For the loop in Figure 2-12 to be a structured loop, the logic must return to the `eof` question when the embedded sequence ends.

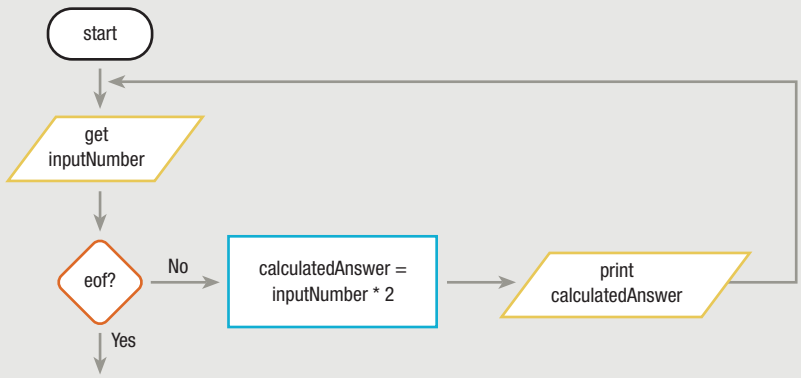
The flowchart in Figure 2-15 shows the flow of logic returning to the `eof` immediately after the sequence. Figure 2-15 shows a structured flowchart, but the flowchart has one major flaw—it doesn't do the job of continuously doubling different numbers.

FIGURE 2-15: STRUCTURED, BUT NONFUNCTIONAL, FLOWCHART OF NUMBER-DOUBLING PROBLEM



Follow the flowchart in Figure 2-15 through a typical program run. Suppose when the program starts, the user enters a 9 for the value of `inputNumber`. That's not `eof`, so the number doubles, and 18 prints out as the `calculatedAnswer`. Then the question `eof?` is asked again. It can't be `eof` because a new value representing the sentinel (ending) value can't be entered. The logic never returns to the `get inputNumber` task, so the value of `inputNumber` never changes. Therefore, 9 doubles again and the answer 18 prints again. It's still not `eof`, so the same steps are repeated. This goes on *forever*, with the answer 18 printing repeatedly. The program logic shown in Figure 2-15 is structured, but it doesn't work as intended; the program in Figure 2-16 works, but it isn't structured!

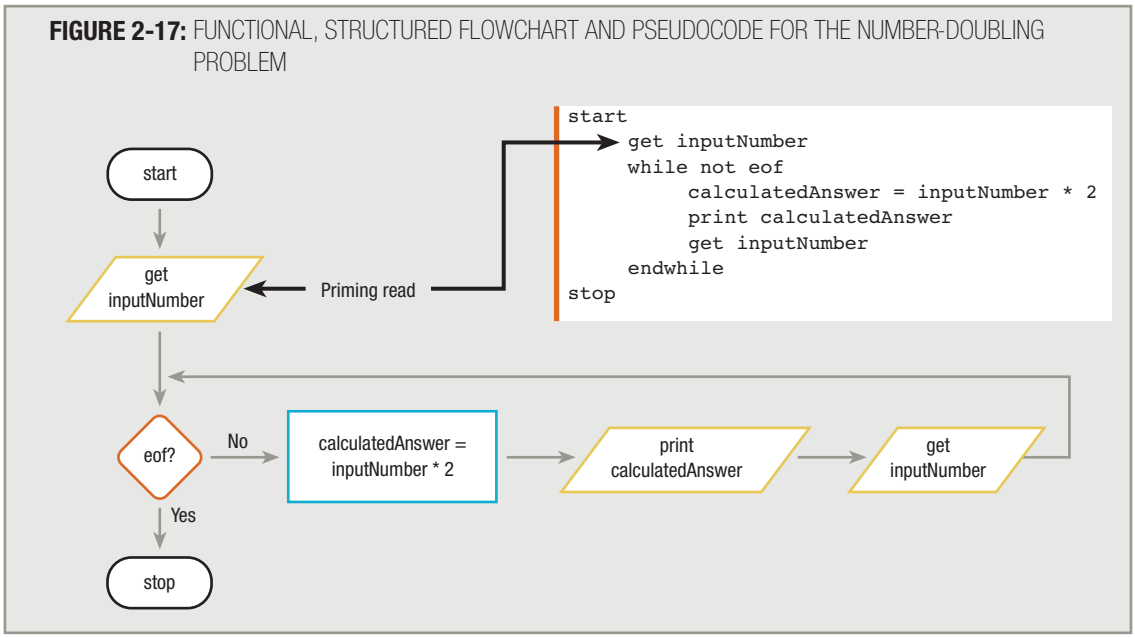
FIGURE 2-16: FUNCTIONAL BUT NONSTRUCTURED FLOWCHART



TIP The loop in Figure 2-16 is not structured because in a structured loop, after the tasks execute within the loop, the flow of logic must return directly to the loop-controlling question. In Figure 2-16, the logic does not return to the loop-controlling question; instead, it goes "too high" outside the loop to repeat the `get inputNumber` task.

How can the number-doubling problem be both structured and work as intended? Often, for a program to be structured, you must add something extra. In this case, it's an extra `get inputNumber` step. Consider the solution in Figure 2-17; it's structured *and* it does what it's supposed to do. The program logic illustrated in Figure 2-17 contains a sequence and a loop. The loop contains another sequence.

FIGURE 2-17: FUNCTIONAL, STRUCTURED FLOWCHART AND PSEUDOCODE FOR THE NUMBER-DOUBLING PROBLEM



The additional `get inputNumber` step is typical in structured programs. The first of the two input steps is the priming input, or priming read. The term *priming* comes from the fact that the read is first, or *primary* (what gets the process going, as in "priming the pump"). The purpose of the priming read step is to control the upcoming loop that begins with the `eof` question. The last element within the structured loop gets the next, and all subsequent, input values. This is also typical in structured loops—the last step executed within the loop alters the condition tested in the question that begins the loop, which in this case is the `eof` question.

As an additional way to determine whether a flowchart segment is structured, you can try to write pseudocode for it. Examine the unstructured flowchart in Figure 2-12 again. To write pseudocode for it, you would begin with the following:

```
start
     get inputNumber
```

When you encounter the `eof` question in the flowchart, you know that either a selection or loop structure should begin. Because you return to a location higher in the flowchart when the answer to the `eof` question is *no* (that is, while the `not eof` condition continues to be *true*), you know that a loop is beginning. So you continue to write the pseudocode as follows:

```
start
     get inputNumber
     while not eof
          calculatedAnswer = inputNumber * 2
          print calculatedAnswer
```

Continuing, the step after `print calculatedAnswer` is `get inputNumber`. This ends the `while` loop that began with the `eof` question. So the pseudocode becomes:

```
start
     get inputNumber
     while not eof
          calculatedAnswer = inputNumber * 2
          print calculatedAnswer
          get inputNumber
     endwhile
stop
```

This pseudocode is identical to the pseudocode in Figure 2-17 and now matches the flowchart in the same figure. It does not match the flowchart in Figure 2-12, because that flowchart contains only one `get inputNumber` step. Creating the pseudocode correctly using the `while` statement requires you to repeat the `get inputNumber` statement. The structured pseudocode makes use of a priming read and forces the logic to become structured—a sequence followed by a loop that contains a sequence of three statements.

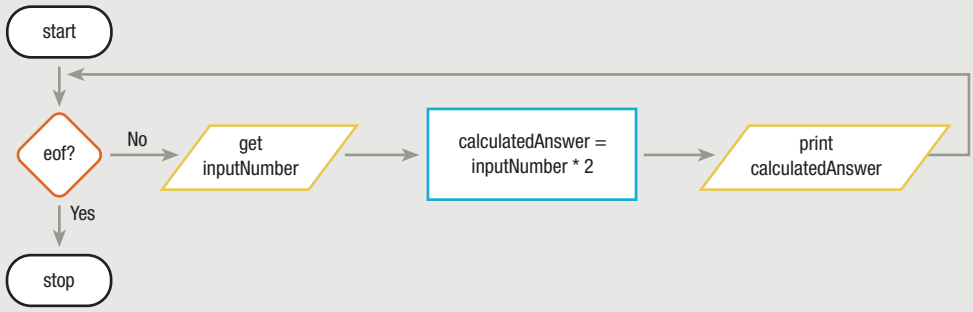
TIP

Years ago, programmers could avoid using structure by inserting a "go to" statement into their pseudocode. A "go to" statement would say something like "after print answer, go to the first get number box", and would be the equivalent of drawing an arrow starting after "print answer" and pointing directly to the first "get number" box in the flowchart. Because "go to" statements cause spaghetti code, they are not allowed in structured programming. Some programmers call structured programming "goto-less" programming.

Figure 2-18 shows another way you might attempt to draw the logic for the number-doubling program. At first glance, the figure might seem to show an acceptable solution to the problem—it is structured, containing a single loop with a sequence of three steps within it, and it appears to eliminate the need for the priming input statement. When the program starts, the `eof` question is asked. The answer is *no*, so the program gets an input number, doubles it, and prints it. Then, if it is still not `eof`, the program gets another number, doubles it, and prints it. The program continues until `eof` is encountered when getting input. The last time the `get inputNumber` statement executes, it encounters `eof`, but the program does not stop—instead, it calculates and

prints one last time. This last output is extraneous—the `eof` value should not be doubled and printed. As a general rule, an `eof` question should always come immediately after an input statement. Therefore, the best solution to the number-doubling problem remains the one shown in Figure 2-17—the solution containing the priming input statement.

FIGURE 2-18: STRUCTURED BUT INCORRECT SOLUTION TO THE NUMBER-DOUBLING PROBLEM



TIP

A few languages do not require the priming read. For example, programs written using the Visual Basic programming language can "look ahead" to determine whether the end of file will be reached on the next input record. However, most programming languages cannot predict the end of file until an actual read operation is performed, and they require a priming read to properly handle file data.

UNDERSTANDING THE REASONS FOR STRUCTURE

At this point, you may very well be saying, "I liked the original number-doubling program just fine. I could follow it. Also, the first program had one less step in it, so it was less work. Who cares if a program is structured?"

Until you have some programming experience, it is difficult to appreciate the reasons for using only the three structures—sequence, selection, and loop. However, staying with these three structures is better for the following reasons:

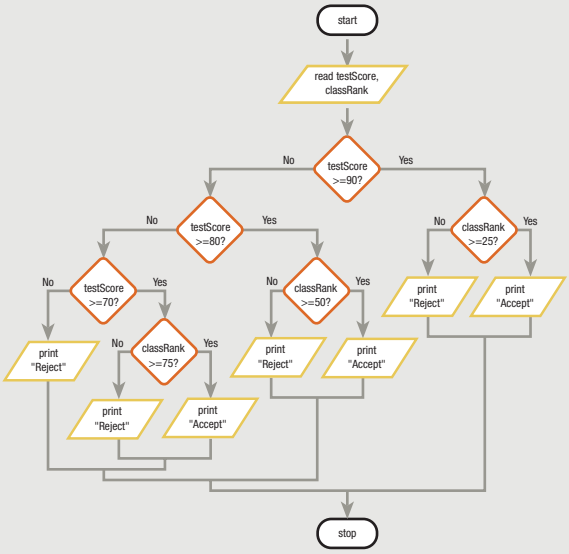
- *Clarity*—The number-doubling program is a small program. As programs get bigger, they get more confusing if they're not structured.
- *Professionalism*—All other programmers (and programming teachers you might encounter) expect your programs to be structured. It's the way things are done professionally.
- *Efficiency*—Most newer computer languages are structured languages with syntax that lets you deal efficiently with sequence, selection, and looping. Older languages, such as assembly languages, COBOL, and RPG, were developed before the principles of structured programming were discovered. However, even programs that use those older languages can be written in a structured form, and structured programming is expected on the job today. Newer languages such as C#, C++, and Java enforce structure by their syntax.

- *Maintenance*—You, as well as other programmers, will find it easier to modify and maintain structured programs as changes are required in the future.
- *Modularity*—Structured programs can be easily broken down into routines or modules that can be assigned to any number of programmers. The routines are then pieced back together like modular furniture at each routine's single entry or exit point. Additionally, often a module can be used in multiple programs, saving development time in the new project.

Most programs that you purchase are huge, consisting of thousands or millions of statements. If you've worked with a word-processing program or spreadsheet, think of the number of menu options and keystroke combinations available to the user. Such programs are not the work of one programmer. The modular nature of structured programs means that work can be divided among many programmers; then the modules can be connected, and a large program can be developed much more quickly. Money is often a motivating factor—the faster you write a program and make it available for use, the sooner it begins making money for the developer.

Consider the college admissions program from the beginning of this chapter. It has been rewritten in structured form in Figure 2-19 and is easier to follow now. Figure 2-19 also shows structured pseudocode for the same problem.

FIGURE 2-19: FLOWCHART AND PSEUDOCODE OF STRUCTURED COLLEGE ADMISSION PROGRAM




```
start
     read testScore, classRank
     if testScore >= 90 then
          if classRank >= 25 then
               print "Accept"
          else
               print "Reject"
          endif
     else
          if testScore >= 80 then
               if classRank >= 50 then
                    print "Accept"
               else
                    print "Reject"
               endif
          else
               if testScore >= 70 then
                    if classRank >= 75 then
                         print "Accept"
                    else
                         print "Reject"
                    endif
               else
                    print "Reject"
               endif
          endif
     endif
stop
```

```
start
    read testScore, classRank
    if testScore >= 90 then
            if classRank >= 25 then
                    print "Accept"
            else
                    print "Reject"
            endif
    else
            if testScore >= 80 then
                    if classRank >= 50 then
                            print "Accept"
                    else
                            print "Reject"
                    endif
            else
                    if testScore >= 70 then
                            if classRank >= 75 then
                                    print "Accept"
                            else
                                    print "Reject"
                            endif
                    else
                            print "Reject"
                    endif
            endif
    endif
stop
```

TIP Don't be alarmed if it is difficult for you to follow the many nested `if`s within the pseudocode in Figure 2-19. After you study the selection process in more detail, reading this type of pseudocode will become much easier for you.

In the lower portion of Figure 2-19, the pseudocode is repeated using colored backgrounds to help you identify the indentations that match, distinguishing the different levels of the nested structures.

TIP As you examine Figure 2-19, notice that the bottoms of the three `testScore` decision structures join at the bottom of the diagram. These three joinings correspond to the last three `endif` statements in the pseudocode.

RECOGNIZING STRUCTURE

Any set of instructions can be expressed in a structured format. If you can teach someone how to perform any ordinary activity, then you can express it in a structured way. For example, suppose you wanted to teach a child how to play Rock, Paper, Scissors. In this game, two players simultaneously show each other one hand, in one of three positions—clenched in a fist, representing a rock; opened flat, representing a piece of paper; or with two fingers extended in a V, representing scissors. The goal is to guess which hand position your opponent might show, so that you can show the one that beats it. The rules are that a flat hand beats a fist (because a piece of paper can cover a rock), a fist beats a hand with two extended fingers (because a rock can smash a pair of scissors), and a hand with two extended fingers beats a flat hand (because scissors can cut paper). Figure 2-20 shows the pseudocode for the game.

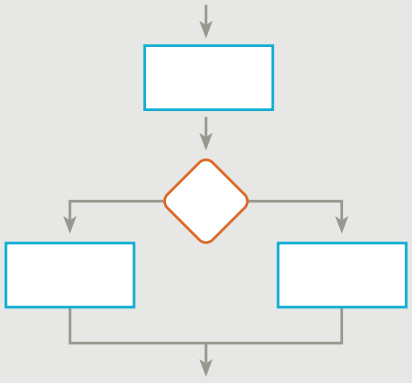
Figure 2-20 also shows a fairly complicated set of statements. Its purpose is not to teach you how to play a game (although you could learn how to play by following the logic), but rather to convince you that any task to which you can apply rules can be expressed logically using only combinations of sequence, selection, and looping. In this example, a game continues while a friend agrees to play, and within that loop, several decisions must be made in order to determine the winner.

FIGURE 2-20: PSEUDOCODE FOR THE ROCK, PAPER, SCISSORS GAME

```
start
    ask friend to play a game of Rock, Paper, Scissors
    while answer is yes
        extend yourHand and myHand
        if yourHand = "Paper" then
            if myHand = "Rock" then
                winner = yourHand
            else
                if myHand = "Scissors" then
                    winner = myHand
                else
                    winner = tie
                endif
            endif
        else
            if yourHand = "Scissors" then
                if myHand = "Rock" then
                    winner = myHand
                else
                    if myHand = "Paper" then
                        winner = yourHand
                    else
                        winner = tie
                    endif
                endif
            else
                if myHand = "Rock" then
                    winner = tie
                else
                    if myHand = "Paper" then
                        winner = myHand
                    else
                        winner = yourHand
                    endif
                endif
            endif
        endif
        ask friend to play a game of Rock, Paper, Scissors
    endwhile
stop
```

When you are just learning about structured program design, it is difficult to detect whether a flowchart of a program's logic is structured. For example, is the flowchart segment in Figure 2-21 structured?

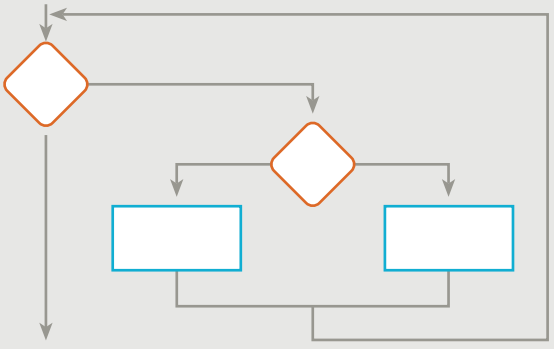
FIGURE 2-21: EXAMPLE 1



Yes, it is. It has a sequence and a selection structure.

Is the flowchart segment in Figure 2-22 structured?

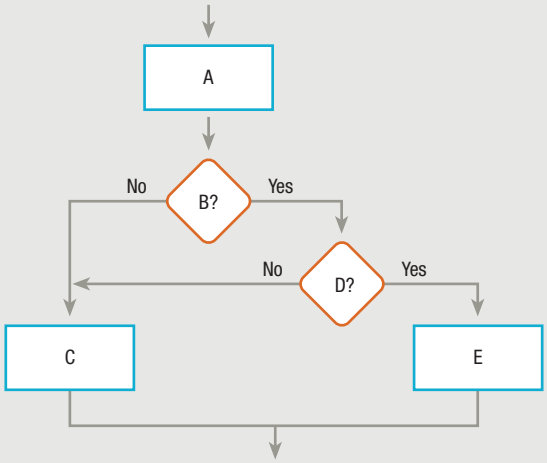
FIGURE 2-22: EXAMPLE 2



Yes, it is. It has a loop, and within the loop is a selection.

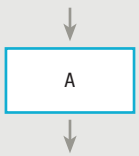
Is the flowchart segment in Figure 2-23 structured? (The symbols are lettered so you can better follow the discussion.)

FIGURE 2-23: EXAMPLE 3



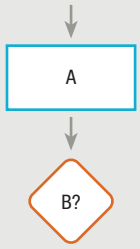
No, it isn't; it is not constructed from the three basic structures. One way to straighten out a flowchart segment that isn't structured is to use what you can call the "spaghetti bowl" method; that is, picture the flowchart as a bowl of spaghetti that you must untangle. Imagine you can grab one piece of pasta at the top of the bowl, and start pulling. As you "pull" each symbol out of the tangled mess, you can untangle the separate paths until the entire segment is structured. For example, with the diagram in Figure 2-23, if you start pulling at the top, you encounter a procedure box, labeled A. (See Figure 2-24.)

FIGURE 2-24: UNTANGLING EXAMPLE 3, FIRST STEP



A single process like A is part of an acceptable structure—it constitutes at least the beginning of a sequence structure. Imagine you continue pulling symbols from the tangled segment. The next item in the flowchart is a question that tests a condition labeled B, as you can see in Figure 2-25.

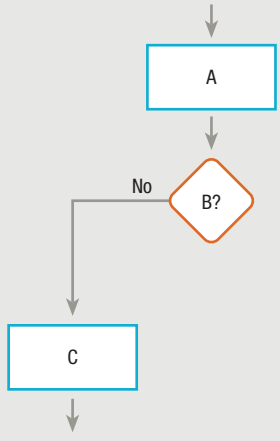
FIGURE 2-25: UNTANGLING EXAMPLE 3, SECOND STEP



At this point, you know the sequence that started with A has ended. Sequences never have decisions in them, so the sequence is finished; either a selection or a loop is beginning. A loop must return to the question at some later point. You can see from the original logic in Figure 2-23 that whether the answer to B is yes or no, the logic never returns to B. Therefore, B begins a selection structure, not a loop structure.

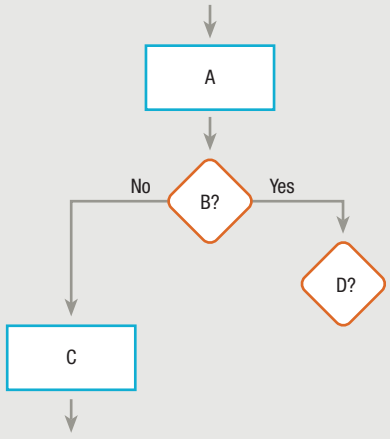
To continue detangling the logic, you (imaginarily) pull up on the flowline that emerges from the left side (the "No" side) of Question B. You encounter C, as shown in Figure 2-26. When you continue beyond C, you reach the end of the flowchart.

FIGURE 2-26: UNTANGLING EXAMPLE 3, THIRD STEP



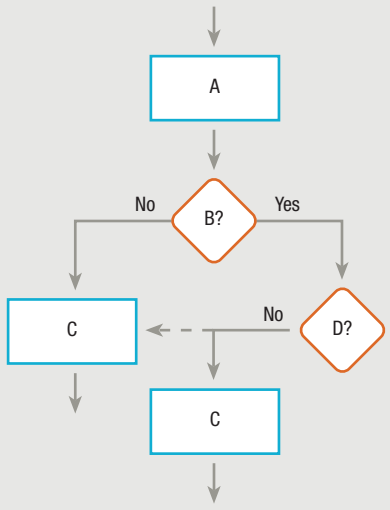
Now you can turn your attention to the "Yes" side (the right side) of the condition tested in B. When you pull up on the right side, you encounter Question D. (See Figure 2-27.)

FIGURE 2-27: UNTANGLING EXAMPLE 3, FOURTH STEP



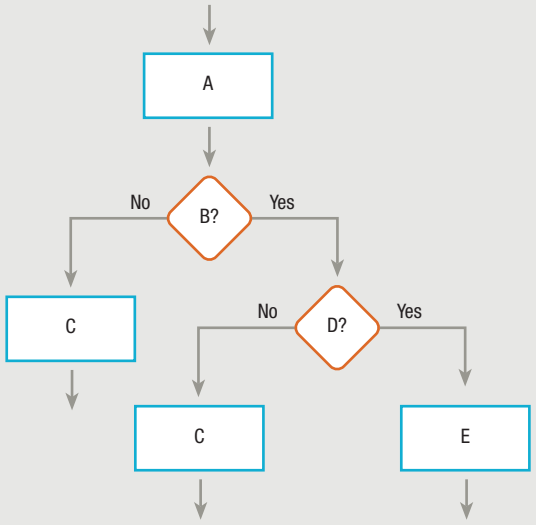
In Figure 2-23, follow the line on the left side of Question D. The line extending from the selection is attached to a task outside the selection. The line emerging from the left side of selection D is attached to Step C. You might say the D selection is becoming entangled with the B selection, so you must untangle the structures by repeating the step that is causing the tangle. (In this example, you repeat Step C to untangle it from the other usage of C.) Continue pulling on the flowline that emerges from Step C until you reach the end of the program segment, as shown in Figure 2-28.

FIGURE 2-28: UNTANGLING EXAMPLE 3, FIFTH STEP



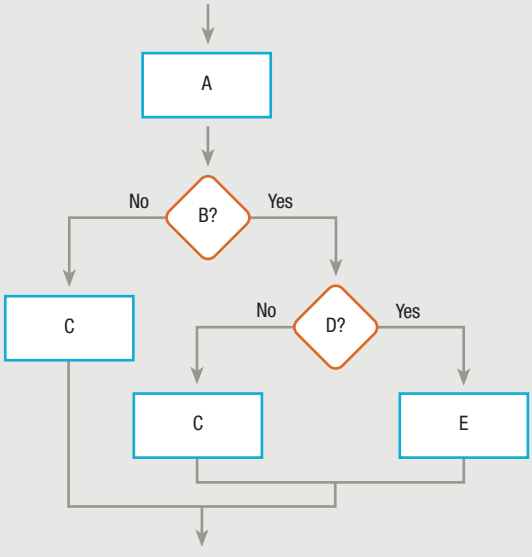
Now pull on the right side of Question D. Process E pops up, as shown in Figure 2-29; then you reach the end.

FIGURE 2-29: UNTANGLING EXAMPLE 3, SIXTH STEP



At this point, the untangled flowchart has three loose ends. The loose ends of Question D can be brought together to form a selection structure; then the loose ends of Question B can be brought together to form another selection structure. The result is the flowchart shown in Figure 2-30. The entire flowchart segment is structured—it has a sequence (A) followed by a selection inside a selection.

FIGURE 2-30: FINISHED FLOWCHART AND PSEUDOCODE FOR UNTANGLING EXAMPLE 3




```
do A
if B is true then
     if D is true then
          do E
     else
          do C
     endif
else
     do C
endif
```

TIP If you want to try structuring a very difficult example of an unstructured program, see Appendix A.

THREE SPECIAL STRUCTURES—CASE, DO WHILE, AND DO UNTIL

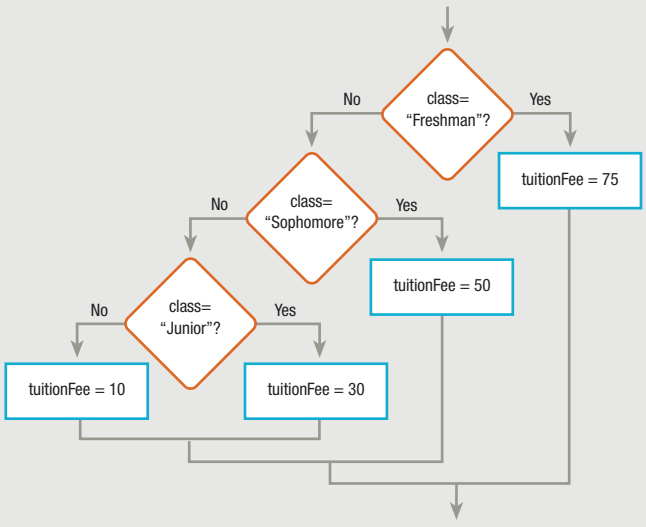
TIP You can skip this section for now without any loss in continuity. Your instructor may prefer to discuss the case structure with the Decision chapter (Chapter 5), and the do-while and do-until loops with the Looping chapter (Chapter 6).

You can solve any logic problem you might encounter using only the three structures: sequence, selection, and loop. However, many programming languages allow three more structures: the case structure and the do-while and do-until loops. These structures are never *needed* to solve any problem—you can always use a series of selections instead of the case structure, and you can always use a sequence plus a while loop in place of the do-while or do-until loops. However, sometimes these additional structures are convenient. Programmers consider them all to be acceptable, legal structures.

THE CASE STRUCTURE

You can use the **case structure** when there are several distinct possible values for a single variable you are testing, and each value requires a different course of action. Suppose you administer a school at which tuition is $75, $50, $30, or $10 per credit hour, depending on whether a student is a freshman, sophomore, junior, or senior. The structured flowchart and pseudocode in Figure 2-31 show a series of decisions that assigns the correct tuition to a student.

FIGURE 2-31: FLOWCHART AND PSEUDOCODE OF TUITION DECISIONS




```
if class = "Freshman" then
     tuitionFee = 75
else
     if class = "Sophomore" then
          tuitionFee = 50
     else
          if class = "Junior" then
               tuitionFee = 30
          else
               tuitionFee = 10
          endif
     endif
endif
```

TIP

The indentation in the pseudocode in Figure 2-31 reflects the nested nature of the decisions, as illustrated in the flowchart. For clarity, some programmers might prefer to write the pseudocode as follows:

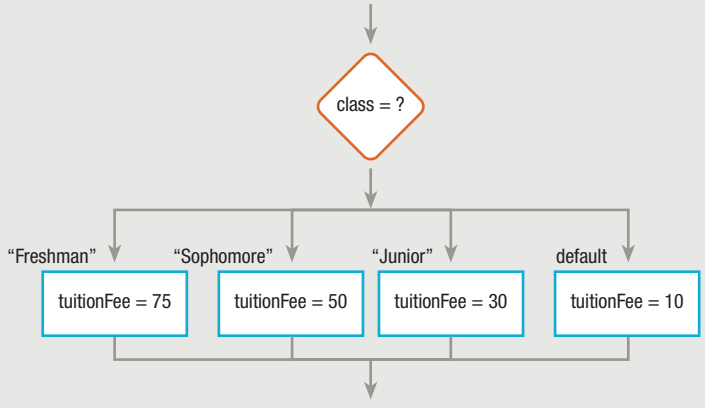
```
if class = "Freshman" then
   tuitionFee = 75
else if class = "Sophomore" then
   tuitionFee = 50
else if class = "Junior" then
   tuitionFee = 30
endif
```

This style, with `else` and the next `if` on the same line and a single `endif` at the end, is often preferred by Visual Basic programmers because it resembles a style they use in their programs. However, this book will use the style shown in Figure 2-31, with each `endif` aligned with its corresponding `if` statement.

The logic shown in Figure 2-31 is absolutely correct and completely structured. The `class="Junior"` selection structure is contained within the `class="Sophomore"` structure, which is contained within the `class="Freshman"` structure. Note that there is no need to ask if a student is a senior, because if a student is not a freshman, sophomore, or junior, it is assumed the student is a senior.

Even though the program segments in Figure 2-31 are correct and structured, many programming languages permit using a case structure, as shown in Figure 2-32. When using the case structure, you test a variable against a series of values, taking appropriate action based on the variable's value. To many, such programs seem easier to read, and the case structure is allowed because the same results *could* be achieved with a series of structured selections (thus making the program structured). That is, if the first program is structured and the second one reflects the first one point by point, then the second one must be structured also.

FIGURE 2-32: FLOWCHART AND PSEUDOCODE OF CASE STRUCTURE




```
case based on class
     case "Freshman"
          tuitionFee = 75
     case "Sophomore"
          tuitionFee = 50
     case "Junior"
          tuitionFee = 30
     default
          tuitionFee = 10
endcase
```

TIP The term "default" used in Figure 2-32 means "if none of the other cases were true." Each programming language you learn may use a different syntax for the default case.

Even though a programming language permits you to use the case structure, you should understand that the case structure is just a convenience that might make a flowchart, pseudocode, or actual program code easier to understand at first glance. When you write a series of decisions using the case structure, the computer still makes a series of individual decisions, just as though you had used many if-then-else combinations. In other words, you might prefer looking at the diagram in Figure 2-32 to understand the tuition fees charged by a school, but a computer actually makes the decisions as shown in Figure 2-31—one at a time. When you write your own programs, it is always acceptable to express a complicated decision-making process as a series of individual selections.

TIP You usually use the case structure only when a series of decisions is based on different values stored in a single variable. If multiple variables are tested, then most programmers use a series of decisions.

THE DO-WHILE AND DO-UNTIL LOOPS

Recall that a structured loop (often called a while loop) looks like Figure 2-33. A special-case loop called a do-while or do-until loop looks like Figure 2-34.

FIGURE 2-33: WHILE LOOP

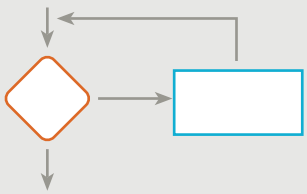
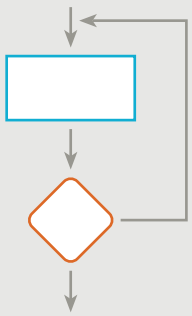


FIGURE 2-34: STRUCTURE OF A DO-WHILE OR DO-UNTIL (POSTTEST) LOOP



An important difference exists between these two structures. In a while loop, you ask a question and, depending on the answer, you might or might not enter the loop to execute the loop's procedure. Conversely, in **do-while** and **do-until loops**, you ensure that the procedure executes at least once; then, depending on the answer to the controlling question, the loop may or may not execute additional times. In a do-while loop, the loop body continues to execute as long as the answer to the controlling question is yes, or true. In a do-until loop, the loop body continues to execute as long as the answer to the controlling question is no, or false; that is, the body executes *until* the controlling question is yes or true.

TIP Notice that the word "do" begins the names of both the do-while and do-until loops. This should remind you that the action you "do" precedes testing the condition.

In a while loop, the question that controls a loop comes at the beginning, or "top," of the loop body. A while loop is also called a **pretest loop** because a condition is tested before entering the loop even once. In a do-while or do-until loop, the question that controls the loop comes at the end, or "bottom," of the loop body. Do-while and do-until loops are also called **posttest loops** because a condition is tested after the loop body has executed.

You encounter examples of do-until looping every day. For example:

```
do
     pay bills
until all bills are paid
```

and

```
do
     wash dishes
until all dishes are washed
```

Similarly, you encounter examples of do-while looping every day. For example:

```
do
     pay bills
while more bills remain to be paid
```

and

```
do
     wash dishes
while more dishes remain to be washed
```

In these examples, the activity (paying bills or washing dishes) must occur at least one time. You ask the question that determines whether you continue only after the activity has been executed at least once. The only difference in these structures is whether the answer to the bottom loop-controlling question must be false for the loop to continue (as in all bills are paid), which is a do-until loop, or true for the loop to continue (as in more bills remain to be paid), which is a do-while loop.

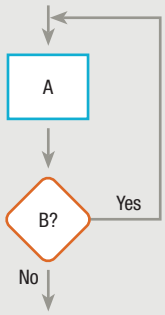
You are never required to use a posttest loop. You can duplicate the same series of actions generated by any posttest loop by creating a sequence followed by a standard, pretest while loop. For example, the following code performs the bill-paying task once, then asks the loop-controlling question at the top of a while loop, in which the action might be performed again:

```
pay bills
while there are more bills to pay
     pay bills
endwhile
```

Consider the flowcharts and pseudocode in Figures 2-35 and 2-36.

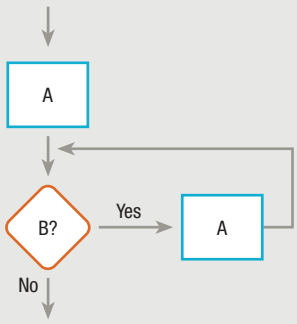
In Figure 2-35, A is done, and then B is asked. If B is yes, then A is done and B is asked again. In Figure 2-36, A is done, and then B is asked. If B is yes, then A is done and B is asked again. In other words, both flowcharts and pseudocode segments do exactly the same thing.

FIGURE 2-35: FLOWCHART AND PSEUDOCODE FOR DO-WHILE LOOP




```
do
     A
while B is true
```

FIGURE 2-36: FLOWCHART AND PSEUDOCODE FOR SEQUENCE FOLLOWED BY WHILE LOOP




```
do A
while B is true
     do A
endwhile
```

Because programmers understand that any posttest loop (do-while or do-until) can be expressed with a sequence followed by a while loop, most languages allow the posttest loop. (Frequently, languages allow one type of posttest loop or the other.) Again, you are never required to use a posttest loop; you can always accomplish the same tasks with a sequence followed by a pretest while loop.

Figure 2-37 shows an unstructured loop. It is neither a while loop (which begins with a decision and, after an action, returns to the decision) nor a do-while or do-until loop (which begins with an action and ends with a decision that might repeat the action). Instead, it begins like a posttest loop (a do-while or a do-until loop), with a process followed by a decision, but one branch of the decision does not repeat the initial process; instead, it performs an additional new action before repeating the initial process. If you need to use the logic shown in Figure 2-37—performing a task, asking a question, and perhaps performing an additional task before looping back to the first process—then the way to make the logic structured is to repeat the initial process within the loop, at the end of the loop. Figure 2-38 shows the same logic as Figure 2-37, but now it is structured logic, with a sequence of two actions occurring within the loop.

Does this diagram look familiar to you? It uses the same technique of repeating a needed step that you saw earlier in this chapter, when you learned the rationale for the priming read.

FIGURE 2-37: UNSTRUCTURED LOOP

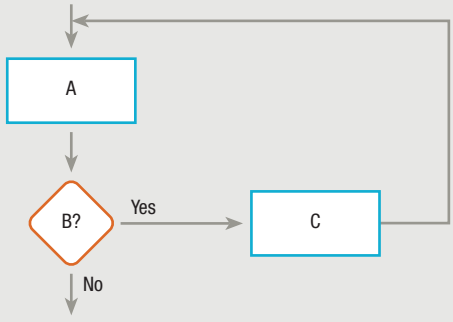
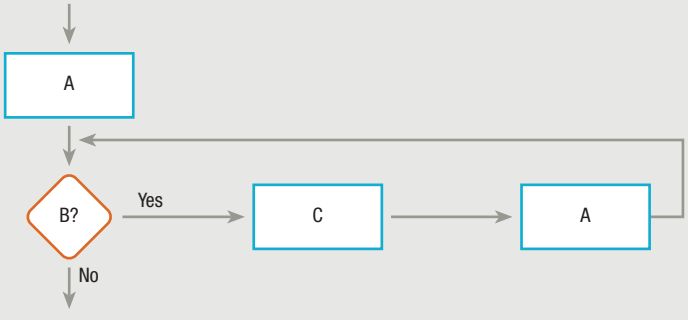


FIGURE 2-38: SEQUENCE AND STRUCTURED LOOP THAT ACCOMPLISH THE SAME TASKS AS FIGURE 2-37



It is difficult for beginning programmers to distinguish among while, do-while, and do-until loops. A while loop asks the question first—for example, while you are hungry, eat. The answer to the question might never be true and the loop body might never execute. A while loop is the only type of loop you ever need in order to solve a problem. You can think of a do-while loop as one that continues to execute while a condition remains true—for example, process records while not end of file is true, or eat food while hungry is true. On the other hand, a do-until loop continues while a condition is false, or, in other words, until the condition becomes true—for example, address envelopes until there are no more envelopes, or eat food until you are full. When you use a do-while or a do-until loop, at least one performance of the action always occurs.

TIP

Especially when you are first mastering structured logic, you might prefer to only use the three basic structures—sequence, selection, and while loop. Every logical problem can be solved using only these three structures, and you can understand all of the examples in the rest of this book using only these three.

CHAPTER SUMMARY

- The popular name for snarled program statements is spaghetti code.
- Clearer programs can be constructed using only three basic structures: sequence, selection, and loop. These three structures can be combined in an infinite number of ways by stacking and nesting them. Each structure has one entry and one exit point; one structure can attach to another only at one of these points.
- A priming read or priming input is the statement that reads the first input data record prior to starting a structured loop. The last step within the loop gets the next, and all subsequent, input values.
- You use structured techniques to promote clarity, professionalism, efficiency, and modularity.
- One way to straighten a flowchart segment that isn't structured is to imagine the flowchart as a bowl of spaghetti that you must untangle.
- You can use a case structure when there are several distinct possible values for a variable you are testing. When you write a series of decisions using the case structure, the computer still makes a series of individual decisions.
- In a pretest while loop, you ask a question and, depending on the answer, you might never enter the loop to execute the loop's body. In a posttest do-while loop (which executes as long as the answer to the controlling question is true) or a posttest do-until loop (which executes as long as the answer to the controlling question is false), you ensure that the loop body executes at least once. You can duplicate the same series of actions generated by any posttest loop by creating a sequence followed by a while loop.

KEY TERMS

Spaghetti code is snarled, unstructured program logic.

A **structure** is a basic unit of programming logic; each structure is a sequence, selection, or loop.

With a **sequence structure**, you perform an action or task, and then you perform the next action, in order. A sequence can contain any number of tasks, but there is no chance to branch off and skip any of the tasks.

With a **selection**, or **decision**, **structure**, you ask a question, and, depending on the answer, you take one of two courses of action. Then, no matter which path you follow, you continue with the next task.

An **if-then-else** is another name for a selection structure.

Dual-alternative ifs define one action to be taken when the tested condition is true, and another action to be taken when it is false.

Single-alternative ifs take action on just one branch of the decision.

The **null case** is the branch of a decision in which no action is taken.

With a **loop structure**, you continue to repeat actions based on the answer to a question.

Repetition and **iteration** are alternate names for a loop structure.

A **while...do**, or more simply, a **while** loop, is a loop in which a process continues while some condition continues to be true.

Attaching structures end-to-end is called **stacking** structures.

Placing a structure within another structure is called **nesting** the structures.

A **block** is a group of statements that execute as a single unit.

A **priming read** or **priming input** is the statement that reads the first input data record prior to starting a structured loop.

You can use the **case structure** when there are several distinct possible values for a single variable you are testing, and each requires a different course of action.

In **do-while** and **do-until loops**, you ensure that a procedure executes at least once; then, depending on the answer to the controlling question, the loop may or may not execute additional times.

A while loop is also called a **pretest loop** because a condition is tested before entering the loop even once.

Do-while and do-until loops are also called **posttest loops** because a condition is tested after the loop body has executed.

REVIEW QUESTIONS

1. **Snarled program logic is called _____ code.**
   a. snake
   b. spaghetti
   c. string
   d. gnarly

2. **A sequence structure can contain _____.**
   a. only one task
   b. exactly three tasks
   c. no more than three tasks
   d. any number of tasks

3. **Which of the following is not another term for a selection structure?**
   a. decision structure
   b. if-then-else structure
   c. loop structure
   d. dual-alternative if structure

4. **The structure in which you ask a question, and, depending on the answer, take some action and then ask the question again, can be called all of the following except _____.**
   a. if-then-else
   b. loop
   c. repetition
   d. iteration

5. **Placing a structure within another structure is called _____ the structures.**

   a. stacking
   b. nesting
   c. building
   d. untangling

6. **Attaching structures end-to-end is called _____.**

   a. stacking
   b. nesting
   c. building
   d. untangling

7. **The statement `if age >= 65 then seniorDiscount = "yes"` is an example of a _____.**

   a. single-alternative if
   b. loop
   c. dual-alternative if
   d. sequence

8. **The statement `while temperature remains below 60, leave the furnace on` is an example of a _____.**

   a. single-alternative if
   b. loop
   c. dual-alternative if
   d. sequence

9. **The statement `if age < 13 then movieTicket = 4.00 else movieTicket = 8.50` is an example of a _____.**

   a. single-alternative if
   b. loop
   c. dual-alternative if
   d. sequence

10. **Which of the following attributes do all three basic structures share?**

    a. Their flowcharts all contain exactly three processing symbols.
    b. They all contain a decision.
    c. They all begin with a process.
    d. They all have one entry and one exit point.

11. **When you read input data in a loop within a program, the input statement that precedes the loop _____.**

    a. is called a priming input
    b. cannot result in `eof`
    c. is the only part of a program allowed to be unstructured
    d. executes hundreds or even thousands of times in most business programs

12. A group of statements that execute as a unit is a _____.

a. cohort
b. family
c. chunk
d. block

13. Which of the following is acceptable in a structured program?

a. placing a sequence within the true half of a dual-alternative decision
b. placing a decision within a loop
c. placing a loop within one of the steps in a sequence
d. All of these are acceptable.

14. Which of the following is not a reason for enforcing structure rules in computer programs?

a. Structured programs are clearer to understand than unstructured ones.
b. Other professional programmers will expect programs to be structured.
c. Structured programs can be broken down into modules easily.
d. Structured programs usually are shorter than unstructured ones.

15. Which of the following is not a benefit of modularizing programs?

a. Modular programs are easier to read and understand than nonmodular ones.
b. Modular components are reusable in other programs.
c. If you use modules, you can ignore the rules of structure.
d. Multiple programmers can work on different modules at the same time.

16. Which of the following is true of structured logic?

a. Any task can be described using some combination of the three structures.
b. You can use structured logic with newer programming languages, such as Java and C#, but not with older ones.
c. Structured programs require that you break the code into easy-to-handle modules that each contain no more than five actions.
d. All of these are true.

17. The structure that you can use when you must make a decision with several possible outcomes, depending on the value of a single variable, is the _____.

a. multiple-alternative if structure
b. case structure
c. do-while structure
d. do-until structure

18. Which type of loop ensures that an action will take place at least one time?

a. a do-until loop
b. a while loop
c. a do-over loop
d. any structured loop

19. A do-until loop can always be converted to _____.

a. a while followed by a sequence
b. a sequence followed by a while
c. a case structure
d. a selection followed by a while

20. Which of the following structures is never required by any program?

a. a while
b. a do-until
c. a selection
d. a sequence

FIND THE BUGS

As you learned in Chapter 1, program errors have been called "bugs" since the early days of computer programming. The term is often said to have originated from an actual moth that was discovered trapped in the circuitry of a computer at Harvard University in 1945. Actually, the term "bug" was in use prior to 1945 to mean trouble with any electrical apparatus; even during Thomas Edison's life, it meant an "industrial defect." However, the process of finding and correcting program errors has come to be known as debugging.

Each of the following pseudocode segments contains one or more bugs that you must find and correct.

1. This pseudocode segment is intended to describe determining whether you have passed or failed a course based on the average score of two classroom tests.

```
input midtermGrade
input finalGrade
average = (midGrade + finalGrade) / 2
print avg
if average >= 60 then
   print "Pass"
endif
else
   print "Fail"
```

2. This pseudocode segment is intended to describe computing the number of miles per gallon you get with your automobile. The program segment should continue as long as the user enters a positive value for miles traveled.

```
input gallonsOfGasUsed
input milesTraveled
while milesTraveled > 0
       milesPerGallon = gallonsOfGasUsed / milesTraveled
        print milesPerGal
endwhile
```

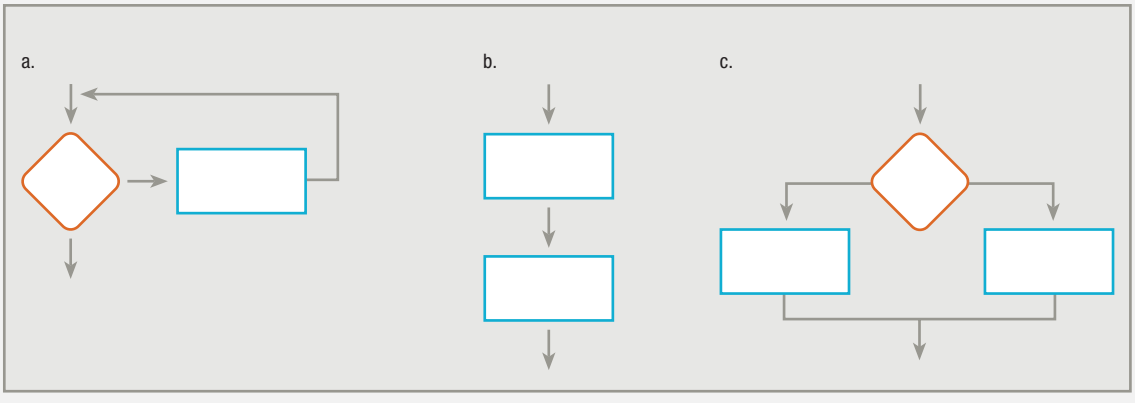
3. This pseudocode segment is intended to describe computing the cost per day for a vacation. The user enters a value for total dollars available to spend and can continue to enter new dollar amounts while the amount entered is not 0. For each new amount entered, if the amount of money available to spend per day is below $100, a message displays.

```
input totalDollarsAvailable
while totalDollarsAvailable not = 0
   dollarsPerDay = totalMoneyAvailable / 7
   print dollarsPerDay
endwhile
input totalDollarsAvailable
if dollarsPerDay > 100 then
   print "You better search for a bargain vacation"
endwhile
```

EXERCISES

1. Match the term with the structure diagram. (Because the structures go by more than one name, there are more terms than diagrams.)

1. sequence
2. selection
3. loop
4. do-while
5. decision
6. if-then-else
7. iteration



2. Match the term with the pseudocode segment. (Because the structures go by more than one name, there are more terms than pseudocode segments.)

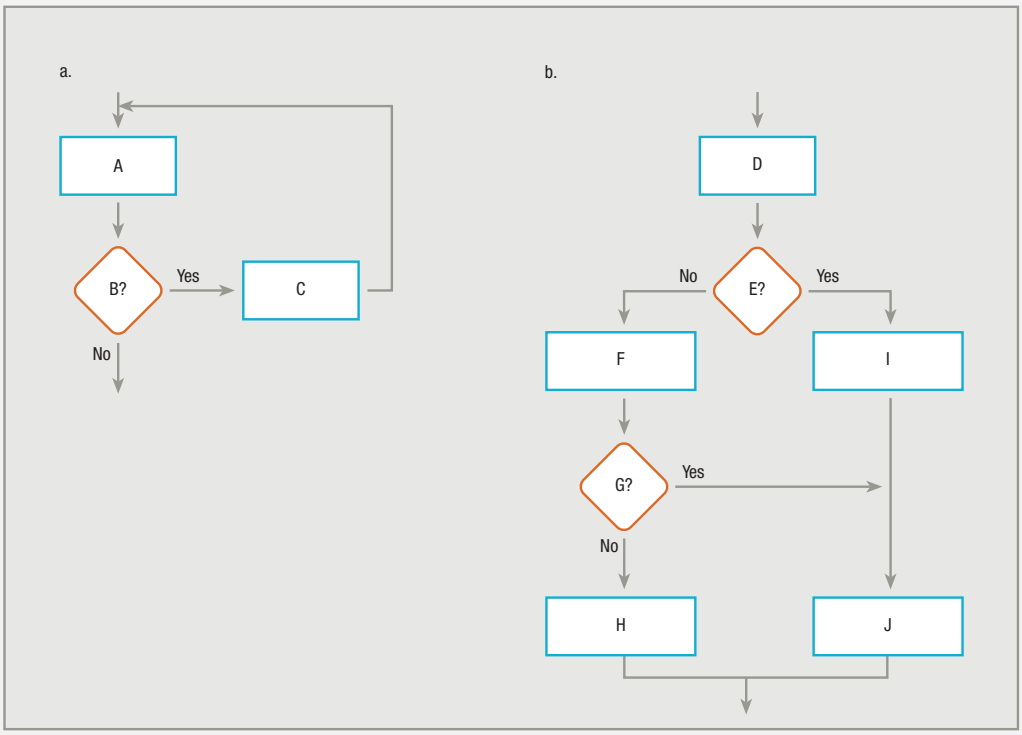
1. sequence
2. selection
3. loop
4. decision
5. if-then-else
6. iteration

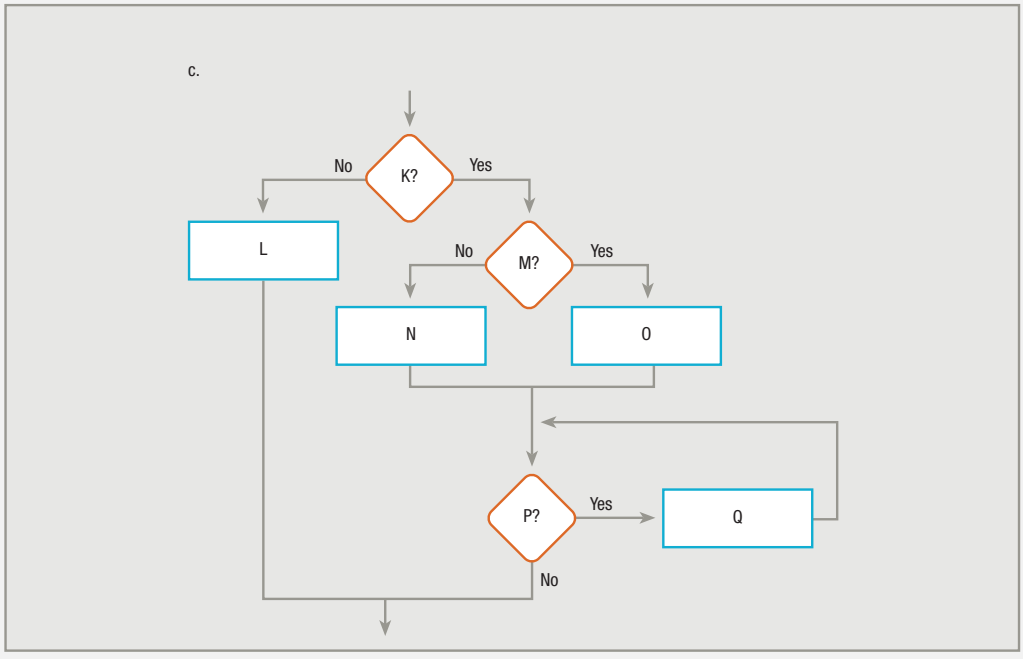
a.
```
while not eof
      print theAnswer
endwhile
```

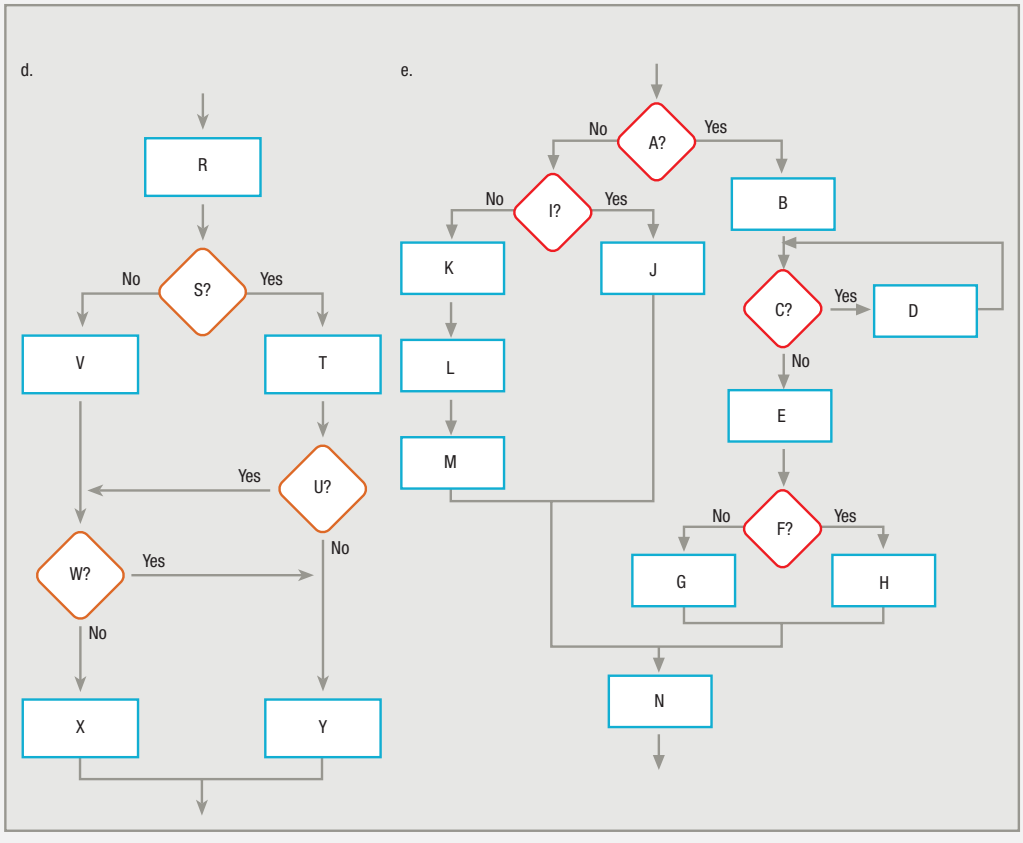
b.
```
if inventoryQuantity  >  0 then
      do fillOrderProcess
else
      do backOrderNotification
endif
```

c.
```
do localTaxCalculation
do stateTaxCalculation
do federalTaxCalculation
```

3. Is each of the following segments structured, or unstructured? If unstructured, redraw it so that it does the same thing but is structured.




c.
K?
No
Yes
L
M?
No
Yes
N
O
P?
Yes
Q
No


d.
R
S?
No
Yes
V
T
U?
Yes
No
W?
Yes
No
X
Y
e.
A?
No
Yes
I?
No
Yes
K
J
B
L
C?
Yes
D
No
M
E
F?
No
Yes
G
H
N

4. **Write pseudocode for each example (a through e) in Exercise 3.**

5. **Assume you have created a mechanical arm that can hold a pen. The arm can perform the following tasks:**

   - ☐ Lower the pen to a piece of paper.
   - ☐ Raise the pen from the paper.
   - ☐ Move the pen one inch along a straight line. (If the pen is lowered, this action draws a one-inch line from left to right; if the pen is raised, this action just repositions the pen one inch to the right.)
   - ☐ Turn 90 degrees to the right.
   - ☐ Draw a circle that is one inch in diameter.

   Draw a structured flowchart or write pseudocode describing the logic that would cause the arm to draw the following:

   a. a one-inch square
   b. a two-inch by one-inch rectangle
   c. a string of three beads

   Have a fellow student act as the mechanical arm and carry out your instructions.

6. **Assume you have created a mechanical robot that can perform the following tasks:**

   - ☐ Stand up.
   - ☐ Sit down.
   - ☐ Turn left 90 degrees.
   - ☐ Turn right 90 degrees.
   - ☐ Take a step.

   Additionally, the robot can determine the answer to one test condition:

   - ☐ Am I touching something?

   Place two chairs 20 feet apart, directly facing each other. Draw a structured flowchart or write pseudocode describing the logic that would allow the robot to start from a sitting position in one chair, cross the room, and end up sitting in the other chair.

   Have a fellow student act as the robot and carry out your instructions.

7. **Draw a structured flowchart or write structured pseudocode describing your preparation to go to work or school in the morning. Include at least two decisions and two loops.**

8. **Draw a structured flowchart or write structured pseudocode describing your preparation to go to bed at night. Include at least two decisions and two loops.**

9. **Choose a very simple children's game and describe its logic, using a structured flowchart or pseudocode. For example, you might try to explain Musical Chairs; Duck, Duck, Goose; the card game War; or the elimination game Eenie, Meenie, Minie, Moe.**

10. **Draw a structured flowchart or write structured pseudocode describing how your paycheck is calculated. Include at least two decisions.**

11. **Draw a structured flowchart or write structured pseudocode describing the steps a retail store employee should follow to process a customer purchase. Include at least two decisions.**

DETECTIVE WORK

1. **In this chapter, you learned what spaghetti code is. What is "ravioli code"?**

2. **Who was Edsger Dijkstra? What programming statement did he want to eliminate?**

3. **Who were Bohm and Jacopini? What contribution did they make to programming?**

UP FOR DISCUSSION

1. **Just because every logical program can be solved using only three structures (sequence, selection, and loop) does not mean there cannot be other useful structures. For example, the case, do-while, and do-until structures are never required, but they exist in many programming languages and can be quite useful. Try to design a new structure of your own and explain situations in which it would be useful.**