MAKING DECISIONS

After studying Chapter 5, you should be able to:

- Evaluate Boolean expressions to make comparisons
- Use the relational comparison operators
- Understand AND logic
- Understand OR logic
- Use selections within ranges
- Understand precedence when combining AND and OR selections
- Understand the case structure
- Use decision tables

EVALUATING BOOLEAN EXPRESSIONS TO MAKE COMPARISONS

One reason people think computers are smart lies in a computer program's ability to make decisions. For example, a medical diagnosis program that can decide if your symptoms fit various disease profiles seems quite intelligent, as does a program that can offer you potential vacation routes based on your destination.

The selection structure (also called the decision structure) involved in such programs is not new to you—it's one of the basic structures of structured programming. See Figures 5-1 and 5-2.

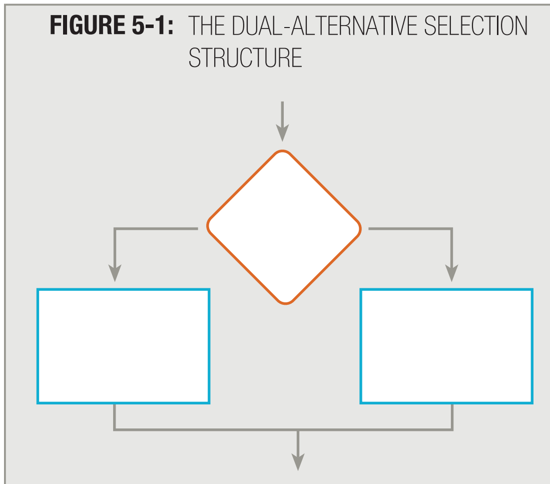


FIGURE 5-1: THE DUAL-ALTERNATIVE SELECTION STRUCTURE

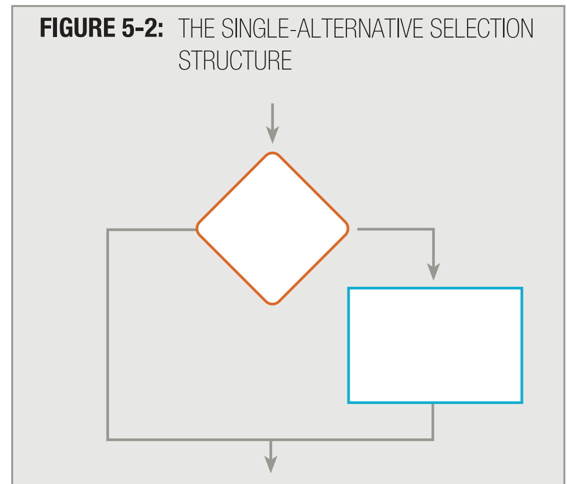


FIGURE 5-2: THE SINGLE-ALTERNATIVE SELECTION STRUCTURE

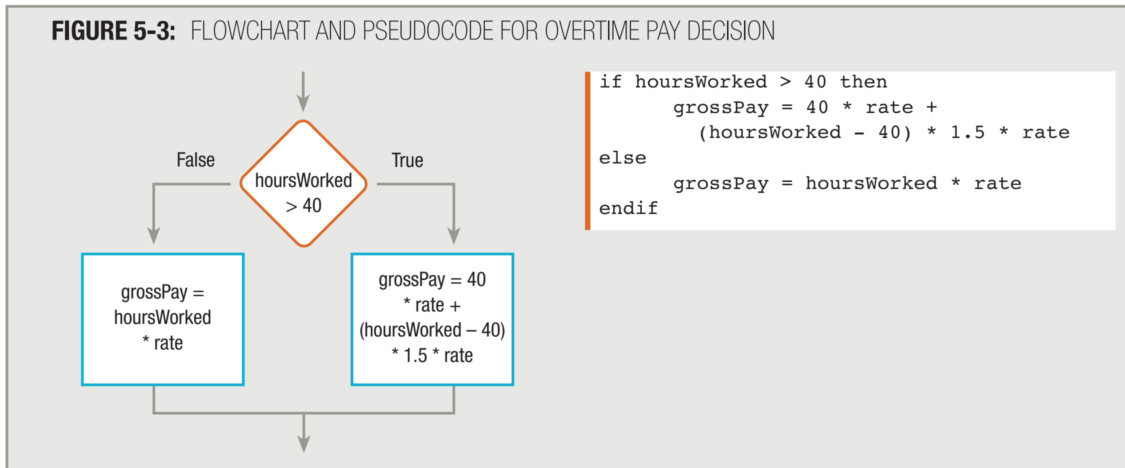
You can refer to the structure in Figure 5-1 as a **dual-alternative**, or **binary**, selection because there is an action associated with each of two possible outcomes. Depending on the answer to the question represented by the diamond, the logical flow proceeds either to the left branch of the structure or to the right. The choices are mutually exclusive; that is, the logic can flow only to one of the two alternatives, never to both. This selection structure is also called an **if-then-else** structure because it fits the statement:

```
if the answer to the question is yes, then
    do something
else
    do somethingElse
endif
```

The flowchart segment in Figure 5-2 represents a **single-alternative**, or **unary**, selection where action is required for only one outcome of the question. You call this form of the if-then-else structure an **if-then**, because no alternative or "else" action is included or necessary.

TIP You can call a single-alternative decision (or selection) a *single-sided decision*. Similarly, a dual-alternative decision (or selection) is a *double-sided decision*.

For example, Figure 5-3 shows the flowchart and pseudocode for a typical if-then-else decision in a business program. Many organizations pay employees time and a half (one and one-half times their usual hourly rate) for hours in excess of 40 per week. The logic segments in the figure show this decision.

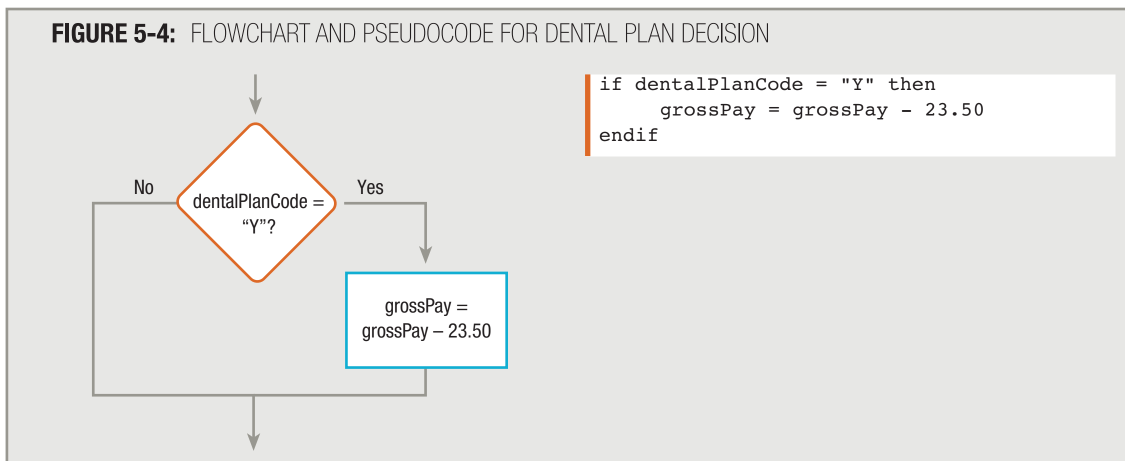



```
if hoursWorked > 40 then
      grossPay = 40 * rate +
        (hoursWorked - 40) * 1.5 * rate
else
      grossPay = hoursWorked * rate
endif
```

FIGURE 5-3: FLOWCHART AND PSEUDOCODE FOR OVERTIME PAY DECISION

In the example in Figure 5-3, the longer calculation that adds a time-and-a-half factor to an employee's gross pay executes only when the expression `hoursWorked > 40` is true. The overtime calculation exists in the **if clause** of the decision—the part of the decision that holds the action or actions that execute when the tested condition in the decision is true. The shorter, regular pay calculation, which produces `grossPay` by multiplying `hoursWorked` by `rate`, constitutes the **else clause** of the decision—the part that executes only when the tested condition in the decision is false.

The typical if-then decision in Figure 5-4 shows an employee's paycheck being reduced if the employee participates in the dental plan. No action is taken if the employee is not a dental plan participant.




```
if dentalPlanCode = "Y" then
     grossPay = grossPay - 23.50
endif
```

FIGURE 5-4: FLOWCHART AND PSEUDOCODE FOR DENTAL PLAN DECISION

The expressions `hoursWorked > 40` and `dentalPlanCode = "Y"` that appear in Figures 5-3 and 5-4, respectively, are Boolean expressions. In Chapter 4, you learned that in programming, an expression is a statement, or part of a statement, that has a value. For example, an arithmetic expression is one that performs arithmetic, resulting in a value. A **Boolean expression** is one that represents only one of two states, usually expressed as true or false. Every decision you make in a computer program involves evaluating a Boolean expression. True/false evaluation is "natural" from a computer's standpoint, because computer circuitry consists of two-state, on-off switches, often represented by 1 or 0. Every computer decision yields a true-or-false, yes-or-no, 1-or-0 result.

TIP George Boole was a mathematician who lived from 1815 to 1864. He approached logic more simply than his predecessors did, by expressing logical selections with common algebraic symbols. He is considered a pioneer in mathematical logic, and Boolean (true/false) expressions are named for him.

USING THE RELATIONAL COMPARISON OPERATORS

Usually, you can compare only values that are of the same type; that is, you can compare numeric values to other numbers and character values to other characters. You can ask every programming question by using one of only six types of comparison operators in a Boolean expression. For any two values that are the same type, you can decide whether:

- The two values are equal.
- The first value is greater than the second value.
- The first value is less than the second value.
- The first value is greater than or equal to the second value.
- The first value is less than or equal to the second value.
- The two values are not equal.

TIP Usually, character variables are not considered to be equal unless they are identical, including the spacing and whether they appear in uppercase or lowercase. For example, "black pen" is *not* equal to "blackpen", "BLACK PEN", or "Black Pen".

TIP Some programming languages allow you to compare a character to a number. If this is the case, then a single character's numeric code value is used in the comparison. For example, most microcomputers use either the ASCII or Unicode coding system. In both of these systems, an uppercase "A" is represented numerically as a 65, an uppercase "B" is a 66, and so on. See Appendix B for more information on ASCII code and how numbers are used to store data.

In any Boolean expression, the two values used can be either variables or constants. For example, the expression `currentTotal = 100?` compares the value stored in a variable, `currentTotal`, to a numeric constant, `100`. Depending on the `currentTotal` value, the expression is true or false. In the expression `currentTotal = previousTotal?` both values are variables, and the result is also true or false depending on the values stored in each of the two variables. Although it's legal to do so, you would never use expressions in which you compare two

unnamed constants—for example, `20 = 20?` or `30 = 40?`. Such expressions are considered **trivial** because each will always evaluate to the same result: true for the first expression and false for the second.

Each programming language supports its own set of **relational comparison operators**, or comparison symbols, that express these Boolean tests. For example, many languages such as Visual Basic and Pascal use the equal sign (=) to express testing for equivalency, so `balanceDue = 0` compares `balanceDue` to zero. COBOL programmers can use the equal sign, but they also can spell out the expression, as in `balanceDue equal to 0`. RPG programmers use the two-letter operator `EQ` in place of a symbol. C#, C++, and Java programmers use two equal signs to test for equivalency, so they write `balanceDue == 0` to compare the two values. Although each programming language supports its own syntax for comparing values' equivalency, all languages provide for the same logical concept of equivalency.

TIP

Visual Basic uses the single equal sign both for assignment and when testing for equivalency; the interpretation of the operator depends on the context. The reason some languages use two equal signs for comparisons is to avoid confusion with assignment statements such as `balanceDue = 0`. In C++, C#, or Java, this statement only assigns the value 0 to `balanceDue`; it does not compare `balanceDue` to zero.

TIP

Whenever you use a comparison operator, you must provide a value on each side of the operator. Comparison operators are sometimes called *binary operators* because of this requirement. Some programmers use the terms "comparison operator," "relational operator," and "**logical operator**" interchangeably. However, many prefer to reserve the term "logical operator" for manipulations on single bits.

Most languages allow you to use the algebraic signs for greater than (>) and less than (<) to make the corresponding comparisons. Additionally, COBOL, which is very similar to English, allows you to spell out the comparisons in expressions such as `daysPastDue is greater than 30` or `packageWeight is less than maximumWeightAllowed`. RPG uses the two-letter abbreviations `GT` and `LT` to represent greater than or less than. When you create a flowchart or pseudocode, you can use any form of notation you want to express "greater than" and "less than." It's simplest to use the symbols > and < if you are comfortable with their meaning. As with equivalency, the syntax changes when you change languages, but the concepts of greater than and less than exist in all programming languages.

Most programming languages allow you to express "greater than or equal to" by typing a greater-than sign immediately followed by an equal sign (>=). When you are drawing a flowchart or writing pseudocode, you might prefer a greater-than sign with a line under it (≥) because mathematicians use that symbol to mean "greater than or equal to." However, when you write a program, you type >= as two separate characters, because no single key on the keyboard expresses this concept. Similarly, "less than or equal to" is written with two symbols, < immediately followed by =.

TIP

The operators >= and <= are always treated as a single unit; no spaces separate the two parts of the operator. Also, the equal sign always appears second. No programming language allows => or =< as a comparison operator.

Any logical situation can be expressed using just three types of comparisons: equal, greater than, and less than. You never need the three additional comparisons (greater than or equal to, less than or equal to, or not equal to), but using them often makes decisions more convenient. For example, assume you need to issue a 10 percent discount to any customer whose age is 65 or greater, and charge full price to other customers. You can use the greater-than-or-equal-to symbol to write the logic as follows:

```
if customerAge >= 65 then
    discount = 0.10
else
    discount = 0
endif
```

As an alternative, if you want to use only one of the three basic comparisons (=, >, and <), you can express the same logic by writing:

```
if customerAge < 65 then
    discount = 0
else
    discount = 0.10
endif
```

In any decision for which `a >= b` is true, then `a < b` is false. Conversely, if `a >= b` is false, then `a < b` is true. By rephrasing the question and swapping the actions taken based on the outcome, you can make the same decision in multiple ways. The clearest route is often to ask a question so the positive or true outcome results in the unusual action. For example, assume that charging a customer full price is the ordinary course, and that providing a discount is the unusual occurrence. When your company policy is to “provide a discount for those who are 65 and older,” the phrase “greater than or equal to” comes to mind, so it is the most natural to use. Conversely, if your policy is to “provide no discount for those under 65,” then it is more natural to use the “less than” syntax. Either way, the same people receive a discount.

Comparing two amounts to decide if they are *not* equal to each other is the most confusing of all the comparisons. Using “not equal to” in decisions involves thinking in double negatives, which makes you prone to include logical errors in your programs. For example, consider the flowchart segment in Figure 5-5.

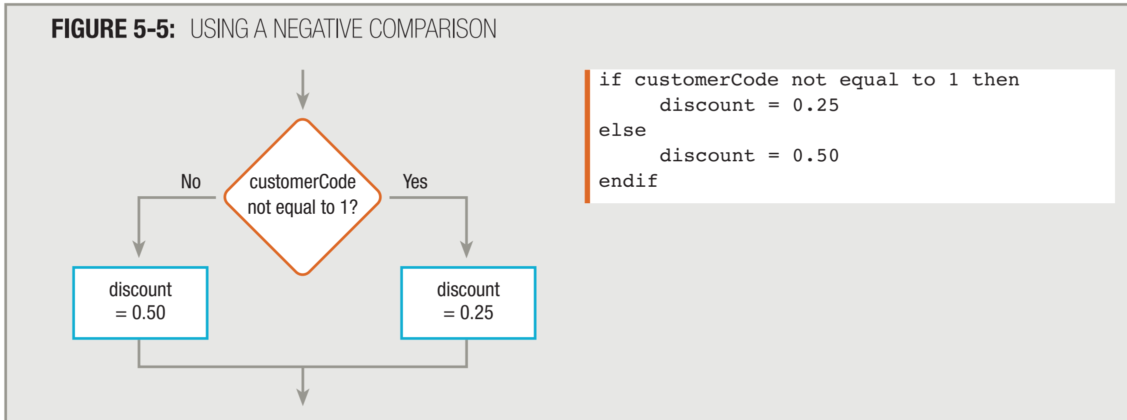


FIGURE 5-5: USING A NEGATIVE COMPARISON

In Figure 5-5, if the value of `customerCode` *is* equal to 1, the logical flow follows the false branch of the selection. If `customerCode not equal to 1` is true, the `discount` is 0.25; if `customerCode not equal to 1` is not true, it means the `customerCode` *is* 1, and the `discount` is 0.50. Even using the phrase "`customerCode not equal to 1 is not true`" is awkward.

Figure 5-6 shows the same decision, this time asked in the positive. Making the decision `if customerCode` *is* `1 then discount = 0.50` is clearer than trying to determine what `customerCode` is *not*.

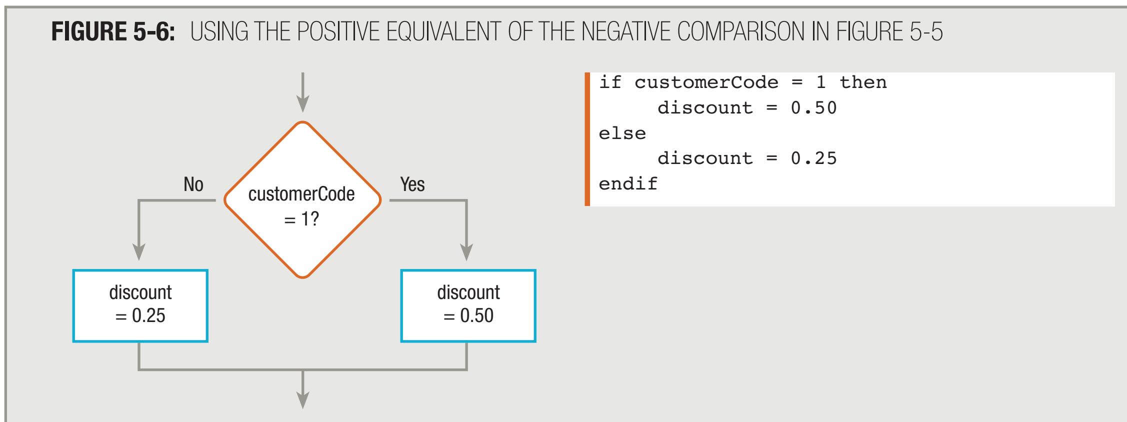


FIGURE 5-6: USING THE POSITIVE EQUIVALENT OF THE NEGATIVE COMPARISON IN FIGURE 5-5

Besides being awkward to use, the "not equal to" comparison operator is the one most likely to be different in the various programming languages you may use. COBOL allows you to write "not equal to"; Visual Basic and Pascal use a less-than sign followed immediately by a greater-than sign (<>); C#, C++, C, and Java use an exclamation point followed by an equal sign (!=). In a flowchart or in pseudocode, you can use the symbol that mathematicians use to mean "not equal," an equal sign with a slash through it (≠). When you program, you will not be able to use this symbol, because no single key on the keyboard produces it.

TIP

Although NOT comparisons can be awkward to use, there are times when your meaning is clearest if you use one. Frequently, this occurs when you take action only when some comparison is expressed negatively—for example, when one value is not equal to another value. Examples of situations in which a negative comparison makes sense include the following:

```
if customerZipCode is not equal to localZipCode then
    add DELIVERY_CHARGE to total
endif

if creditCardBalance is not 0 then
     financeCharge = balance * INTEREST_RATE
endif
```

In these cases, action is taken when two values are not equal. The mainline logic of many programs, including those you have worked with in this book, includes a negative comparison that controls a loop. The pseudocode you have seen for almost every program includes a statement similar to: `while not eof, perform mainLoop()`.

Figure 5-7 summarizes the six comparison operators and contrasts trivial (both true and false) examples with typical examples of their use.

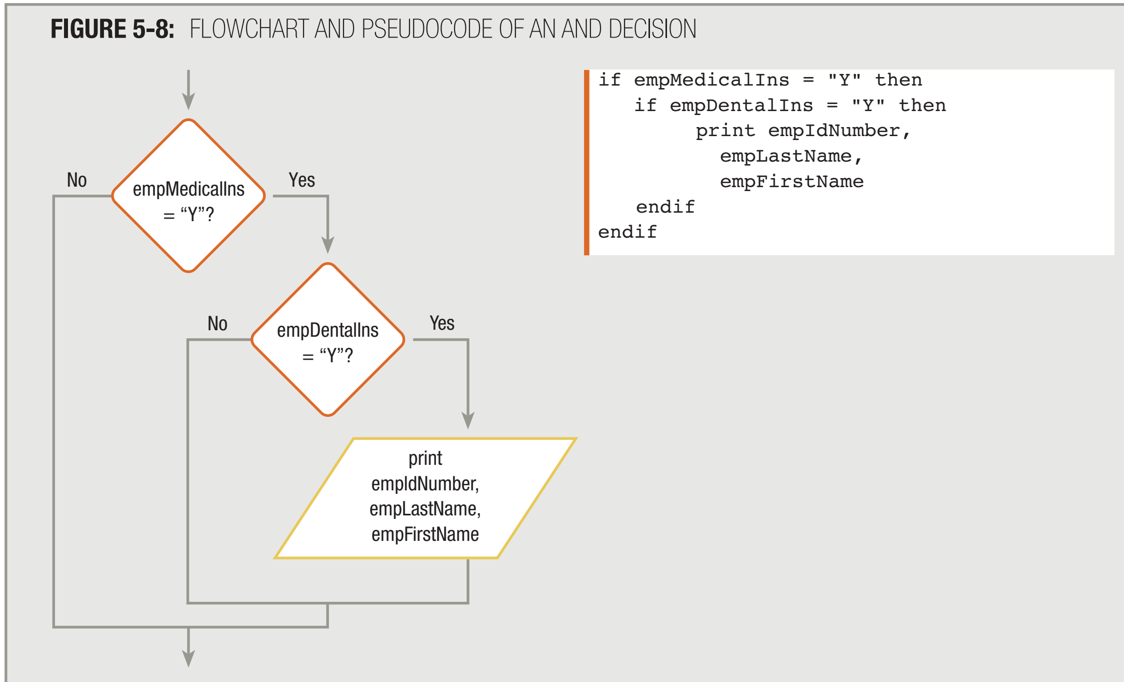
FIGURE 5-7: RELATIONAL COMPARISONS

Comparison	Trivial true example	Trivial false example	Typical example
Equal to	7 = 7?	7 = 4?	amtOrdered = 12?
Greater than	12 > 3?	4 > 9?	hoursWorked > 40?
Less than	1 < 8?	13 < 10?	hourlyWage < 5.65?
Greater than or equal to	5 >= 5?	3 >= 9?	customerAge >= 65?
Less than or equal to	4 <= 4?	8 <= 2?	daysOverdue <= 60?
Not equal to	16 <> 3?	18 <> 18?	customerBalance <> 0?

UNDERSTANDING AND LOGIC

Often, you need more than one selection structure to determine whether an action should take place. For example, suppose that your employer wants a report that lists workers who have registered for both insurance plans offered by the company: the medical plan and the dental plan. This type of situation is known as an **AND decision** because the employee's record must pass two tests—participation in the medical plan *and* participation in the dental plan—before you write that employee's information on the report. A compound, or AND, decision requires a **nested decision**, or a **nested if**. A nested decision is a decision "inside of" another decision. The logic looks like Figure 5-8.

FIGURE 5-8: FLOWCHART AND PSEUDOCODE OF AN AND DECISION




```
if empMedicalIns = "Y" then
   if empDentalIns = "Y" then
        print empIdNumber,
          empLastName,
          empFirstName
   endif
endif
```

TIP You first learned about nesting structures in Chapter 2.

TIP A series of nested `if` statements can also be called a **cascading if statement**.

The AND decision shown in Figure 5-8 is part of a much larger program. To help you develop this program, suppose your employer provides you with the employee data file description shown in Figure 5-9, and you learn that the medical and dental insurance fields contain a single character, "Y" or "N", indicating each employee's participation status. With your employer's approval, you develop the sample output shown in Figure 5-10.

FIGURE 5-9: EMPLOYEE FILE DESCRIPTION

```
EMPLOYEE FILE DESCRIPTION
File Name: EMPFILE
FIELD DESCRIPTION       DATA TYPE     COMMENTS
ID Number               Numeric       4 digits, 0 decimal places
Last Name               Character     15 characters
First Name              Character     15 characters
Department              Numeric       1 digit
Hourly Rate             Numeric       2 decimal places
Medical Plan            Character     1 character, Y or N
Dental Plan             Character     1 character, Y or N
Number of Dependents    Numeric       0 decimal places
```

FIGURE 5-10: SAMPLE REPORT LISTING EMPLOYEES PARTICIPATING IN BOTH INSURANCE PLANS

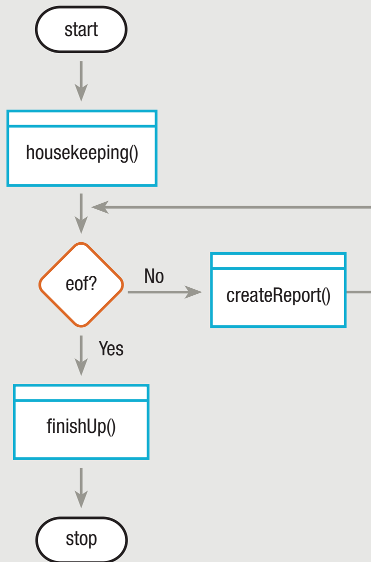
```
Employees with Medical and Dental Insurance

ID Number          Last Name           First Name

1246               Kroening            Virginia
1419               Lewis               Kathleen
2765               Bowman              Bradley
3872               Daniels             James
```

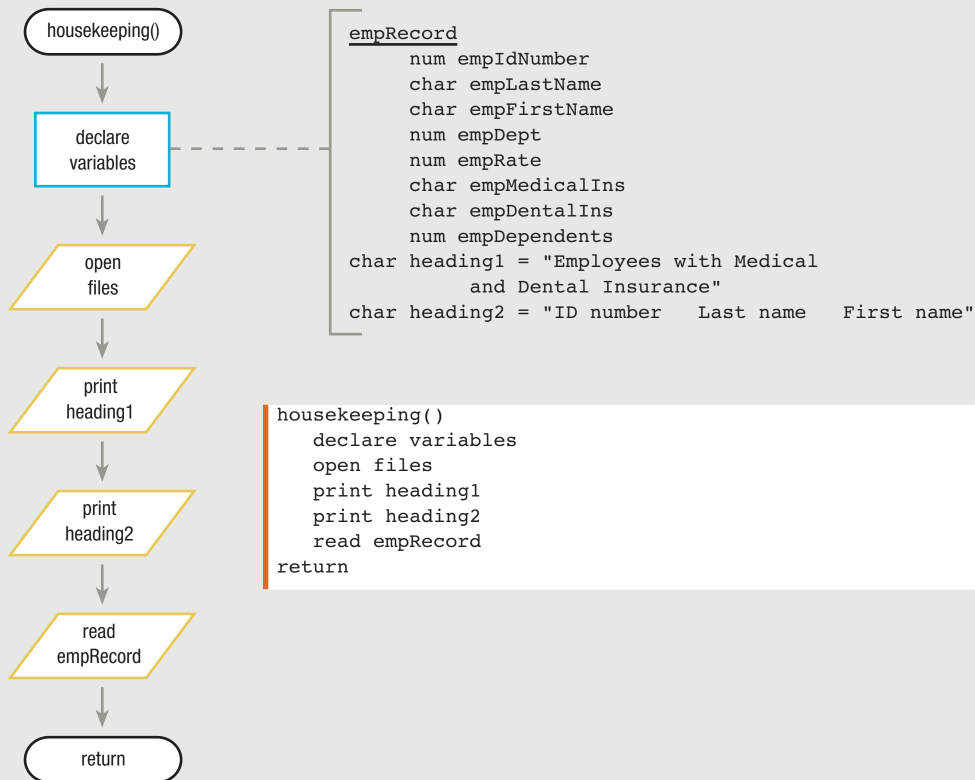
The mainline logic and `housekeeping()` routines for this program are diagrammed in Figures 5-11 and 5-12.

FIGURE 5-11: FLOWCHART AND PSEUDOCODE OF MAINLINE LOGIC FOR MEDICAL AND DENTAL PARTICIPANT REPORT



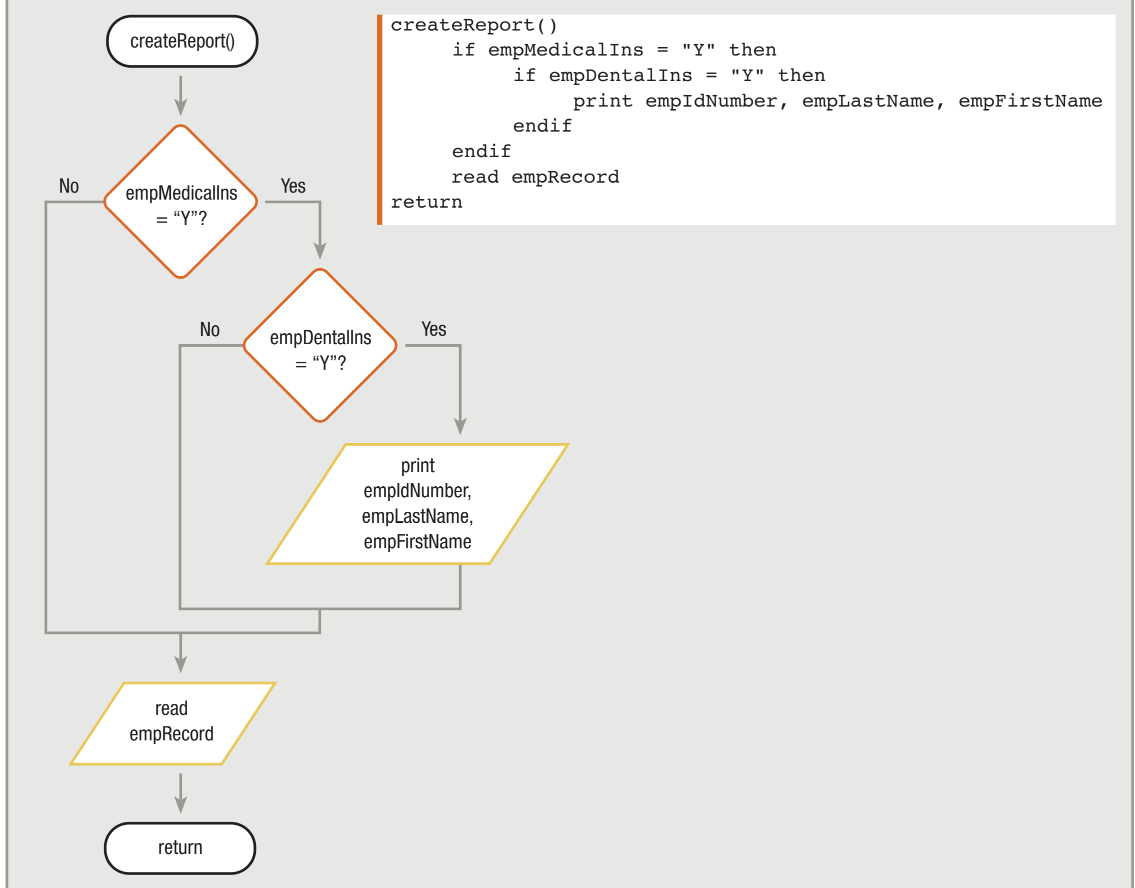

```
start
   perform housekeeping()
   while not eof
      perform createReport()
   endwhile
   perform finishUp()
stop
```

FIGURE 5-12: FLOWCHART AND PSEUDOCODE OF `housekeeping()` MODULE FOR MEDICAL AND DENTAL PARTICIPANT REPORT



At the end of the `housekeeping()` module, the first employee record is read into computer memory. Assuming that the `eof` condition is not yet met, the logical flow proceeds to the `createReport()` method. If the program required data for all employees to be printed, this method would simply print the information from the current record and get the next record. However, in this case, the output should contain only the names of those employees who participate in both the medical and dental insurance plans. Therefore, within the `createReport()` module of this program, you ask the questions that determine whether the current employee's record will print; if the employee's data meet the medical and dental insurance requirements, then you print the record. Whether or not you take the path that prints the record, the last thing you do in the `createReport()` method is to read the next input record. Figure 5-13 shows the `createReport()` module.

FIGURE 5-13: THE `createReport()` MODULE OF A PROGRAM THAT LISTS EMPLOYEES WHO ARE BOTH MEDICAL AND DENTAL INSURANCE PROGRAM PARTICIPANTS




```
createReport()
     if empMedicalIns = "Y" then
          if empDentalIns = "Y" then
               print empIdNumber, empLastName, empFirstName
          endif
     endif
     read empRecord
return
```

TIP

At the end of the `housekeeping()` module in Figure 5-12, instead of the statement `read empRecord`, an interactive program might prompt the user for values for each of the eight data fields. Instead of the single read statement, you might choose to call a method containing eight pairs of statements, such as `print "Please enter employee ID number"` and `read empIdNumber`. The programs in this chapter read data from a file to keep them simpler.

The `createReport()` module works like this: If the employee has medical insurance, *then* and *only then* test to see if the employee has dental insurance. If so, *then* and *only then* print the employee's data. The dental insurance question is nested entirely within half of the medical insurance question structure. If an employee does not carry medical insurance, there is no need to ask about the dental insurance; the employee is already disqualified from the report. Pseudocode for the entire program is shown in Figure 5-14. Notice how the second (dental insurance) decision within the `createReport()` method is indented within the first (medical insurance) decision. This technique shows that the second question is asked only when the result of the first comparison is true.

FIGURE 5-14: PSEUDOCODE OF PROGRAM THAT PRINTS RECORDS OF EMPLOYEES WHO PARTICIPATE IN BOTH THE MEDICAL AND DENTAL INSURANCE PLANS

```
start
    perform housekeeping()
    while not eof
      perform createReport()
    endwhile
    perform finishUp()
stop

housekeeping()
    declare variables
    open files
    print heading1
    print heading2
    read empRecord
return

createReport()
    if empMedicalIns = "Y" then
         if empDentalIns = "Y" then
            print empIdNumber, empLastName, empFirstName
         endif
    endif
    read empRecord
return

finishUp()
    close files
return
```

```
empRecord
     num empIdNumber
     char empLastName
     char empFirstName
     num empDept
     num empRate
     char empMedicalIns
     char empDentalIns
     num empDependents
char heading1 = "Employees with Medical
          and Dental Insurance"
char heading2 = "ID number   Last name   First name"
```

WRITING NESTED AND DECISIONS FOR EFFICIENCY

When you nest decisions because the resulting action requires that two conditions be true, you must decide which of the two decisions to make first. Logically, either selection in an AND decision can come first. However, when there are two selections, you often can improve your program's performance by making an appropriate choice as to which selection to make first.

For example, Figure 5-15 shows the nested decision structure in the `createReport()` method logic of the program that produces a report of employees who participate in both the medical and dental insurance plans. Alternatively, you can write the decision as in Figure 5-16.

FIGURE 5-15: FINDING MEDICAL AND DENTAL PLAN PARTICIPANTS, CHECKING MEDICAL FIRST

```
if empMedicalIns = "Y" then
     if empDentalIns = "Y" then
          print empIdNumber, empLastName, empFirstName
     endif
endif
```

FIGURE 5-16: FINDING DENTAL AND MEDICAL PLAN PARTICIPANTS, CHECKING DENTAL FIRST

```
if empDentalIns = "Y" then
     if empMedicalIns = "Y" then
           print empIdNumber, empLastName, empFirstName
     endif
endif
```

Examine the decision statements in Figures 5-15 and 5-16. If you want to print employees who participate in the medical AND dental plans, you can ask about the medical plan first, eliminate those employees who do not participate, and ask about the dental plan only for those employees who "pass" the medical insurance test. Or, you could ask about the dental plan first, eliminate those who do not participate, and ask about the medical plan only for those employees who "pass" the dental insurance test. Either way, the final list contains only those employees who have both kinds of insurance.

Does it make a difference which question is asked first? As far as the output goes, no. Either way, the same employee names appear on the report—those with both types of insurance. As far as program efficiency goes, however, it *might* make a difference which question is asked first.

Assume you know that out of 1,000 employees in your company, about 90 percent, or 900, participate in the medical insurance plan. Assume you also know that out of 1,000 employees, only about half, or 500, participate in the dental plan.

The medical and dental insurance program will ask the first question in the `createReport()` method 1,000 times during its execution—once for each employee record contained in the input file. If the program uses the logic in Figure 5-15, it asks the first question **`empMedicalIns = "Y"?`** 1,000 times. For approximately 90 percent of the employees, or 900 of the records, the answer is true, meaning the **`empMedicalIns`** field contains the character "Y". So 100 employees are eliminated, and 900 proceed to the next question about dental insurance. Only about half of the employees participate in the dental plan, so 450 out of the 900 will appear on the printed report.

Using the alternate logic in Figure 5-16, the program asks the first question **`empDentalIns = "Y"?`** 1,000 times. Because only about half of the company's employees participate, only 500 will "pass" this test and proceed to the medical insurance question. Then about 90 percent of the 500, or 450 employees, will appear on the printed report. Whether you use the logic in Figure 5-15 or 5-16, the same 450 employees who have both types of insurance appear on the report.

The difference lies in the fact that when you use the logic in Figure 5-15, the program must ask 1,900 questions to produce the report—the medical insurance question tests all 1,000 employee records, and 900 continue to the dental insurance question. If you use the logic in Figure 5-16 to produce the report, the program asks only 1,500 questions—all 1,000 records are tested for dental insurance, but only 500 proceed to the medical insurance question. By asking about the dental insurance first, you "save" 400 decisions.

The 400-question difference between the first set of decisions and the second set really doesn't take much time on most computers. But it will take *some* time, and if there are hundreds of thousands of employees instead of only 1,000, or if many such decisions have to be made within a program, performance time can be significantly improved by asking questions in the proper order.

In many AND decisions, you have no idea which of two events is more likely to occur; in that case, you can legitimately ask either question first. In addition, even though you know the probability of each of two conditions, the two events might not be mutually exclusive; that is, one might depend on the other. For example, if employees with dental insurance are significantly more likely to carry medical insurance than those who don't carry dental insurance, the order in which to ask the questions might matter less or not matter at all. However, if you do know the probabilities of the conditions, or can make a reasonable guess, the general rule is: *In an AND decision, first ask the question that is less likely to be true.* This eliminates as many records as possible from having to go through the second decision, which speeds up processing time.

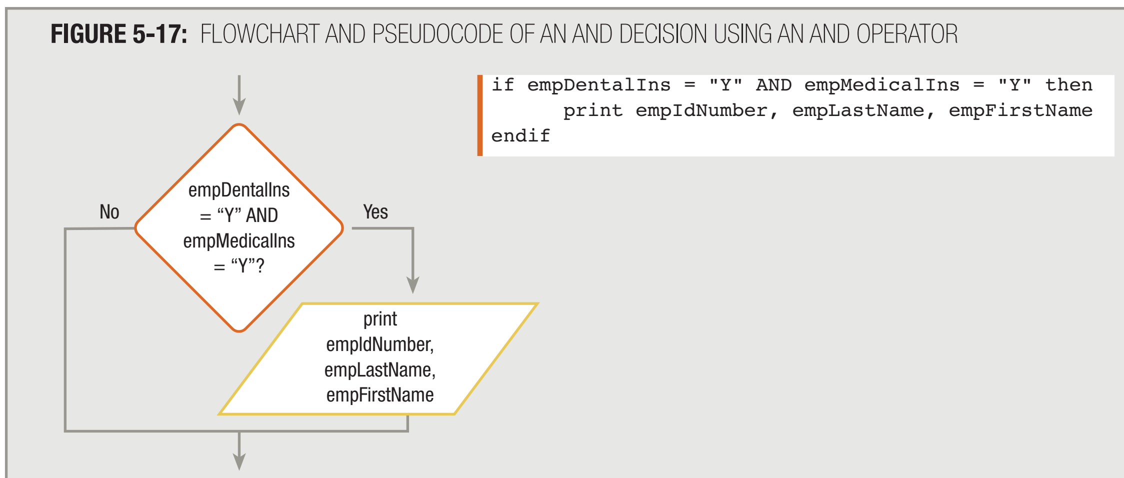
COMBINING DECISIONS IN AN AND SELECTION

Most programming languages allow you to ask two or more questions in a single comparison by using a **logical AND operator**. For example, if you want to select employees who carry both medical and dental insurance, you can use nested `ifs`, or you can include both decisions in a single statement by writing `empDentalIns = "Y" AND empMedicalIns = "Y"?`. When you use one or more AND operators to combine two or more Boolean expressions, each Boolean expression must be true in order for the entire expression to be evaluated as true. For example, if you ask, "Are you at least 18, and are you a registered voter, and did you vote in the last election?", the answer to all three parts of the question must be "yes" before the response can be a single, summarizing "yes". If any part of the question is false, then the entire question is false.

TIP You can think of an AND expression in an algebraic way if you consider 0 to be false and any nonzero value to be true. The AND operator works like multiplication (not addition, as you might suspect). A true expression AND a true expression yields a true result because 1 * 1 is 1. Any other combination yields a false result because 1 * 0, 0 * 1, and 0 * 0 all result in 0.

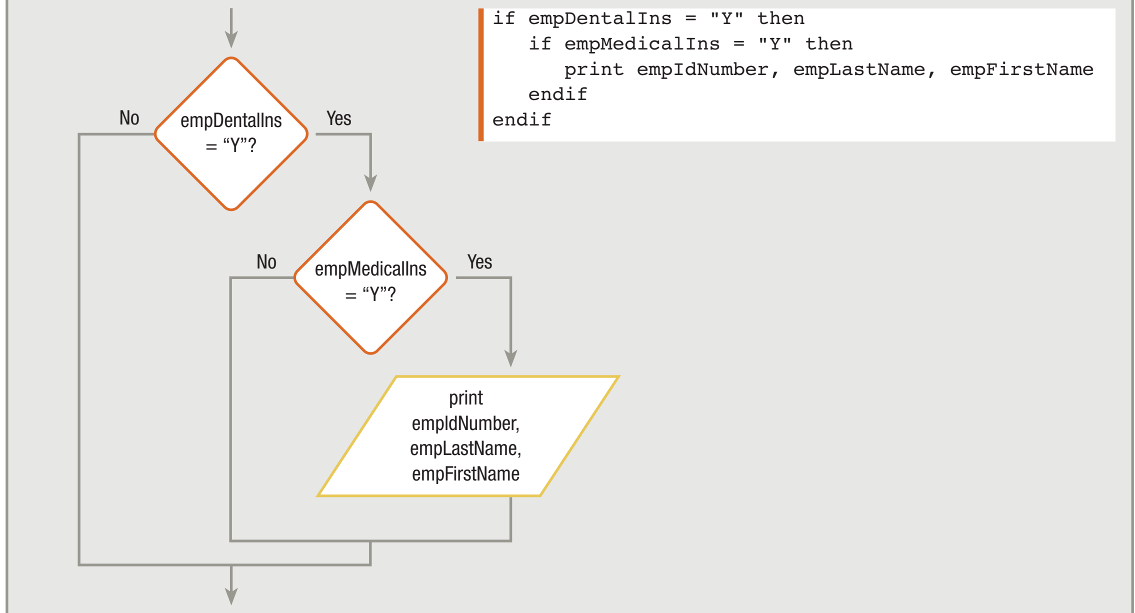
If the programming language you use allows an AND operator (and almost all do), you still must realize that the question you place first is the question that will be asked first, and cases that are eliminated based on the first question will not proceed to the second question. The computer can ask only one question at a time; even when your logic follows the flowchart segment in Figure 5-17, the computer will execute the logic in the flowchart in Figure 5-18.

FIGURE 5-17: FLOWCHART AND PSEUDOCODE OF AN AND DECISION USING AN AND OPERATOR




```
if empDentalIns = "Y" AND empMedicalIns = "Y" then
      print empIdNumber, empLastName, empFirstName
endif
```

FIGURE 5-18: FLOWCHART AND PSEUDOCODE OF COMPUTER LOGIC OF PROGRAM CONTAINING AN AND OPERATOR IN THE DECISION (THE COMPUTER STILL MAKES TWO SEPARATE DECISIONS, EVEN THOUGH AN AND OPERATOR IS USED)




```
if empDentalIns = "Y" then
   if empMedicalIns = "Y" then
      print empIdNumber, empLastName, empFirstName
   endif
endif
```

TIP

The AND operator in Java, C++, and C# consists of two ampersands, with no spaces between them (&&).

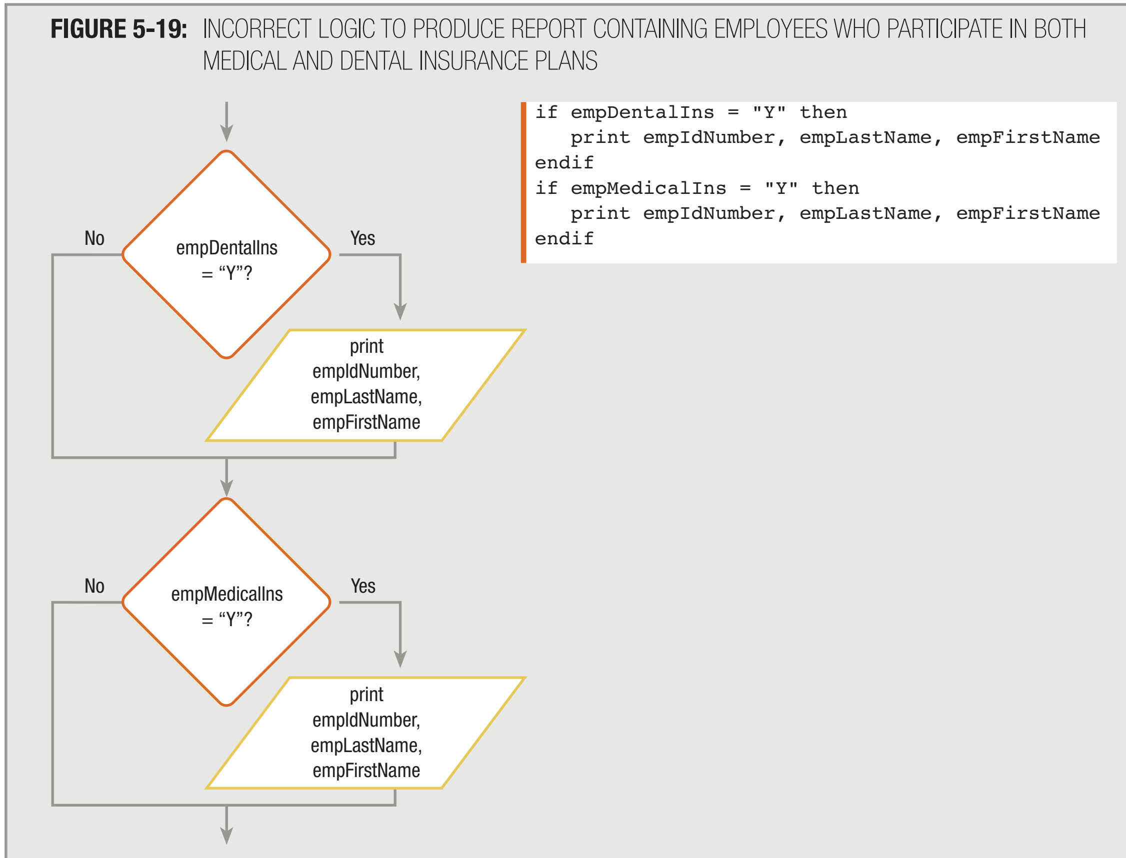
TIP

Using an AND operator in a decision that involves multiple conditions does not eliminate your responsibility for determining which of the conditions to test first. Even when you use an AND operator, the computer makes decisions one at a time, and makes them in the order you ask them. If the first question in an AND expression evaluates to false, then the entire expression is false, and the second question will not even be tested. Not bothering to test the second expression when it would make no difference in the ultimate result is called **short-circuiting**. (Some languages—for example, VB .NET—provide special non-short-circuiting operators. However, the standard AND operator is short-circuiting.)

AVOIDING COMMON ERRORS IN AN AND SELECTION

When you must satisfy two or more criteria to initiate an event in a program, you must make sure that the second decision is made entirely within the first decision. For example, if a program's objective is to print a report of those employees who carry both medical and dental insurance, then the program segment shown in Figure 5-19 contains three different types of logic errors.

FIGURE 5-19: INCORRECT LOGIC TO PRODUCE REPORT CONTAINING EMPLOYEES WHO PARTICIPATE IN BOTH MEDICAL AND DENTAL INSURANCE PLANS




```
if empDentalIns = "Y" then
   print empIdNumber, empLastName, empFirstName
endif
if empMedicalIns = "Y" then
   print empIdNumber, empLastName, empFirstName
endif
```

The diagram shows that the program asks the dental insurance question first. However, if an employee participates in the dental program, the employee's record prints immediately. The employee record should not print, because the employee might not have the medical insurance. In addition, the program should eliminate an employee without dental insurance from the next selection, but every employee's record proceeds to the medical insurance question, where it might print, whether the employee has dental insurance or not. Additionally, any employee who has both medical and dental insurance, having passed each test successfully, will appear twice on this report. For many reasons, the logic shown in Figure 5-19 is *not* correct for this problem.

Beginning programmers often make another type of error when they must make two comparisons on the same field when using a logical AND operator. For example, suppose you want to list employees who make between $10.00 and $11.99 per hour, inclusive. When you make this type of decision, you are basing it on a **range** of values—every value between low and high limits. For example, you want to select employees whose `empRate` is greater than or equal to

10.00 AND whose `empRate` is less than 12.00; therefore, you need to make two comparisons on the same field. Without the logical AND operator, the comparison is:

```
if empRate >= 10.00 then
     if empRate < 12.00 then
          print empIdNumber, empLastName, empFirstName
     endif
endif
```

To check for `empRate` values that are 10.00 or greater, you can use either `empRate > 9.99?` or `empRate >= 10.00?`. To check for `empRate` values under 12.00, you can write `empRate <= 11.99?` or `empRate < 12.00?`.

The correct way to make this comparison with the AND operator is as follows:

```
if empRate >= 10.00 AND empRate < 12.00 then
     print empIdNumber, empLastName, empFirstName
endif
```

You substitute the AND operator for the phrase `then if`. However, some programmers might try to make the comparison as follows:

```
if empRate >= 10.00 AND < 12.00 then
     print empIdNumber, empLastName, empFirstName
endif
```

In most languages, the phrase `empRate >= 10.00 AND < 12.00` is incorrect. The logical AND is usually a binary operator that requires a complete Boolean expression on each side. The expression to the right of the AND, `< 12.00`, is not a complete Boolean expression; you must indicate *what* is being compared to 12.00.

In some programming languages, such as COBOL and RPG, you can write the equivalent of `empRate >= 10.00 AND < 12.00?` and the `empRate` variable is implied for both comparisons. Still, it is clearer, and therefore preferable, to use the two full expressions, `empRate >= 10.00 AND empRate < 12.00?`.

UNDERSTANDING OR LOGIC

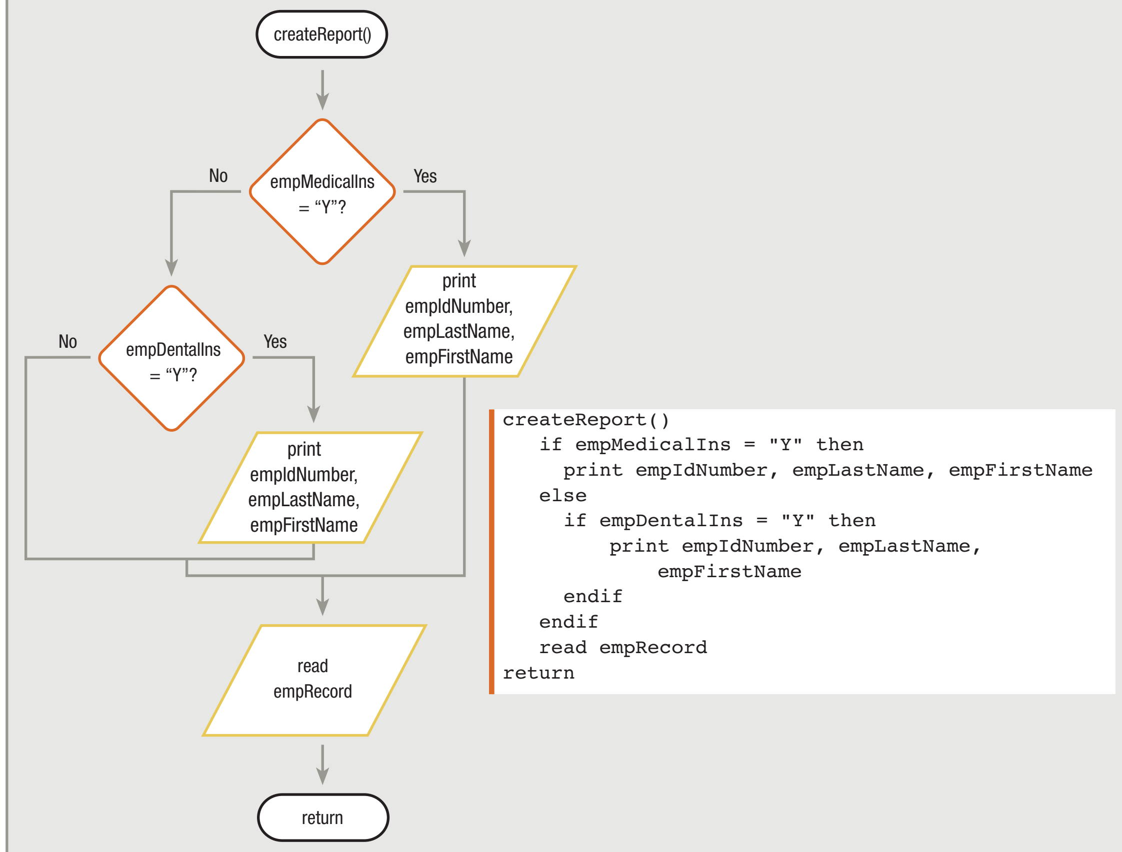
Sometimes, you want to take action when one *or* the other of two conditions is true. This is called an **OR decision** because either a first condition must be met *or* a second condition must be met for an event to take place. If someone asks you, “Are you free Friday or Saturday?”, only one of the two conditions has to be true in order for the answer to the whole question to be “yes”; only if the answers to both halves of the question are false is the value of the entire expression false.

You can think of an OR expression in an algebraic way if you consider 0 to be false and any nonzero value to be true. The OR operator works like addition. A false expression OR a false expression yields a false result because 0 + 0 is 0. Any other combination yields a true result because 1 + 0, 0 + 1, and 1 + 1 all result in nonzero values.

For example, suppose your employer wants a list of all employees who participate in either the medical or dental plan. Assuming you are using the same input file described in Figure 5-9, the mainline logic and `housekeeping()` module for this program are identical to those used in Figures 5-11 and 5-12. You only need to change the heading on the sample output (Figure 5-10) and change the `heading1` variable in Figure 5-12 from `heading1 = "Employees with Medical and Dental Insurance"` to `heading1 = "Employees with Medical or Dental Insurance"`. The only substantial changes to the program occur in the `createReport()` module.

Figure 5-20 shows the possible logic for the `createReport()` method in this OR selection. As each record enters the `createReport()` method, you ask the question `empMedicalIns = "Y"?`, and if the result is true, you print the employee data. Because the employee needs to participate in only one of the two insurance plans to be selected for printing, there is no need for further questioning after you have determined that an employee has medical insurance. If the employee does not participate in the medical insurance plan, only then do you need to ask if `empDentalIns = "Y"?`. If the employee does not have medical insurance, but does have dental, you want this employee information to print on the report.

FIGURE 5-20: FLOWCHART AND PSEUDOCODE FOR `createReport()` MODULE OF PROGRAM THAT PRINTS RECORDS OF EMPLOYEES WHO PARTICIPATE IN EITHER THE MEDICAL OR DENTAL INSURANCE PLAN

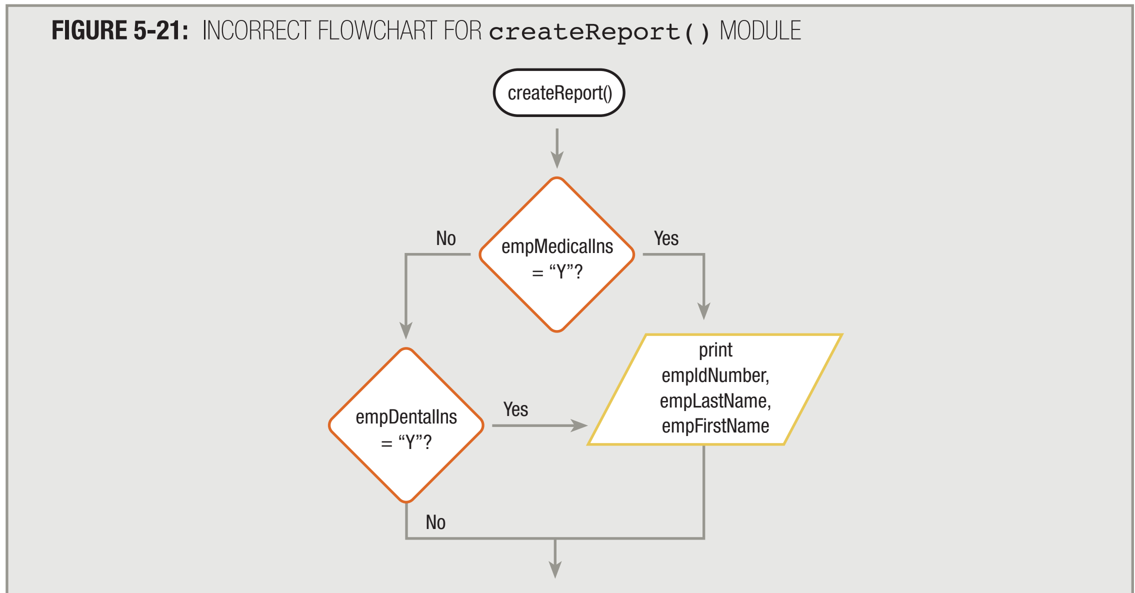



```
createReport()
   if empMedicalIns = "Y" then
     print empIdNumber, empLastName, empFirstName
   else
     if empDentalIns = "Y" then
         print empIdNumber, empLastName,
             empFirstName
     endif
   endif
   read empRecord
return
```

AVOIDING COMMON ERRORS IN AN OR SELECTION

You might have noticed that the statement `print empIdNumber, empLastName, empFirstName` appears twice in the flowchart and in the pseudocode shown in Figure 5-20. The temptation is to redraw the flowchart in Figure 5-20 to look like Figure 5-21. Logically, you can argue that the flowchart in Figure 5-21 is correct because the correct employee records print. However, this flowchart is not allowed because it is not structured.

FIGURE 5-21: INCORRECT FLOWCHART FOR `createReport()` MODULE



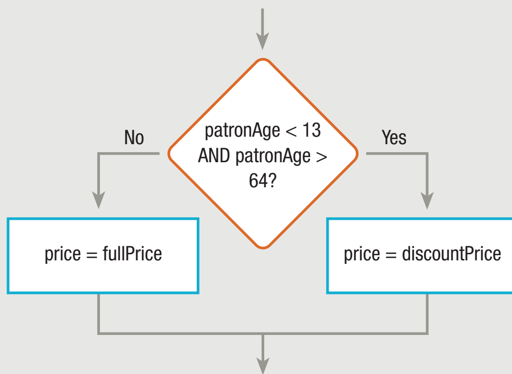
TIP If you do not see that Figure 5-21 is not structured, go back and review Chapter 2. In particular, review the example that begins at Figure 2-21.

TIP An additional source of error that is specific to the OR selection stems from a problem with language and the way people use it too casually. When your boss needs a report of all employees who carry medical or dental insurance, she is likely to say, "I need a report of all the people who have medical insurance and all those who have dental insurance." The request contains the word "and," and the report contains people who have one type of insurance "and" people who have another. However, the records you want to print are those from employees who have medical insurance OR dental insurance OR both. The logical situation requires an OR decision. Instead of saying "people who have medical insurance and people who have dental insurance," it would be clearer if your boss asked for "people who have medical or dental insurance." In other words, it would be more correct to put the question-joining "or" conjunction between the insurance types held by each person than between the people, but bosses and other human beings often do not speak like computers. As a programmer, you have the job of clarifying what really is being requested, and determining that often a request for A *and* B means a request for A *or* B.

The way we casually use English can cause another type of error when you require a decision based on a value falling within a range of values. For example, a movie theater manager might say, "Provide a discount to patrons who are

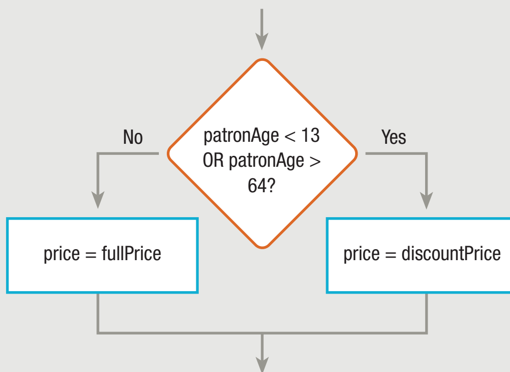
under 13 years old and those who are over 64 years old; otherwise, charge the full price." Because the manager has used the word "and" in the request, you might be tempted to create the decision shown in Figure 5-22; however, this logic will not provide a discounted price for any movie patron. You must remember that every time the decision in Figure 5-22 is made, it is made using a single data record. If the age field in that record contains an age lower than 13, then it cannot possibly contain an age over 64. Similarly, if it contains an age over 64, then there is no way it can contain an age under that. Therefore, there is no value that could be stored in the age field of a movie patron record for which both parts of the AND question are true—and the price will never be set to the `discountPrice` for any record. Figure 5-23 shows the correct logic.

FIGURE 5-22: INCORRECT LOGIC THAT ATTEMPTS TO PROVIDE A DISCOUNT FOR MOVIE PATRONS UNDER 13 AND FOR MOVIE PATRONS OVER 64




```
if patronAge < 13 AND patronAge > 64 then
   price = discountPrice
else
   price = fullPrice
endif
```

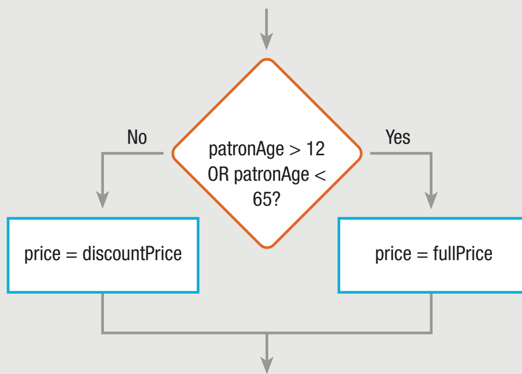
FIGURE 5-23: CORRECT LOGIC THAT PROVIDES A DISCOUNT FOR MOVIE PATRONS UNDER 13 AND FOR MOVIE PATRONS OVER 64




```
if patronAge < 13 OR patronAge > 64 then
   price = discountPrice
else
   price = fullPrice
endif
```

A similar error can occur in your logic if the theater manager says something like, "Don't give a discount—that is, charge full price—if a patron is over 12 or under 65." Because the word "or" appears in the request, you might plan your logic like that shown in Figure 5-24.

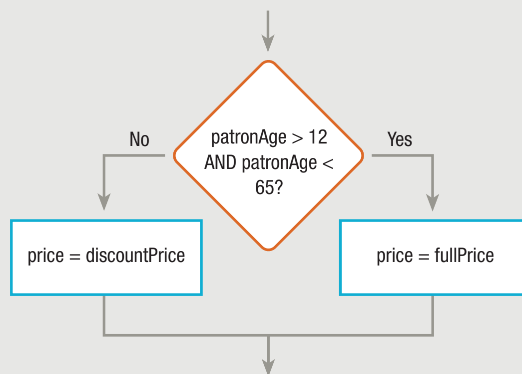
FIGURE 5-24: INCORRECT LOGIC THAT ATTEMPTS TO CHARGE FULL PRICE FOR MOVIE PATRONS OVER 12 AND UNDER 65




```
if patronAge > 12 OR patronAge < 65 then
   price = fullPrice
else
   price = discountPrice
endif
```

As in Figure 5-22, in Figure 5-24, no patron ever receives a discount, because every patron is either over 12 or under 65. Remember, in an OR decision, only one of the conditions needs to be true in order for the entire expression to be evaluated as true. So, for example, because a patron who is 10 is under 65, the full price is charged, and because a patron who is 70 is over 12, the full price also is charged. Figure 5-25 shows the correct logic for this decision.

FIGURE 5-25: CORRECT LOGIC THAT CHARGES FULL PRICE FOR MOVIE PATRONS OVER 12 AND UNDER 65




```
if patronAge > 12 AND patronAge < 65 then
   price = fullPrice
else
   price = discountPrice
endif
```

TIP

Using an OR operator in a decision that involves multiple conditions does not eliminate your responsibility for determining which of the conditions to test first. Even when you use an OR operator, the computer makes decisions one at a time, and makes them in the order you ask them. If the first question in an OR expression evaluates to true, then the entire expression is true, and the second question will not even be tested.

WRITING OR DECISIONS FOR EFFICIENCY

You can write a program that creates a report containing all employees who have either medical or dental insurance by using the `createReport()` method in either Figure 5-26 or Figure 5-27.

FIGURE 5-26: THE `createReport()` MODULE TO SELECT EMPLOYEES WITH MEDICAL OR DENTAL INSURANCE, USING MEDICAL DECISION FIRST

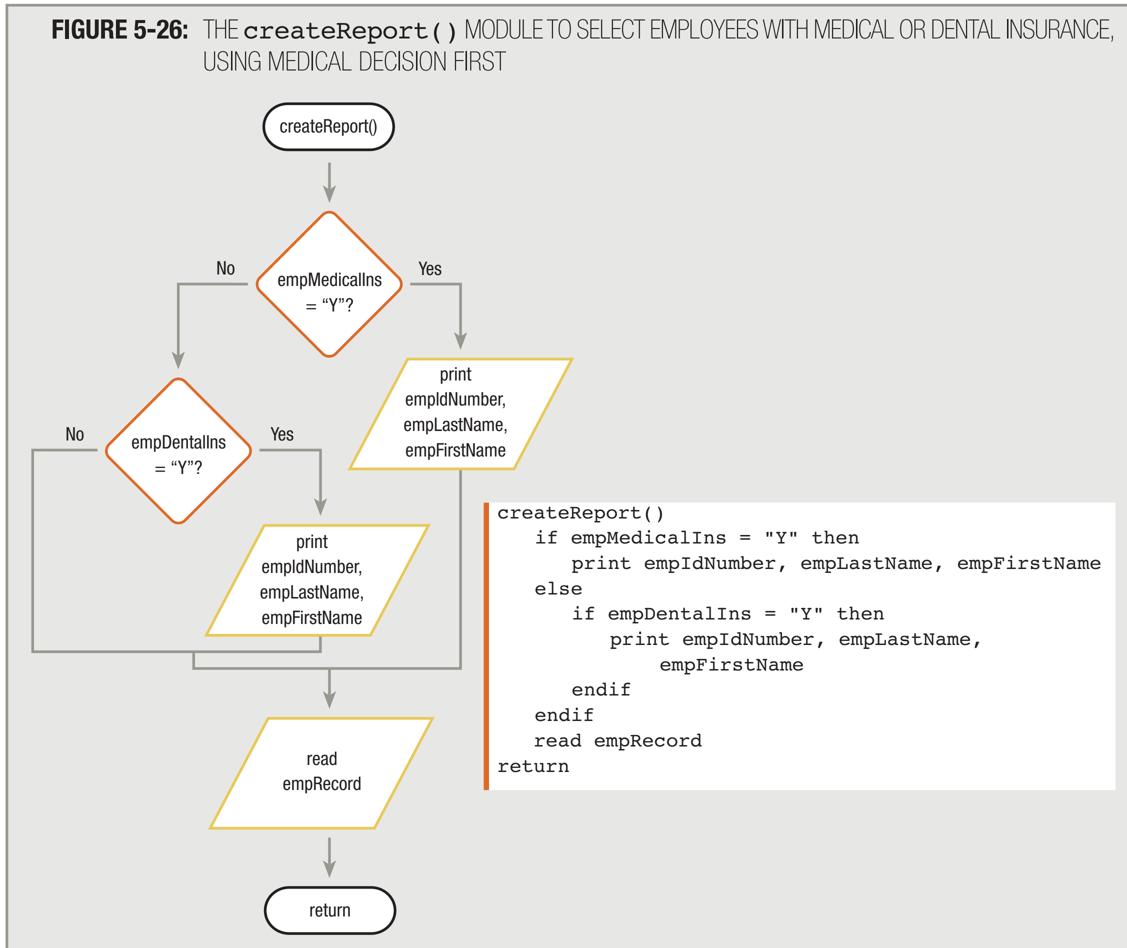
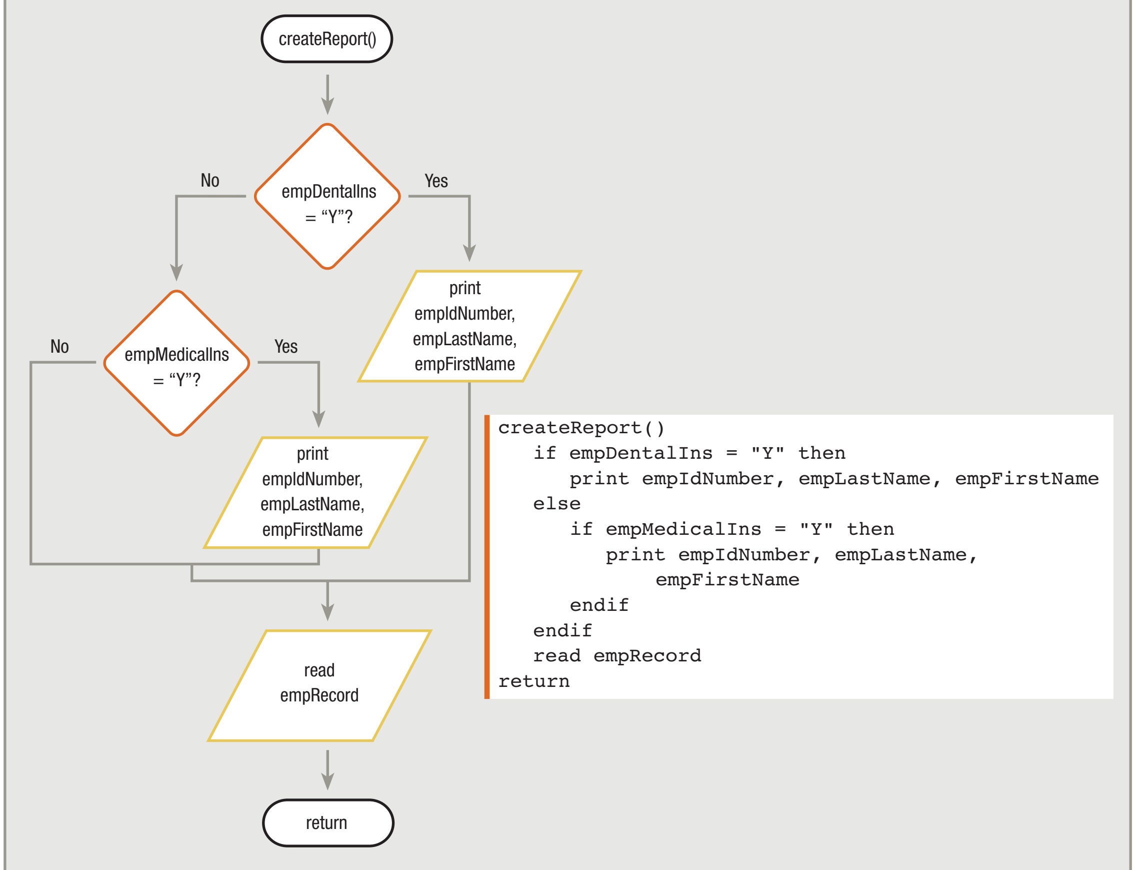


FIGURE 5-27: ALTERNATE `createReport()` MODULE TO SELECT EMPLOYEES WITH MEDICAL OR DENTAL INSURANCE, USING DENTAL DECISION FIRST




```
createReport()
   if empDentalIns = "Y" then
      print empIdNumber, empLastName, empFirstName
   else
      if empMedicalIns = "Y" then
         print empIdNumber, empLastName,
             empFirstName
      endif
   endif
   read empRecord
return
```

You might have guessed that one of these selections is superior to the other, if you have some background information about the relative likelihood of each condition you are testing. For example, once again assume you know that out of 1,000 employees in your company, about 90 percent, or 900, participate in the medical insurance plan, and about half, or 500, participate in the dental plan.

When you use the logic shown in Figure 5-26 to select employees who participate in either insurance plan, you first ask about medical insurance. For 900 employees, the answer is true; you print these employee records. Only about 100 records continue to the next question regarding dental insurance, where about half, or 50, fulfill the requirements to print. In the end, you print about 950 employees.

If you use Figure 5-27, you ask `empDentalIns = "Y"?` first. The result is true for 50 percent, or 500 employees, whose names then print. Five hundred employee records then progress to the medical insurance question, after which 90 percent, or 450, of them print.

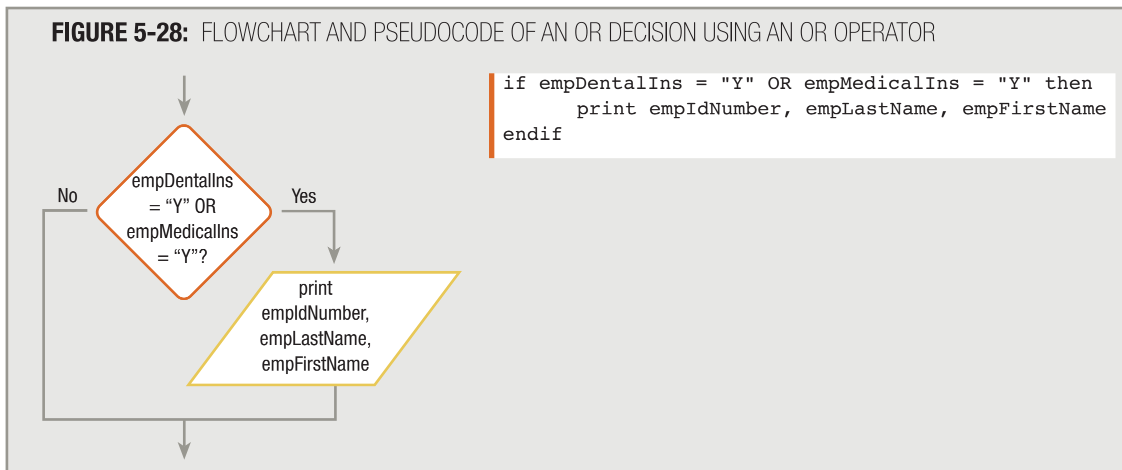
Using either scenario, 950 employee records appear on the list, but the logic used in Figure 5-26 requires 1,100 decisions, whereas the logic used in Figure 5-27 requires 1,500 decisions. The general rule is: *In an OR decision, first ask the question that is more likely to be true.* Because a record qualifies for printing as soon as it passes one test, asking the more likely question first eliminates as many records as possible from having to go through the second decision. The time it takes to execute the program is decreased.

COMBINING DECISIONS IN AN OR SELECTION

When you need to take action when either one or the other of two conditions is met, you can use two separate, nested selection structures, as in the previous examples. However, most programming languages allow you to ask two or more questions in a single comparison by using a **logical OR operator**—for example, `empDentalIns = "Y" OR empMedicalIns = "Y"`. When you use the logical OR operator, only one of the listed conditions must be met for the resulting action to take place. If the programming language you use allows this construct, you still must realize that the question you place first is the question that will be asked first, and cases eliminated by the first question will not proceed to the second question. The computer can ask only one question at a time; even when you draw the flowchart in Figure 5-28, the computer will execute the logic in the flowchart in Figure 5-29.

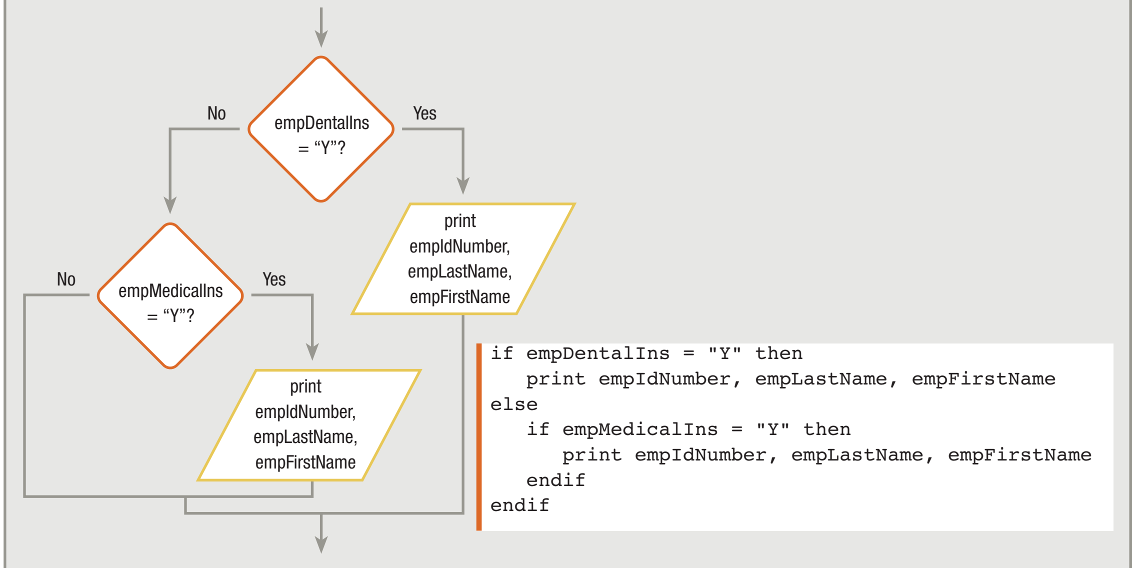
TIP C#, C++, C, and Java use the symbol || to represent the logical OR.

FIGURE 5-28: FLOWCHART AND PSEUDOCODE OF AN OR DECISION USING AN OR OPERATOR




```
if empDentalIns = "Y" OR empMedicalIns = "Y" then
      print empIdNumber, empLastName, empFirstName
endif
```

FIGURE 5-29: FLOWCHART AND PSEUDOCODE OF COMPUTER LOGIC OF PROGRAM CONTAINING AN OR OPERATOR IN THE DECISION; THE COMPUTER STILL MAKES TWO SEPARATE DECISIONS EVEN THOUGH AN OR OPERATOR IS USED




```
if empDentalIns = "Y" then
   print empIdNumber, empLastName, empFirstName
else
   if empMedicalIns = "Y" then
      print empIdNumber, empLastName, empFirstName
   endif
endif
```

USING SELECTIONS WITHIN RANGES

Business programs often need to make selections based on a variable falling within a range of values. For example, suppose you want to print a list of all employees and the names of their supervisors. An employee's supervisor is assigned according to the employee's department number, as shown in Figure 5-30.

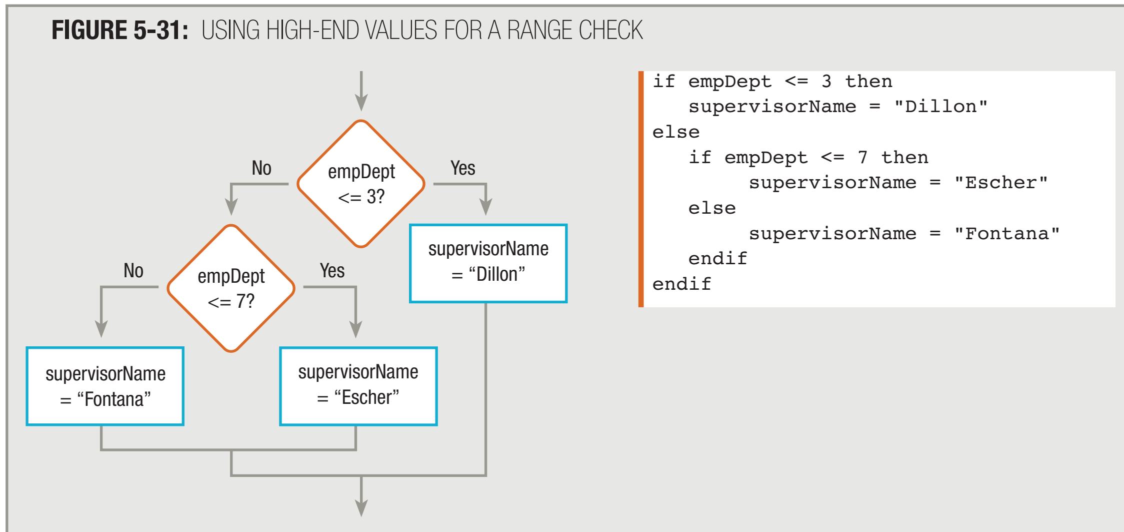
FIGURE 5-30: SUPERVISORS BY DEPARTMENT

DEPARTMENT NUMBER	SUPERVISOR
1–3	Dillon
4–7	Escher
8–9	Fontana

When you write the program that reads each employee's record, you could make nine decisions before printing the supervisor's name, such as `empDept = 1?`, `empDept = 2?`, and so on. However, it is more convenient to find the supervisor by using a range check.

When you use a **range check**, you compare a variable to a series of values between limits. To perform a range check, make comparisons using either the lowest or highest value in each range of values you are using. For example, to find each employee's supervisor as listed in Figure 5-30, either use the values 1, 4, and 8, which represent the low ends of each supervisor's department range, or use the values 3, 7, and 9, which represent the high ends.

Figure 5-31 shows the flowchart and pseudocode that represent the logic for choosing a supervisor name by using the high-end range values. You test the `empDept` value for less than or equal to the high end of the lowest range group. If the comparison evaluates as true, you know the intended value of `supervisorName`. If not, you continue checking.




```
if empDept <= 3 then
   supervisorName = "Dillon"
else
   if empDept <= 7 then
         supervisorName = "Escher"
   else
         supervisorName = "Fontana"
   endif
endif
```

FIGURE 5-31: USING HIGH-END VALUES FOR A RANGE CHECK

TIP In Figure 5-31, notice how each `else` aligns vertically with its corresponding `if`.

For example, consider records containing three different values for `empDept`, and compare how they would be handled by the set of decisions in Figure 5-31.

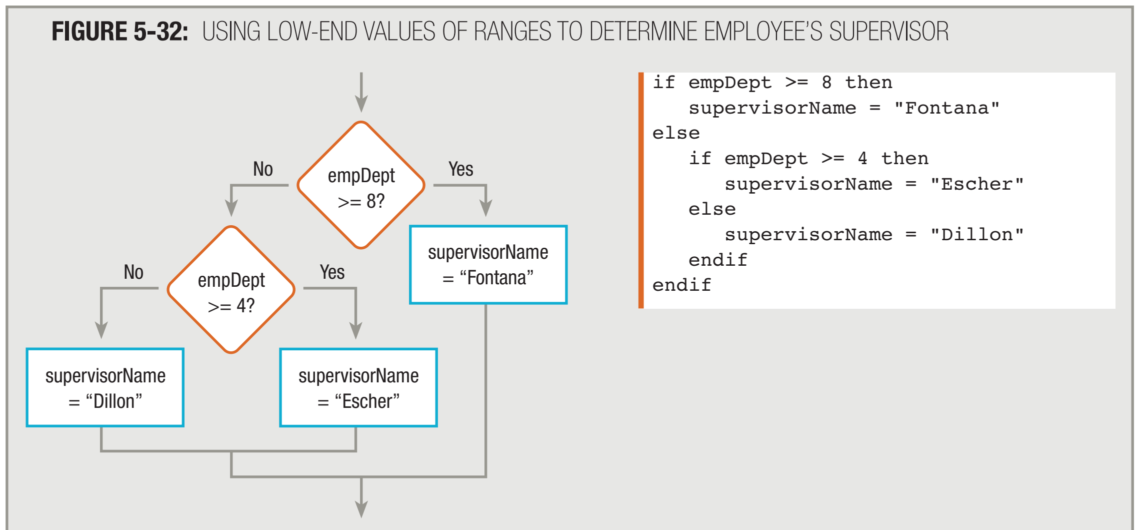
- First, assume that the value of `empDept` for a record is 2. Using the logic in Figure 5-31, the value of the Boolean expression `empDept <= 3` is true, `supervisorName` is set to "Dillon", and the `if` structure ends. In this case, the second decision, `empDept <= 7`, is never made, because the `else` half of `empDept <= 3` never executes.
- Next, assume that for another record, the value of `empDept` is 7. Then, `empDept <= 3` evaluates as false, so the `else` clause of the decision executes. There, `empDept <= 7` is evaluated, and found to be true, so `supervisorName` becomes "Escher".
- Finally, assume that the value of `empDept` is 9. In this case, the first decision, `empDept <= 3`, is false, so the `else` clause executes. Then, the second decision, `empDept <=7`, also evaluates as false, so the `else` clause of the second decision executes, and `supervisorName` is set to "Fontana". In this example, "Fontana" can be called a **default value**, because if neither of the two decision expressions is true, `supervisorName` becomes "Fontana" by default. A default value is the value assigned after a series of selections are all false.

TIP

Using the logic in Figure 5-31, `supervisorName` becomes "Fontana" even if `empDept` is a high, invalid value such as 10, 12, or even 300. The example is intended to be simple, using only two decisions. However, in a business application, you might consider amending the logic so an additional, third decision is made that compares `empDept` less than or equal to 9. Then, you could assign "Fontana" as the supervisor name if `empDept` is less than or equal to 9, and issue an error message if `empDept` is not. You might also want to insert a similar decision at the beginning of the program segment to make sure `empDept` is not less than 1.

The flowchart and pseudocode for choosing a supervisor name using the reverse of this method, by comparing the employee department to the low end of the range values that represent each supervisor's area, appear in Figure 5-32. Using the technique shown in Figure 5-32, you compare `empDept` to the low end (8) of the highest range (8 to 9) first; if `empDept` falls in the range, `supervisorName` is known; otherwise, you check the next lower group. In this example, "Dillon" becomes the default value. That is, if the department number is not greater than or equal to 8, and it is also not greater than or equal to 4, then by default, `supervisorName` is set to "Dillon".

FIGURE 5-32: USING LOW-END VALUES OF RANGES TO DETERMINE EMPLOYEE'S SUPERVISOR




```
if empDept >= 8 then
   supervisorName = "Fontana"
else
   if empDept >= 4 then
      supervisorName = "Escher"
   else
      supervisorName = "Dillon"
   endif
endif
```

COMMON ERRORS USING RANGE CHECKS

Two common errors that occur when programmers perform range checks both entail doing more work than is necessary. Figure 5-33 shows a range check in which the programmer has asked one question too many. If you know that all `empDept` values are positive numbers, then if `empDept` is not greater than or equal to 8, and it is also not greater than or equal to 4, then by default it must be greater than or equal to 1. Asking whether `empDept` is greater than or equal to 1 is a waste of time; no employee record can ever travel the logical path on the far left. You might say that the path that can never be traveled is a **dead** or **unreachable path**, and that the statements written there constitute dead or unreachable code. Providing such a path is always a logical error.

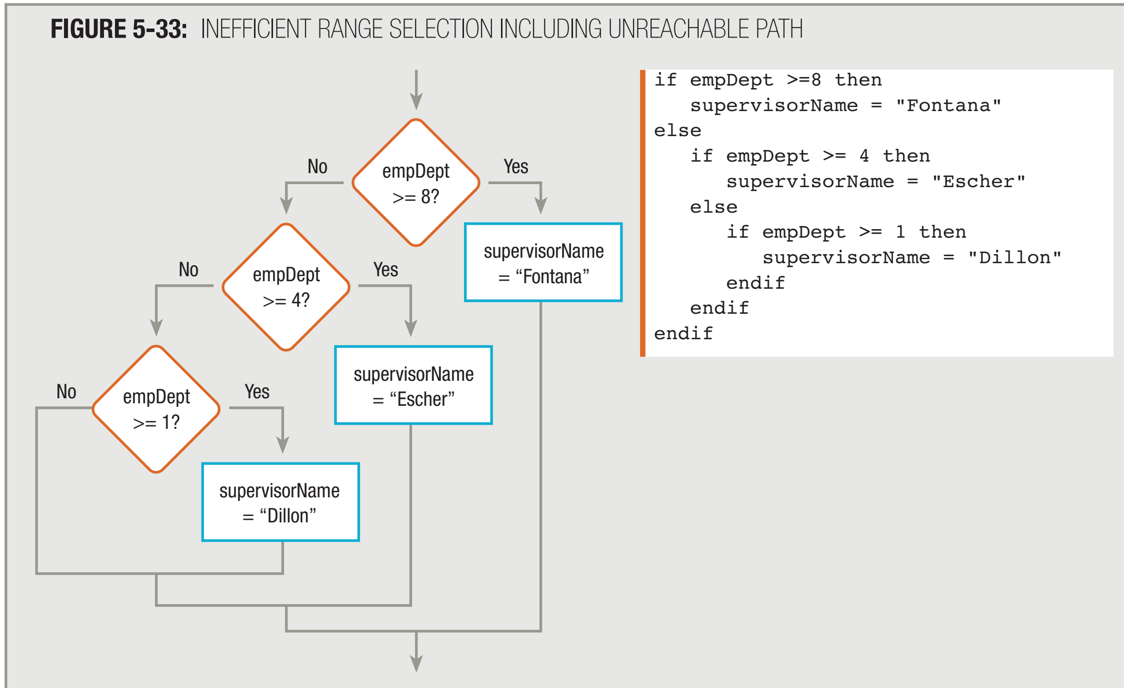


FIGURE 5-33: INEFFICIENT RANGE SELECTION INCLUDING UNREACHABLE PATH

TIP

When you ask questions of human beings, you sometimes ask a question to which you already know the answer. For example, in court, a good trial lawyer seldom asks a question if the answer will be a surprise. With computer logic, however, such questions are an inefficient waste of time.

Another error that programmers make when writing the logic to perform a range check also involves asking unnecessary questions. You should never ask a question if there is only one possible answer or outcome. Figure 5-34 shows an inefficient range selection that asks two unneeded questions. In the figure, if `empDept` is greater than or equal to 8, "Fontana" is the supervisor. If `empDept` is not greater than or equal to 8, then it must be less than 8, so the next question does not have to check for less than 8. The computer logic will never execute the second decision unless `empDept` is already less than 8—that is, unless it follows the false branch of the first selection. If you use the logic in Figure 5-34, you are wasting computer time asking a question that has previously been answered. Similarly, if `empDept` is not greater than or equal to 8 and it is also not greater than or equal to 4, then it must be less than 4. Therefore, there is no reason to compare `empDept` to 4 to determine whether "Dillon" is the supervisor. If the logic makes it past the first two `if` statements in Figure 5-34, then the supervisor must be "Dillon".

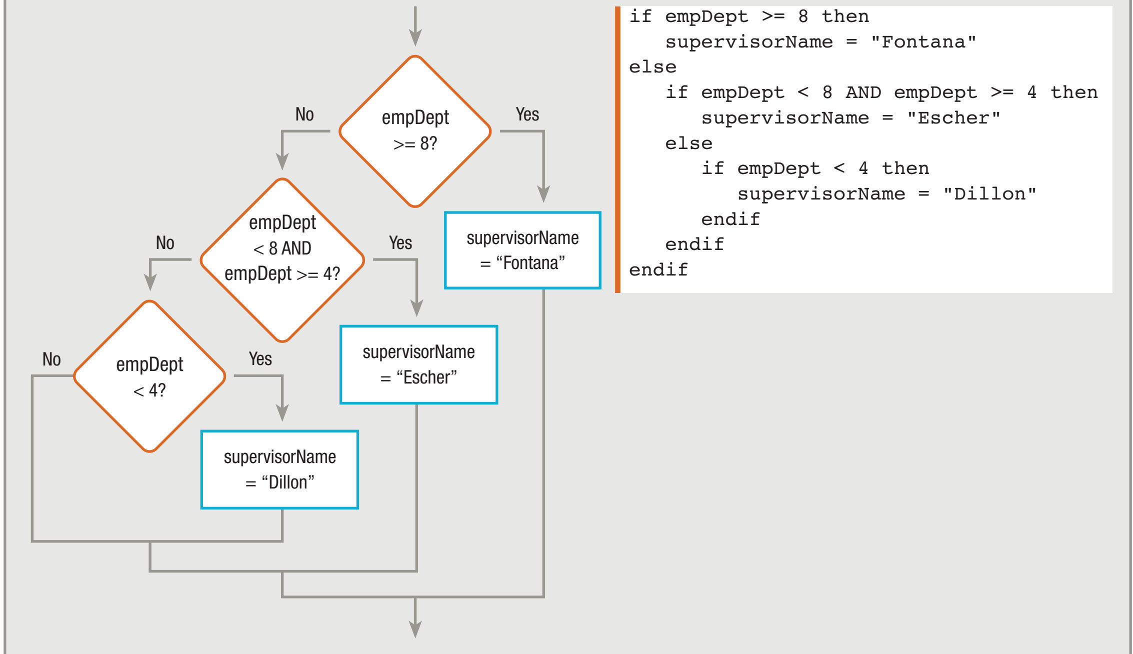


FIGURE 5-34: INEFFICIENT RANGE SELECTION INCLUDING UNNECESSARY QUESTION

TIP Beginning programmers sometimes justify their use of unnecessary questions as "just making really sure." Such caution is unnecessary when writing computer logic.

UNDERSTANDING PRECEDENCE WHEN COMBINING AND AND OR SELECTIONS

Most programming languages allow you to combine as many AND and OR operators in an expression as you need. For example, assume you need to achieve a score of at least 75 on each of three tests in order to pass a course. When multiple conditions must be true before performing an action, you can use an expression like the following:

```
if score1 >= 75 AND score2 >= 75 AND score3 >= 75 then
   classGrade = "Pass"
else
   classGrade = "Fail"
endif
```

On the other hand, if you need to pass only one test in order to pass the course, then the logic is as follows:

```
if score1 >= 75 OR score2 >= 75 OR score3 >= 75 then
   classGrade = "Pass"
else
   classGrade = "Fail"
endif
```

The logic becomes more complicated when you combine AND and OR operators within the same statement. When you combine AND and OR operators, the AND operators take **precedence**, meaning their Boolean values are evaluated first.

For example, consider a program that determines whether a movie theater patron can purchase a discounted ticket. Assume discounts are allowed for children (age 12 and under) and senior citizens (age 65 and older) who attend "G"-rated movies. The following code looks reasonable, but produces incorrect results, because the AND operator evaluates before the OR.

```
if age <= 12 OR age >= 65 AND rating = "G" then
  print "Discount applies"
```

For example, assume a movie patron is 10 years old and the movie rating is "R". The patron should not receive a discount—or be allowed to see the movie! However, within the previous `if` statement, the part of the expression containing the AND, `age >= 65 AND rating = "G"`, evaluates first. For a 10-year-old and an "R"-rated movie, the question is false (on both counts), so the entire `if` statement becomes the equivalent of the following:

```
if age <= 12 OR aFalseExpression
```

Because the patron is 10, `age <= 12` is true, so the original `if` statement becomes the equivalent of:

```
if aTrueExpression OR aFalseExpression
```

which evaluates as true. Therefore, the statement "Discount applies" prints when it should not.

Many programming languages allow you to use parentheses to correct the logic and force the expression `age <= 12 OR age >= 65` to evaluate first, as shown in the following pseudocode:

```
if (age <= 12 OR age >= 65) AND rating = "G" then
   print "Discount applies"
```

With the added parentheses, if the patron's age is 12 or under OR 65 or over, the expression is evaluated as:

```
if aTrueExpression AND rating = "G"
```

When the age value qualifies a patron for a discount, then the rating value must also be acceptable before the discount applies. This was the original intention of the statement.

You always can avoid the confusion of mixing AND and OR decisions by nesting `if` statements instead of using ANDs and ORs. With the flowchart and pseudocode shown in Figure 5-35, it is clear which movie patrons receive the discount. In the flowchart in the figure, you can see that the OR is nested entirely within the Yes branch of the `rating = "G"?` selection. Similarly, by examining the pseudocode in Figure 5-35, you can see by the alignment that if the rating is not “G”, the logic proceeds directly to the last `endif` statement, bypassing any checking of the age at all.

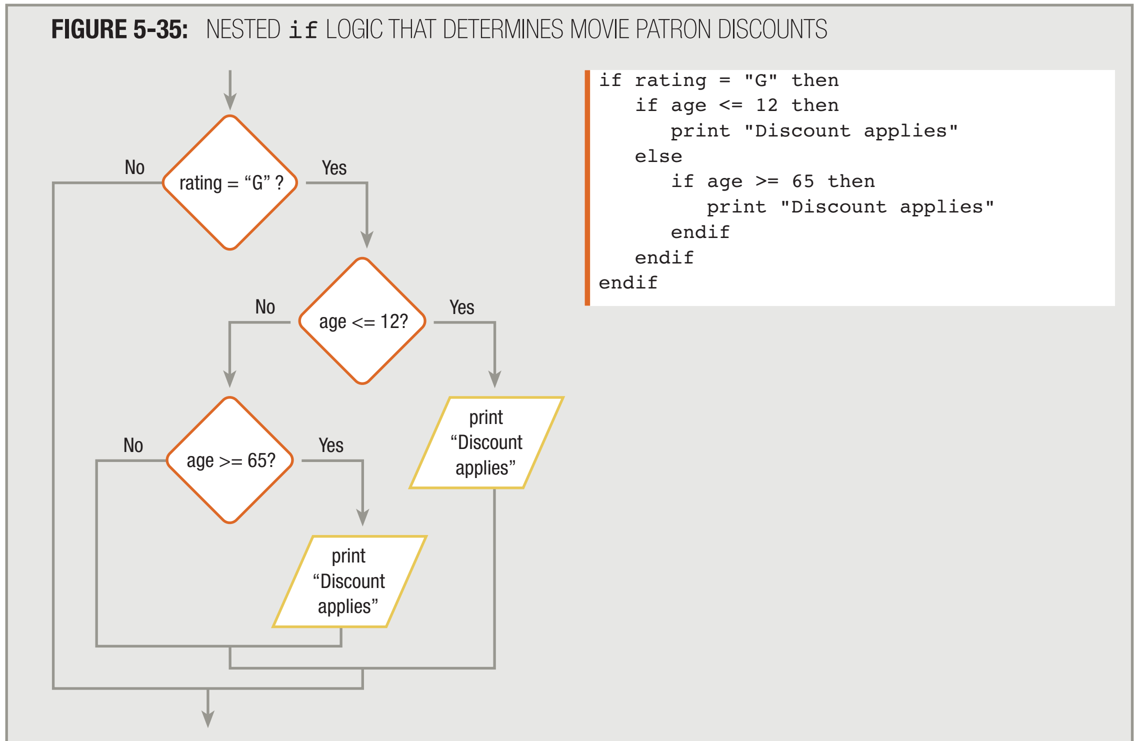


FIGURE 5-35: NESTED `if` LOGIC THAT DETERMINES MOVIE PATRON DISCOUNTS

TIP

In every programming language, multiplication has precedence over addition in an arithmetic statement. That is, the value of 2 + 3 * 4 is 14 because the multiplication occurs before the addition. Similarly, in every programming language, AND has precedence over OR. That’s because computer circuitry treats the AND operator as multiplication and the OR operator as addition. In every programming language, 1 represents true and 0 represents false. So, for example, `aTrueExpression` AND `aTrueExpression` results in true, because 1 * 1 is 1, and `aTrueExpression` AND `aFalseExpression` is false, because 1 * 0 is 0. Similarly, `aFalseExpression` OR `aFalseExpression` AND `aTrueExpression` evaluates to `aFalseExpression` because 0 + 0 * 1 is 0, whereas `aFalseExpression` AND `aFalseExpression` OR `aTrueExpression` evaluates to `aTrueExpression` because 0 * 0 + 1 is 1.

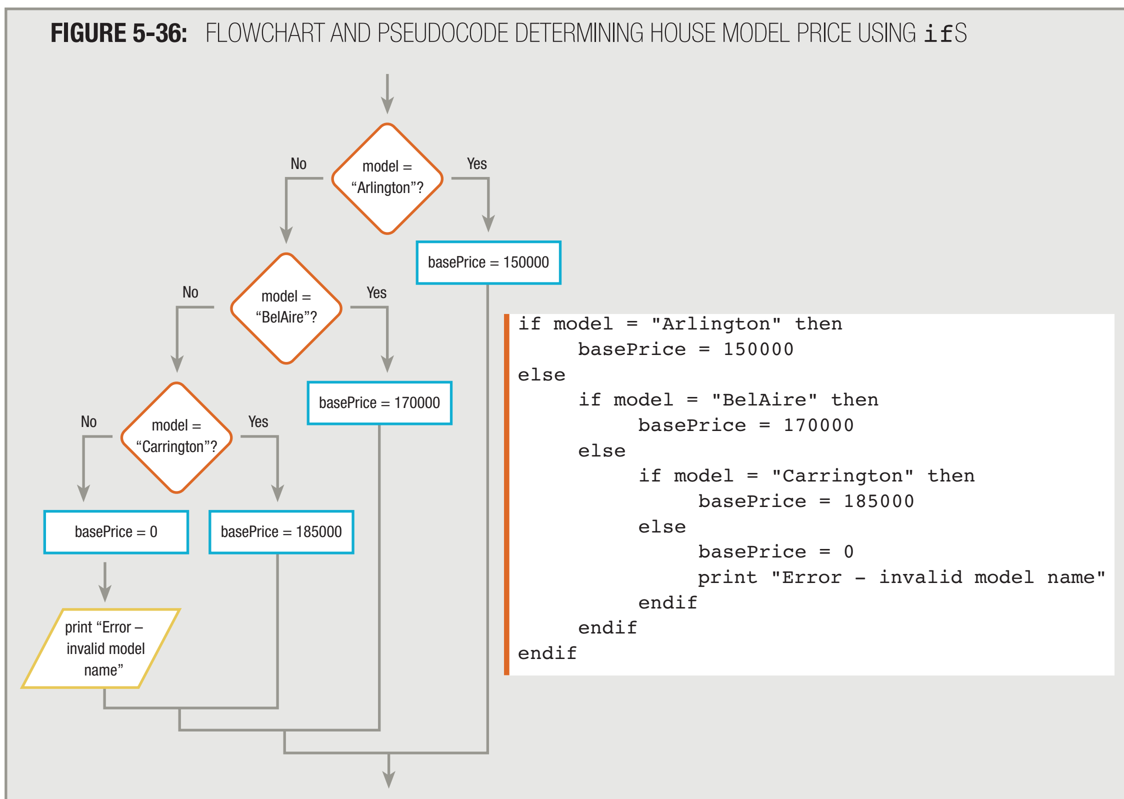
UNDERSTANDING THE CASE STRUCTURE

When you have a series of decisions based on the value stored in a single variable, most languages allow you to use a case structure. You first learned about the case structure in Chapter 2. There, you learned that you can solve any programming problem using only the three basic structures—sequence, selection, and loop. You are never required to use a case structure—you can always substitute a series of nested selections. The **case structure** simply provides a convenient alternative to using a series of decisions when you must make choices based on the value stored in a single variable.

TIP In some languages, the case structure is called the switch statement.

For example, suppose you work for a real estate developer who is selling houses that have one of three different floor plans. The logic segment of a program that determines the base price of the house might look like the logic shown in Figure 5-36.

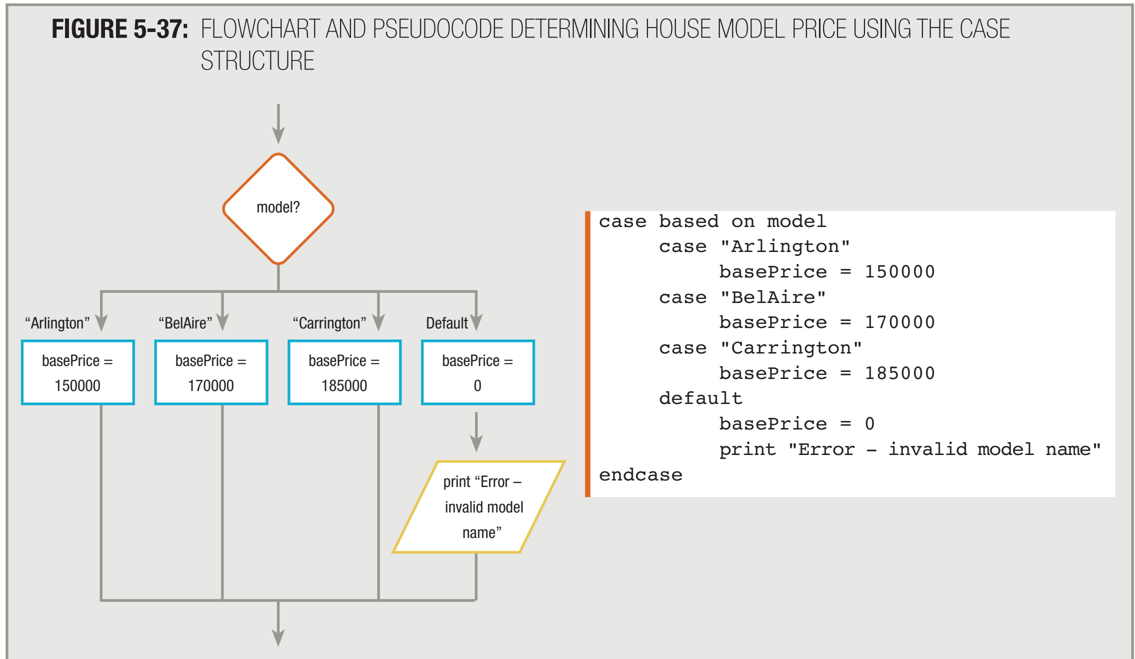
FIGURE 5-36: FLOWCHART AND PSEUDOCODE DETERMINING HOUSE MODEL PRICE USING `if`S




```
if model = "Arlington" then
     basePrice = 150000
else
     if model = "BelAire" then
          basePrice = 170000
     else
          if model = "Carrington" then
               basePrice = 185000
          else
               basePrice = 0
               print "Error – invalid model name"
          endif
     endif
endif
```

The logic shown in Figure 5-36 is completely structured. However, rewriting the logic using a case structure, as shown in Figure 5-37, might make it easier to understand. When using the case structure, you test a variable against a series of values, taking appropriate action based on the variable's value.

FIGURE 5-37: FLOWCHART AND PSEUDOCODE DETERMINING HOUSE MODEL PRICE USING THE CASE STRUCTURE




```
case based on model
     case "Arlington"
          basePrice = 150000
     case "BelAire"
          basePrice = 170000
     case "Carrington"
          basePrice = 185000
     default
          basePrice = 0
          print "Error – invalid model name"
endcase
```

In Figure 5-37, the `model` variable is compared in turn with "Arlington", "BelAire", and "Carrington", and an appropriate `basePrice` value is set. The default case is the case that executes in the event no other cases execute. The logic shown in Figure 5-36 is identical to that shown in Figure 5-37; your choice of method to set the housing model prices is entirely a matter of preference.

When you look at a nested if-else structure containing an outer and inner selection, if the inner nested `if` is within the `if` portion of the outer `if`, the program segment is a candidate for AND logic. On the other hand, if the inner `if` is within the `else` portion of the outer `if`, the program segment might be a candidate for the case structure.

Some languages require a break statement at the end of each case selection segment. In those languages, once a case is true, all the following cases execute until a break statement is encountered. When you study a specific programming language, you will learn how to use break statements if they are required in that language.

USING DECISION TABLES

Some programs require multiple decisions to produce the correct output. Managing all possible outcomes of multiple decisions can be a difficult task, so programmers sometimes use a tool called a decision table to help organize the possible decision outcome combinations.

A **decision table** is a problem-analysis tool that consists of four parts:

- Conditions
- Possible combinations of Boolean values for the conditions
- Possible actions based on the conditions
- The specific action that corresponds to each Boolean value of each condition

For example, suppose a college collects input data like that shown in Figure 5-38. Each student's data record includes the student's age and a variable that indicates whether the student has requested a residence hall that enforces quiet study hours.

FIGURE 5-38: STUDENT RESIDENCE FILE DESCRIPTION

```
STUDENT RESIDENCE FILE DESCRIPTION
File Name: STURESFILE
FIELD DESCRIPTION        DATA TYPE     COMMENTS
ID Number                Numeric       4 digits, 0 decimal places
Last Name                Character     15 characters
First Name               Character     15 characters
Age                      Numeric       0 decimal places
Request for Hall         Character     1 character, Y or N
  with Quiet Hours
```

Assume that the residence hall director makes residence hall assignments based on the following rules:

- Students who are under 21 years old and who request a residence hall with quiet study hours are assigned to Addams Hall.
- Students who are under 21 years old and who do not request a residence hall with quiet study hours are assigned to Grant Hall.
- Students who are 21 years old and over and who request a residence hall with quiet study hours are assigned to Lincoln Hall.
- Students who are 21 years old and over and who do not request a residence hall with quiet study hours are also assigned to Lincoln Hall.

You can create a program that assigns each student to the appropriate residence hall and prints a list of students along with each student's hall assignment. A sample report is shown in Figure 5-39. The mainline logic for this program appears in Figure 5-40. Most programs you write will contain the same basic mainline logic: Each performs start-up or housekeeping tasks, a main loop that acts repeatedly—once for each input record—and a finishing module that performs any necessary program-ending tasks, including closing the open files.

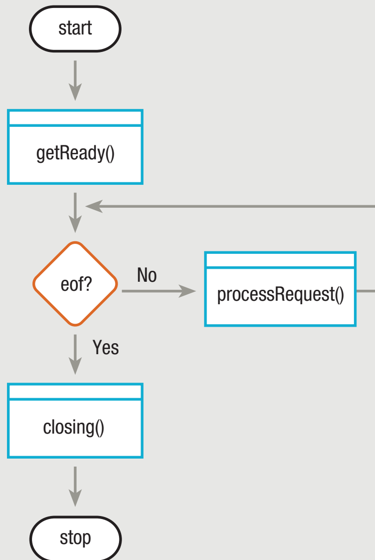
FIGURE 5-39: SAMPLE REPORT LISTING STUDENT RESIDENCE HALL ASSIGNMENTS

```
     Student Residence Hall Assignments

 Student ID      Age     Request for Quiet     Assigned Hall

 1288            21             Y              Lincoln
 1567            20             Y              Addams
 5612            24             N              Lincoln
 7610            18             N              Grant
 7723            20             N              Grant
 8012            19             Y              Addams
```

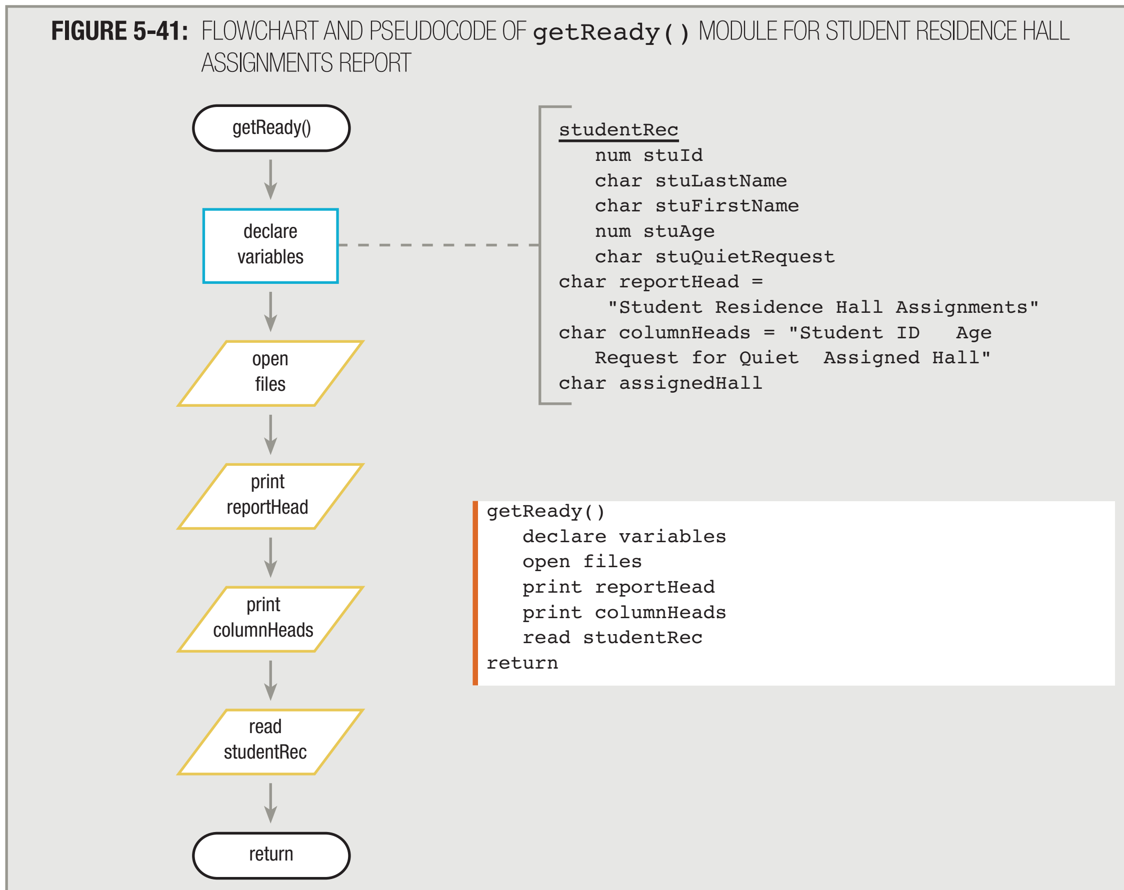
FIGURE 5-40: FLOWCHART AND PSEUDOCODE OF MAINLINE LOGIC FOR STUDENT RESIDENCE HALL ASSIGNMENTS REPORT




```
start
   perform getReady()
   while not eof
      perform processRequest()
   endwhile
   perform closing()
stop
```

The `getReady()` module for the program that produces the residence hall report is shown in Figure 5-41. It declares variables, opens the files, prints the report headings, and reads the first data record into memory.

FIGURE 5-41: FLOWCHART AND PSEUDOCODE OF `getReady()` MODULE FOR STUDENT RESIDENCE HALL ASSIGNMENTS REPORT



Before you draw a flowchart or write the pseudocode for the `processRequest()` module, you can create a decision table to help you manage all the decisions. You can begin to create a decision table by listing all possible conditions. They are:

- `stuAge < 21`, or not
- `stuQuietRequest = "Y"`, or not

Next, determine how many possible Boolean value combinations exist for the conditions. In this case, there are four possible combinations, shown in Figure 5-42. A student can be under 21, request a residence hall with quiet hours, both, or neither. Because each condition has two outcomes and there are two conditions, there are 2 * 2, or four, possibilities. Three conditions would produce eight possible outcome combinations (2 * 2 * 2); four conditions would produce 16 possible outcome combinations (2 * 2 * 2 * 2), and so on.

FIGURE 5-42: POSSIBLE OUTCOMES OF RESIDENCE HALL REQUEST CONDITIONS

Condition	Outcome			
stuAge < 21	T	T	F	F
stuQuietRequest = "Y"	T	F	T	F

Next, add rows to the decision table to list the possible outcome actions. A student might be assigned to Addams, Grant, or Lincoln Hall. Figure 5-43 shows an expanded decision table that includes these three possible outcomes.

FIGURE 5-43: DECISION TABLE INCLUDING POSSIBLE OUTCOMES OF RESIDENCE HALL DECISIONS

Condition	Outcome			
stuAge < 21	T	T	F	F
stuQuietRequest = "Y"	T	F	T	F
assignedHall = "Addams"				
assignedHall = "Grant"				
assignedHall = "Lincoln"				

You choose one required outcome for each possible combination of conditions. As shown in Figure 5-44, you place an X in the Addams Hall row when `stuAge` is less than 21 and the student requests a residence hall with quiet study hours. You place an X in the Grant Hall row when a student is under 21 but does not request a residence hall with quiet hours. Finally, you place Xs in the Lincoln Hall row for both `stuQuietRequest` values when a student is not under 21 years old—only one residence hall is available for students 21 and over, whether they have requested a hall with quiet hours or not.

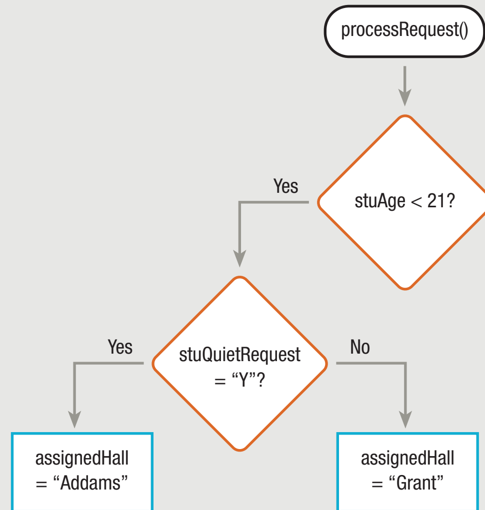
FIGURE 5-44: COMPLETED DECISION TABLE FOR RESIDENCE HALL SELECTION

Condition	Outcome			
stuAge < 21	T	T	F	F
stuQuietRequest = "Y"	T	F	T	F
assignedHall = "Addams"	X			
assignedHall = "Grant"		X		
assignedHall = "Lincoln"			X	X

The decision table is complete (count the Xs—there are four possible outcomes). Take a moment and confirm that each residence hall selection is the appropriate value based on the original specifications. Now that the decision table is complete, you can start to plan the logic.

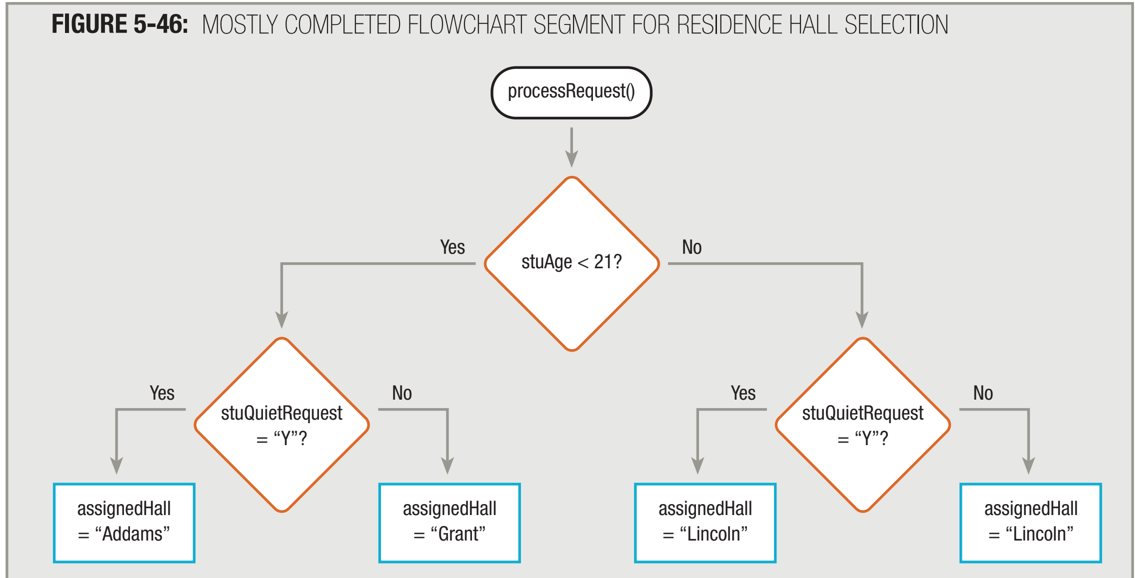
If you choose to use a flowchart to express the logic, you start by drawing a path to the outcome shown in the first column. This result (which occurs when `stuAge < 21` and `stuQuietRequest = "Y"`) sets the residence hall to “Addams”. Next, add the resulting action shown in the second column of the decision table, which occurs when `stuAge < 21` is true and `stuQuietRequest = "Y"` is false. In those cases, the residence hall becomes “Grant”. See Figure 5-45.

FIGURE 5-45: PARTIALLY COMPLETED FLOWCHART SEGMENT FOR RESIDENCE HALL SELECTION



Next, on the false outcome side of the `stuAge < 21` question, you add the resulting action shown in the third column of the decision table—set the residence hall to “Lincoln”. This action occurs when `stuAge < 21` is false and `stuQuietRequest = "Y"` is true. Finally, add the resulting action shown in the fourth column of the decision table, which occurs when both conditions are false. When a student is not under 21 and does not request a hall with quiet study hours, then the assigned hall is “Lincoln”. See Figure 5-46.

FIGURE 5-46: MOSTLY COMPLETED FLOWCHART SEGMENT FOR RESIDENCE HALL SELECTION



The decision making in the flowchart segment is now complete and accurately assigns each student to the correct residence hall. To finish it, all you need to do is tie up the loose ends of the decision structure, print a student's ID number and residence hall assignment, and read the next record. However, if you examine the two result boxes on the far right in Figure 5-46, you see that the assigned residence hall is identical—"Lincoln" in both cases. When a student is not under 21, whether the `stuQuietRequest` equals "Y" or not, the residence hall assignment is the same; therefore, there is no point in asking the `stuQuietRequest` question. Additionally, many programmers prefer that the True or Yes side of a flowchart decision always appears on the right side of a flowchart. Figure 5-47 shows the complete residence hall assignment program, including the redrawn `processRequest()` module, which has only one "Lincoln" assignment statement and True results to the right of each selection. Figure 5-47 also shows the pseudocode for the same problem.

Perhaps you could have created the final decision-making `processRequest()` module without creating the decision table first. If so, you need not use the table. Decision tables are more useful to the programmer when the decision-making process becomes more complicated. Additionally, they serve as a useful graphic tool when you want to explain the decision-making process of a program to a user who is not familiar with flowcharting symbols.

TIP In Appendix C, you can walk through the process used to create a larger decision table.

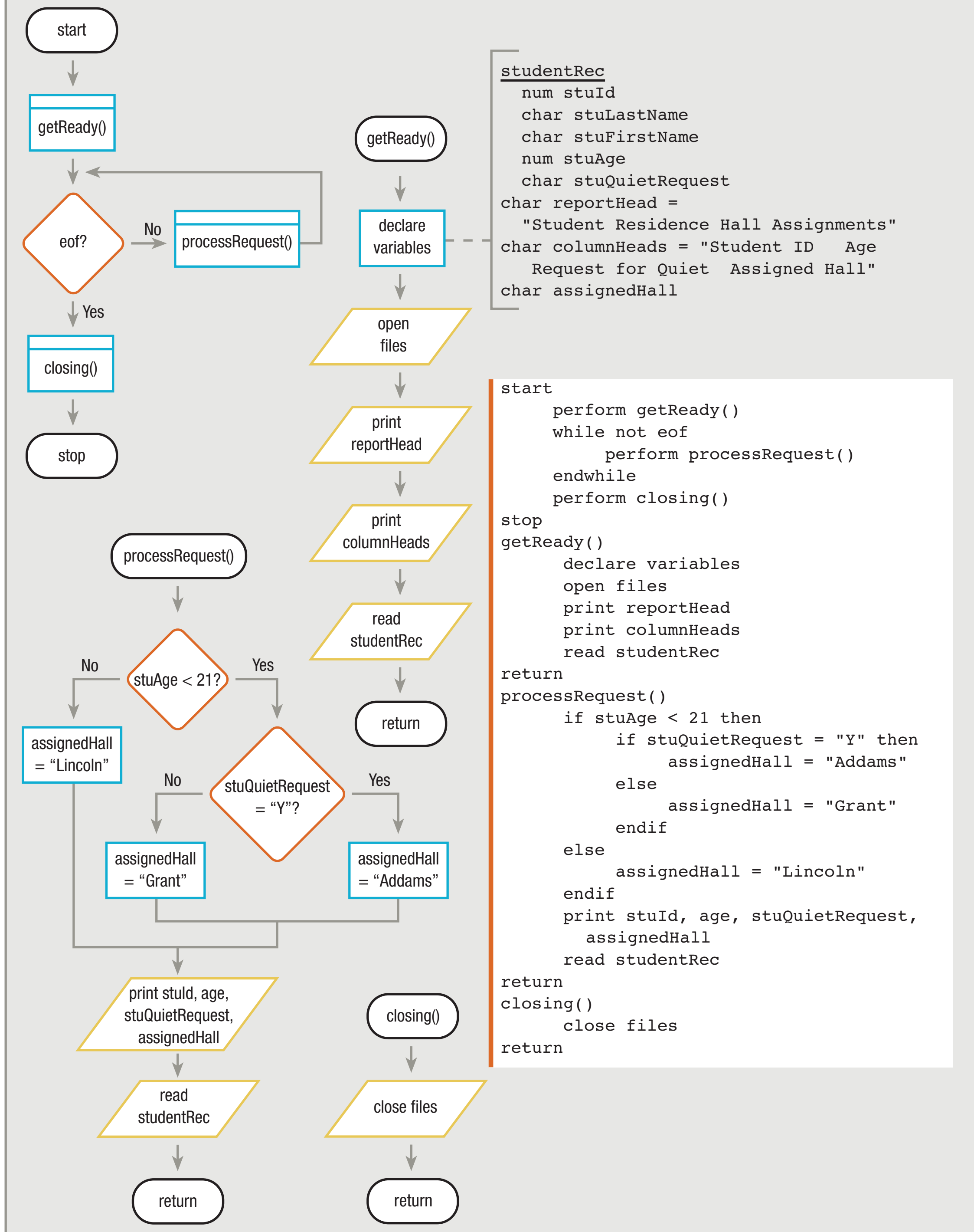


FIGURE 5-47: COMPLETE FLOWCHART AND PSEUDOCODE FOR RESIDENCE HALL SELECTION PROBLEM

CHAPTER SUMMARY

- Every decision you make in a computer program involves evaluating a Boolean expression. You can use dual-alternative, or binary, selections or if-then-else structures to choose between two possible outcomes. You also can use single-alternative, or unary, selections or if-then structures when there is only one outcome for the question where an action is required.
- For any two values that are the same type, you can use relational comparison operators to decide whether the two values are equal, the first value is greater than the second value, or the first value is less than the second value. The two values used in a Boolean expression can be either variables or constants.
- An AND decision occurs when two conditions must be true in order for a resulting action to take place. An AND decision requires a nested decision, or a nested `if`.
- In an AND decision, first ask the question that is less likely to be true. This eliminates as many records as possible from having to go through the second decision, which speeds up processing time.
- Most programming languages allow you to ask two or more questions in a single comparison by using a logical AND operator.
- When you must satisfy two or more criteria to initiate an event in a program, you must make sure that the second decision is made entirely within the first decision, and that you use a complete Boolean expression on both sides of the AND.
- An OR decision occurs when you want to take action when one or the other of two conditions is true.
- Errors occur in OR decisions when programmers do not maintain structure. An additional source of errors that are particular to the OR selection stems from people using the word AND to express OR requirements.
- In an OR decision, first ask the question that is more likely to be true.
- Most programming languages allow you to ask two or more questions in a single comparison by using a logical OR operator.
- To perform a range check, make comparisons with either the lowest or highest value in each range of values you are using.
- Common errors that occur when programmers perform range checks include asking unnecessary and previously answered questions.
- The case structure provides a convenient alternative to using a series of decisions when you must make choices based on the value stored in a single variable.
- A decision table is a problem-analysis tool that consists of conditions, possible combinations of Boolean values for the conditions, possible actions based on the conditions, and the action that corresponds to each Boolean value of each condition.

KEY TERMS

A **dual-alternative**, or **binary**, selection structure offers two actions, each associated with one of two possible outcomes. It is also called an **if-then-else** structure.

In a **single-alternative**, or **unary**, selection structure, an action is required for only one outcome of the question. You call this form of the selection structure an **if-then**, because no “else” action is necessary.

The **if clause** of a decision holds the action or actions that execute when a Boolean expression in a decision is true.

The **else clause** of a decision holds the action or actions that execute when the Boolean expression in a decision is false.

A **Boolean expression** is one that represents only one of two states, usually expressed as true or false.

A **trivial** Boolean expression is one that always evaluates to the same result.

Relational comparison operators are the symbols that express Boolean comparisons. Examples include =, >, <, >=, <=, and <>.

A **logical operator** (as the term is most often used) compares single bits. However, some programmers use the term synonymously with “relational comparison operator.”

With an **AND decision**, two conditions must both be true for an action to take place. An AND decision requires a **nested decision**, or a **nested if**—that is, a decision “inside of” another decision. A series of nested `if` statements can also be called a **cascading if statement**.

A **logical AND operator** is a symbol that you use to combine decisions so that two (or more) conditions must be true for an action to occur.

Short-circuiting is the compiler technique of not evaluating an expression when the outcome makes no difference.

A **range** of values encompasses every value between a high and low limit.

An **OR decision** contains two (or more) decisions; if at least one condition is met, the resulting action takes place.

A **logical OR operator** is a symbol that you use to combine decisions when any one condition can be true for an action to occur.

When you use a **range check**, you compare a variable to a series of values between limits.

A **default value** is one that is assigned after all test conditions are found to be false.

A **dead** or **unreachable path** is a logical path that can never be traveled.

When an operator has **precedence**, it is evaluated before others.

The **case structure** provides a convenient alternative to using a series of decisions when you must make choices based on the value stored in a single variable.

A **decision table** is a problem-analysis tool that consists of four parts: conditions, possible combinations of Boolean values for the conditions, possible actions based on the conditions, and the specific action that corresponds to each Boolean value of each condition.

REVIEW QUESTIONS

1. **The selection statement** `if quantity > 100 then discountRate = 0.20` **is an example of a __________.**
   a. single-alternative selection
   b. dual-alternative selection
   c. binary selection
   d. all of the above

2. **The selection statement** `if dayOfWeek = "S" then price = 5.00 else price = 6.00` **is an example of a __________.**
   a. unary selection
   b. single-alternative selection
   c. binary selection
   d. all of the above

3. **All selection statements must have __________.**
   a. an `if` clause
   b. an `else` clause
   c. both of these
   d. neither a nor b

4. **An expression like** `amount < 10` **is a __________ expression.**
   a. Gregorian
   b. Boolean
   c. unary
   d. binary

5. **Usually, you compare only variables that have the same __________.**
   a. value
   b. size
   c. name
   d. type

6. **Symbols like > and < are known as __________ operators.**
   a. arithmetic
   b. relational comparison
   c. sequential
   d. scripting accuracy

7. **If you could use only three relational comparison operators, you could get by with __________.**
   a. greater than, less than, and greater than or equal to
   b. less than, less than or equal to, and not equal to
   c. equal to, less than, and greater than
   d. equal to, not equal to, and less than

8. **If `a > b` is false, then which of the following is always true?**
   a. `a < b`
   b. `a <= b`
   c. `a = b`
   d. `a >= b`

9. **Usually, the most difficult comparison operator to work with is ________.**
   a. equal to
   b. greater than
   c. less than
   d. not equal to

10. **Which of the lettered choices is equivalent to the following decision?**

```
if x > 10 then
   if y > 10 then
      print "X"
   endif
endif
```

   a. `if x > 10 AND y > 10 then print "X"`
   b. `if x > 10 OR y > 10 then print "X"`
   c. `if x > 10 AND x > y then print "X"`
   d. `if y > x then print "X"`

11. **The Midwest Sales region of Acme Computer Company consists of five states—Illinois, Indiana, Iowa, Missouri, and Wisconsin. Suppose you have input records containing Acme customer data, including state of residence. To most efficiently select and display all customers who live in the Midwest Sales region, you would use ________.**
   a. five completely separate unnested `if` statements
   b. nested `if` statements using AND logic
   c. nested `if` statements using OR logic
   d. Not enough information is given.

12. **The Midwest Sales region of Acme Computer Company consists of five states—Illinois, Indiana, Iowa, Missouri, and Wisconsin. About 50 percent of the regional customers reside in Illinois, 20 percent in Indiana, and 10 percent in each of the other three states. Suppose you have input records containing Acme customer data, including state of residence. To most efficiently select and display all customers who live in the Midwest Sales region, you would ask first about residency in ________.**
   a. Illinois
   b. Indiana
   c. Wisconsin
   d. either Iowa, Missouri, or Wisconsin—it does not matter which one is first

13. The Boffo Balloon Company makes helium balloons. Large balloons cost $13 a dozen, medium-sized balloons cost $11 a dozen, and small balloons cost $8.60 a dozen. About 60 percent of the company's sales are the smallest balloons, 30 percent are the medium, and large balloons constitute only 10 percent of sales. Customer order records include customer information, quantity ordered, and size. When you write a program to determine price based on size, for the most efficient decision, you should ask first whether the size is ________.

a. large
b. medium
c. small
d. It does not matter.

14. The Boffo Balloon Company makes helium balloons in three sizes, 12 colors, and with a choice of 40 imprinted sayings. As a promotion, the company is offering a 25 percent discount on orders of large, red "Happy Valentine's Day" balloons. To most efficiently select the orders to which a discount applies, you would use ________.

a. three completely separate unnested `if` statements
b. nested `if` statements using AND logic
c. nested `if` statements using OR logic
d. Not enough information is given.

15. Radio station FM-99 keeps a record of every song played on the air in a week. Each record contains the day, hour, and minute the song started, and the title and artist of the song. The station manager wants a list of every title played during the important 8 a.m. commute hour on the two busiest traffic days, Monday and Friday. Which logic would select the correct titles?

a.
```
if day = "Monday" OR day = "Friday" OR hour = 8 then
   print title
endif
```
b.
```
if day = "Monday" then
   if hour = 8 then
      print title
   else
      if day = "Friday" then
         print title
      endif
   endif
endif
```
c.
```
if hour = 8 AND day = "Monday" OR day = "Friday" then
   print title
endif
```
d.
```
if hour = 8 then
   if day = "Monday" OR day = "Friday" then
      print title
   endif
endif
```

16. In the following pseudocode, what percentage raise will an employee in Department 5 receive?

```
if department < 3 then
   raise = 25
else
   if department < 5 then
     raise = 50
   else
     raise = 75
   endif
endif
```

a. 25
b. 50
c. 75
d. impossible to tell

17. In the following pseudocode, what percentage raise will an employee in Department 8 receive?

```
if department < 5 then
   raise = 100
else
   if department < 9 then
     raise = 250
   else
      if department < 14 then
        raise = 375
      endif
   endif
endif
```

a. 100
b. 250
c. 375
d. impossible to tell

18. In the following pseudocode, what percentage raise will an employee in Department 10 receive?

```
if department < 2 then
   raise = 1000
else
   if department < 6 then
     raise = 2500
   else
      if department < 10 then
        raise = 3000
      endif
   endif
endif
```

a. 1000
b. 2500
c. 3000
d. impossible to tell

19. When you use a range check, you compare a variable to the ________ value in the range.

a. lowest
b. middle
c. highest
d. lowest or highest

20. Which of the following is not a part of a decision table?

a. conditions
b. declarations
c. possible actions
d. specific actions that will take place under given conditions

FIND THE BUGS

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

1. This pseudocode should create a report containing annual profit statistics for a retail store. Input records contain a department name (for example, "Cosmetics") and profits for each quarter for the last two years. For each quarter, the program should determine whether the profit is higher, lower, or the same as in the same quarter of the previous year. Additionally, the program should determine whether the annual profit is higher, lower, or the same as in the previous year. For example, the line that displays the Cosmetics Department statistics might read "Cosmetics Same Lower Lower Higher Higher" if profits were the same in the first quarter as last year, lower in the second and third quarters, but higher in the fourth quarter and for the year as a whole.

```
start
   perform housekeeping()
   while not eof
      perform detrmineProfitStatistics()
   perform finalTasks()
stop

housekeeping()
   declare variables
        profitRec
             char department
             num salesQuarter1ThisYear
             num salesQuarter2ThisYear
             num salesQuarter2ThisYear
             num salesQuarter4ThisYear
             num salesQuarter1LastYear
             num salesQuarter2LastYear
             num salesQuarter3ThisYear
             num salesQuarter4LastYear
             char mainHead = "Profit Report"
             char columnHeaders = "Department     Quarter 1
Quarter 2     Quarter 3     Quarter 4     Over All"
        num totalThisYear
        num totalLastYear
        char word1
        char word2
        char word3
        char word4
        char word5
   open files
   perform printHeadings()
   read profitRec
return

printHeadings()
   print mainHeader
   print columnHeaders
return
```

```
determineProfitStatistics()
   if salesQuarter1ThisYear > salesQuarter1LastYear then
      word1 = "Higher"
   else
      if salesQuarter1ThisYear < salesQuarter2LastYear then
            word1 = "Lower"
      else
            word1 = "Same"
      endif
   endif
   if salesQuarter2ThisYear > salesQuarter3LastYear then
      word2 = "Higher"
   else
      if salesQuarter2LastYear < salesQuarter2LastYear then
            word2 = "Lower"
      else
            word2 = "Equal"
      endif
   endif
   if salesQuarter3ThisYear > salesQuarter3LastYear then
      word3 = "Higher"
   else
      if salesQuarter3ThisYear < salesQuarter3LastYear then
            word2 = "Lower"
      else
            word3 = "Same"
      endif
   endif
   if salesQuarter4ThisYear > salesQuarter4LastYear then
      word4 = "Higher"
   else
      if salesQuarter4LastYear < salesQuarter4LastYear then
            word4 = "Lower"
      else
            word4 = "Same"
      endif
   endif
```

```
      totalThisYear = salesQuarter1ThisYear + salesQuarter1ThisYear +
          salesQuarter3LastYear + salesQuarter4ThisYear
      totalLastYear = salesQuarter1LastYear + salesQuarter1LastYear +
          salesQuarter3LastYear + salesQuarter4LastYear
      if totalThisYear > totalLastYear then
         word5 = "Higher"
      else
         if totalThisYear > totalLastYear then
               word5 = "Lower"
         else
               word5 = "Same"
        endif
      endif
      print department, word1, word2, word3, word4, word5
      read profitRec
   return

   finalTasks()
      close files
   return
```

2. **This pseudocode should create a report containing rental agents' commissions at an apartment complex. Input records contain an apartment number, the ID number and name of the agent who rented the apartment, and the number of bedrooms in the apartment. The commission is $100 for renting a three-bedroom apartment, $75 for renting a two-bedroom apartment, $55 for renting a one-bedroom apartment, and $30 for renting a studio (zero-bedroom) apartment. Each report line should list the apartment number, the salesperson's name and ID number, and the commission earned on the rental.**

```
start
   perform housekeeping()
   while not eof
      perform calculateCommission()
   perform finishUp()
stop
```

```
housekeeping()
   declare variables
        rentalRecord
            num apartmentNum
            num salesPersonID
            char salesPersonName
            num numBedrooms
       char mainHeader = "Commission Report"
       char columnHeaders = "Apartment number     Salesperson ID
          Name     Commission Earned"
       num comm3Bedroom = 100.00
       num comm2Bedroom = 75.00
       num comm1Bedroom = 55.00
       num commStudio = 30.00
   open files
   perform displayHeaders()
   read rentalRecord
stop

displayHeader()
   print mainHeader
   print columnHeaders
return

calculateCommission()
   if numBedrooms = 3 then
      commissionEarned = comm3Bedroom
   else
      if numBedrooms = 3 then
         commissionEarned = comm3Bedroom
      else
         if numBedrooms = 3 then
              commission = comm3Bedroom
         else
             commissionEarned = comStudio
         endif
   endif
   print apartmentNum, salesPersonID, salesPersonName,
      commissionEarned
   read rentalRecord
return

finishUp()
   close files
return
```

EXERCISES

1. **Assume that the following variables contain the values shown:**

```
numberRed = 100    numberBlue = 200    numberGreen = 300
wordRed = "Wagon"  wordBlue = "Sky"    wordGreen = "Grass"
```

For each of the following Boolean expressions, decide whether the statement is true, false, or illegal.

a. `numberRed = numberBlue?`
b. `numberBlue > numberGreen?`
c. `numberGreen < numberRed?`
d. `numberBlue = wordBlue?`
e. `numberGreen = "Green"?`
f. `wordRed = "Red"?`
g. `wordBlue = "Blue"?`
h. `numberRed <= numberGreen?`
i. `numberBlue >= 200?`
j. `numberGreen >= numberRed + numberBlue?`

2. **A candy company wants a list of its best-selling items, including the item number and the name of candy. Best-selling items are those that sell over 2,000 pounds per month. Input records contain fields for the item number (three digits), the name of the candy (20 characters), the price per pound (four digits, two assumed decimal places), and the quantity in pounds sold last month (four digits, no decimals).**

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Draw the flowchart for this program.
d. Write the pseudocode for this program.

3. **The same candy company described in Exercise 2 wants a list of its high-priced, best-selling items. Best-selling items are those that sell over 2,000 pounds per month. High-priced items are those that sell for $10 per pound or more.**

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Draw the flowchart for this program.
d. Write the pseudocode for this program.

4. **The Literary Honor Society needs a list of English majors who have a grade point average of 3.5 or higher. The student record file includes students' last names and first names, major (for example, "History" or "English"), and grade point average (for example, 3.9 or 2.0).**

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Draw the flowchart for this program.
d. Write the pseudocode for this program.

5. **A telephone company charges 10 cents per minute for all calls outside the customer's area code that last over 20 minutes. All other calls are 13 cents per minute. The phone company has a file with one record for every call made in one day. (In other words, a single customer might have many such records on file.) Fields for each call include customer area code (three digits), customer phone number (seven digits), called area code (three digits), called number (seven digits), and call time in minutes (never more than four digits). The company wants a report listing one detail line for each call, including the customer area code and number, the called area code and number, the minutes, and the total charge.**
   a. Design the output for this program; create either sample output or a print chart.
   b. Draw the hierarchy chart for this program.
   c. Create a decision table to use while planning the logic for this program.
   d. Draw the flowchart for this program.
   e. Write the pseudocode for this program.

6. **A nursery maintains a file of all plants in stock. Each record contains the name of a plant, its price, and fields that indicate the plant's light and soil requirements. The light field contains either "sunny", "partial sun", or "shady". The soil field contains either "clay" or "sandy". Only 20 percent of the nursery stock does well in shade, and 50 percent does well in sandy soil. Customers have requested a report that lists the name and price of each plant that would be appropriate in a shady, sandy yard. Consider program efficiency when designing your solution.**
   a. Design the output for this program; create either sample output or a print chart.
   b. Draw the hierarchy chart for this program.
   c. Create a decision table to use while planning the logic for this program.
   d. Draw the flowchart for this program.
   e. Write the pseudocode for this program.

7. **You have declared variables for an insurance company program as follows:**

```
FIELD                          EXAMPLE
num custPolicyNumber           223356
char custLastName              Salvatore
num custAge                    25
num custDueMonth               06
num custDueDay                 24
num custDueYear                2007
num custAccidents              2
```

   Draw the flowchart or write the pseudocode for the selection structures that print the `custPolicyNumber` and `custLastName` for customers whose data satisfy the following requests for lists of policyholders:
   a. over 35 years old
   b. at least 21 years old
   c. no more than 30 years old
   d. due no later than March 15 any year

e. due up to and including January 1, 2007
f. due by April 27, 2010
g. due as early as December 1, 2006
h. fewer than 11 accidents
i. no more than five accidents
j. no accidents

8. Student files contain an ID number (four digits), last and first names (15 characters each), and major field of study (10 characters). Plan a program that lists ID numbers and names for all French or Spanish majors.

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Create a decision table to use while planning the logic for this program.
d. Draw the flowchart for this program.
e. Write the pseudocode for this program.

9. A florist wants to send coupons to her best customers, so she needs a list of names and addresses for customers who placed orders more than three times last year or spent more than $200 last year. Consider program efficiency when designing your solution. The input file description follows:

```
File name: FLORISTCUSTS
FIELD DESCRIPTION      DATA TYPE          COMMENTS
Customer ID            Numeric            4 digits, 0 decimals
First Name             Character          15 characters
Last Name              Character          15 characters
Street Address         Character          20 characters
Orders Last Year       Numeric            0 decimals
Amount Spent           Numeric            2 decimals
  Last Year
```

(Note: To save room, the record does not include a city or state. Assume that all the florist's best customers are in town.)

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Create a decision table to use while planning the logic for this program.
d. Draw the flowchart for this program.
e. Write the pseudocode for this program.

10. A carpenter needs a program that computes the price of any desk a customer orders, based on the following input fields: order number, desk length and width in inches (three digits each, no decimals), type of wood (20 characters), and number of drawers (two digits). The price is computed as follows:

- The charge for all desks is a minimum $200.
- If the surface (length * width) is over 750 square inches, add $50.
- If the wood is "mahogany", add $150; for "oak", add $125. No charge is added for "pine".
- For every drawer in the desk, there is an additional $30 charge.

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Create a decision table to use while planning the logic for this program.
d. Draw the flowchart for this program.
e. Write the pseudocode for this program.

11. A company is attempting to organize carpools to save energy. Each input record contains an employee's name and town of residence. Ten percent of the company's employees live in Wonder Lake. Thirty percent of the employees live in Woodstock. Because these towns are both north of the company, the company wants a list of employees who live in either town, so it can recommend that these employees drive to work together.

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Create a decision table to use while planning the logic for this program.
d. Draw the flowchart for this program.
e. Write the pseudocode for this program.

12. A supervisor in a manufacturing company wants to produce a report showing which employees have increased their production this year over last year, so that she can issue them a certificate of commendation. She wants to have a report with three columns: last name, first name, and either the word "UP" or blanks printed under the column heading PRODUCTION. "UP" is printed when this year's production is a greater number than last year's production. Input exists as follows:

```
PRODUCTION FILE DESCRIPTION
File name: PRODUCTION
FIELD DESCRIPTION          DATA TYPE     COMMENTS
Last Name                  Character     15 characters
First Name                 Character     15 characters
Last Year's Production     Numeric       0 decimals
This Year's Production     Numeric       0 decimals
```

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Create a decision table to use while planning the logic for this program.
d. Draw the flowchart for this program.
e. Write the pseudocode for this program.

13. A supervisor in the same manufacturing company as described in Exercise 12 wants to produce a report from the PRODUCTION input file showing bonuses she is planning to give based on this year's production. She wants to have a report with three columns: last name, first name, and bonus. The bonuses will be distributed as follows.

If this year's production is:

- 1,000 units or fewer, the bonus is $25
- 1,001 to 3,000 units, the bonus is $50
- 3,001 to 6,000 units, the bonus is $100
- 6,001 units and up, the bonus is $200

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Create a decision table to use while planning the logic for this program.
d. Draw the flowchart for this program.
e. Write the pseudocode for this program.

14. Modify Exercise 13 to reflect the following new facts, and have the program execute as efficiently as possible:

- Only employees whose production this year is higher than it was last year will receive bonuses. This is true for approximately 30 percent of the employees.
- Sixty percent of employees produce over 6,000 units per year; 20 percent produce 3,001 to 6,000; 15 percent produce 1,001 to 3,000 units; and only 5 percent produce fewer than 1,001.

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Create a decision table to use while planning the logic for this program.
d. Draw the flowchart for this program.
e. Write the pseudocode for this program.

15. The Richmond Riding Club wants to assign the title of Master or Novice to each of its members. A member earns the title of Master by accomplishing two or more of the following:

- Participating in at least eight horse shows
- Winning a first-place or second-place ribbon in at least two horse shows, no matter how many shows the member has participated in
- Winning a first-place, second-place, third-place, or fourth-place ribbon in at least four horse shows, no matter how many shows the member has participated in

Create a report that prints each club member's name along with the designation "Master" or "Novice". Input exists as follows:

```
RIDING FILE DESCRIPTION
File name: RIDING
FIELD DESCRIPTION        DATA TYPE     COMMENTS
Last Name                Character     15 characters
First Name               Character     15 characters
Number of Shows          Numeric       0 decimals
First-Place Ribbons      Numeric       0 decimals
Second-Place Ribbons     Numeric       0 decimals
Third-Place Ribbons      Numeric       0 decimals
Fourth-Place Ribbons     Numeric       0 decimals
```

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Create a decision table to use while planning the logic for this program.
d. Draw the flowchart for this program.
e. Write the pseudocode for this program.

16. **Freeport Financial Services manages clients' investment portfolios. The company charges for its services based on each client's annual income, net worth, and length of time as a client, as follows:**

- Clients with an annual income over $100,000 and a net worth over $1 million are charged 1.5 percent of their net worth.
- Clients with an annual income over $100,000 and a net worth between $500,000 and $1 million inclusive are charged $8,000.
- Clients with an annual income over $100,000 and a net worth of less than $500,000 are charged $6,000.
- Clients with an annual income from $75,000 up to and including $100,000 are charged 1 percent of their net worth.
- Clients with an income of $75,000 or less are charged $4,000, unless their net worth is over $1 million, in which case they are charged $4,500.
- Any client for over four years gets a 10 percent discount; any client for over seven years gets a 15 percent discount.

Create a report that prints each client's name and the client's annual fee. Input records contain the following data:

```
FINANCIAL SERVICE CLIENTS' FILE DESCRIPTION
File name: CLIENTS
FIELD DESCRIPTION        DATA TYPE     COMMENTS
Last Name                Character     15 characters
First Name               Character     15 characters
Annual Income            Numeric       0 decimals
Portfolio Value          Numeric       0 decimals
Years as Client          Numeric       0 decimals
```

a. Design the output for this program; create either sample output or a print chart.
b. Draw the hierarchy chart for this program.
c. Create a decision table to use while planning the logic for this program.
d. Draw the flowchart for this program.
e. Write the pseudocode for this program.

DETECTIVE WORK

1. **Computers are expert chess players because they can make many good decisions very rapidly. Explore the history of computer chess playing.**

2. **George Boole is considered the father of symbolic logic. Find out about his life.**

UP FOR DISCUSSION

1. **Computer programs can be used to make decisions about your insurability as well as the rates you will be charged for health and life insurance policies. For example, certain preexisting conditions may raise your insurance premiums considerably. Is it ethical for insurance companies to access your health records and then make insurance decisions about you?**

2. **Job applications are sometimes screened by software that makes decisions about a candidate's suitability based on keywords in the applications. Is such screening fair to applicants?**

3. **Medical facilities often have more patients waiting for organ transplants than there are available organs. Suppose you have been asked to write a computer program that selects which of several candidates should receive an available organ. What data would you want on file to be able to use in your program, and what decisions would you make based on the data? What data do you think others might use that you would choose not to use?**