Chapter 4

Making Decisions

Chapter Introduction

In this chapter, you will learn about:

- Boolean expressions and the selection structure
- The relational comparison operators
- AND logic
- OR logic
- Making selections within ranges
- Precedence when combining AND and OR operators

4-1 Boolean Expressions and the Selection Structure

The reason people frequently think computers are smart lies in the ability of computer programs to make decisions. A medical diagnosis program that can decide if your symptoms fit various disease profiles seems quite intelligent, as does a program that can offer different potential vacation routes based on your destination.

Every decision you make in a computer program involves evaluating a **Boolean expression** (An expression that represents only one of two states, usually expressed as true or false.) —an expression whose value can be only true or false. 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. A Boolean expression is used in every selection structure. The selection structure is not new to you—it's one of the basic structures you learned about in Chapter 3. See Figures 4-1 and 4-2.

Figure 4-1

The Dual-Alternative Selection Structure

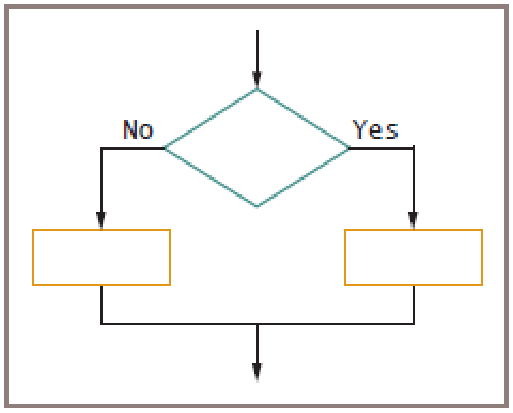
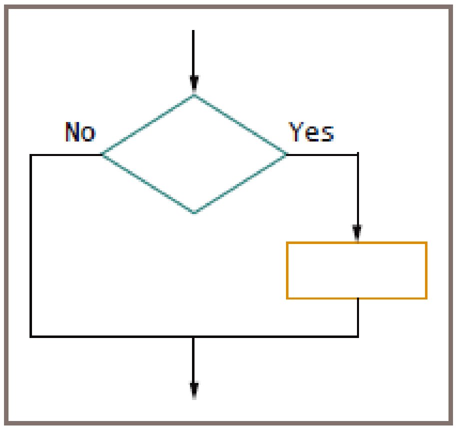


Figure 4-2

The Single-Alternative Selection Structure



Mathematician George Boole (1815–1864) approached logic more simply than his predecessors did, by expressing logical selections with common algebraic symbols. He is considered the founder of mathematical logic, and Boolean (true/false) expressions are named for him.

In Chapter 3 you learned that the structure in Figure 4-1 is a dual-alternative, or binary, selection because an action is 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 to only one of the two alternatives, never to both.

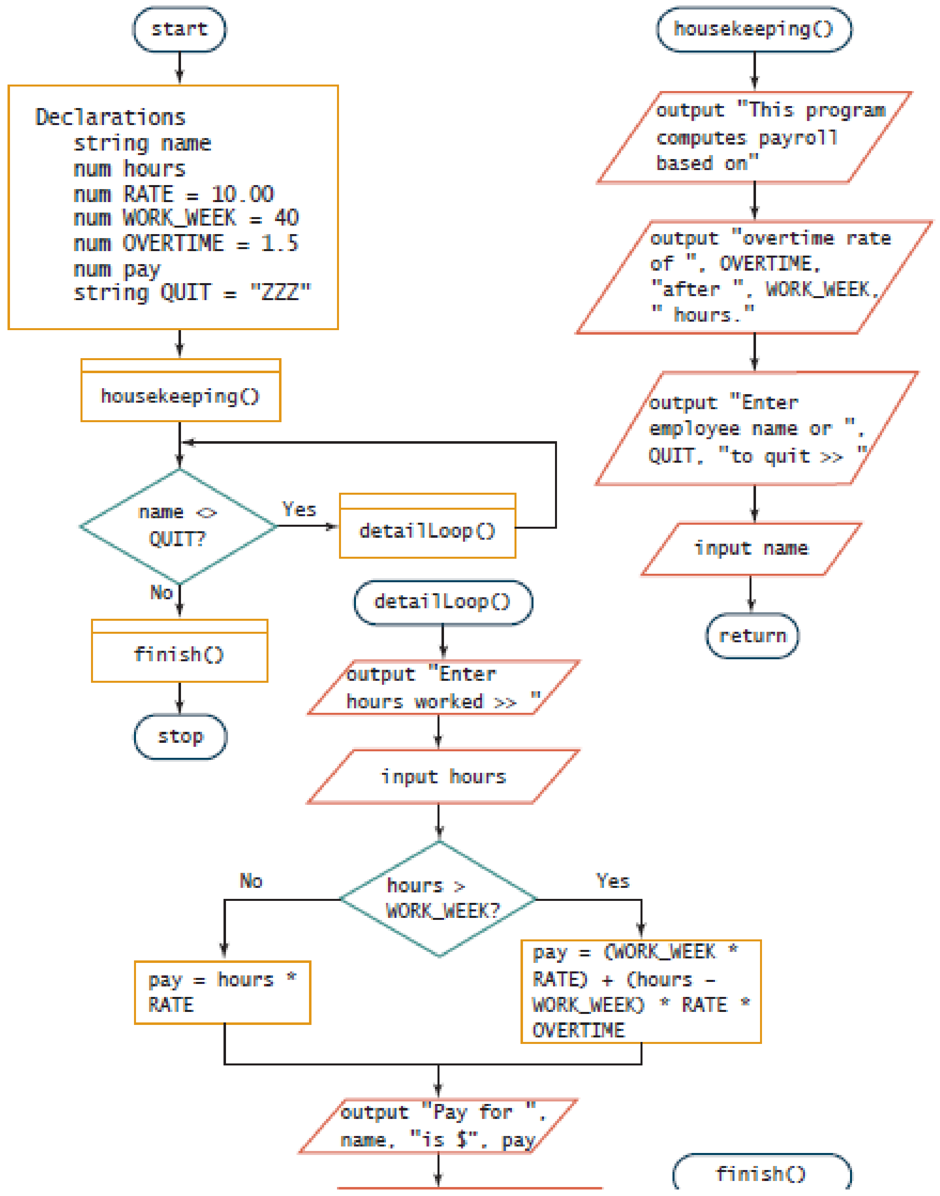
This book follows the convention that the two logical paths emerging from a decision are drawn to the right and left of a diamond in a flowchart. Some programmers draw one of the flowlines emerging from the bottom of the diamond. The exact format of the diagram is not as important as the idea that one logical path flows into a selection, and two possible outcomes emerge.

The flowchart segment in Figure 4-2 represents a single-alternative selection in which action is required for only one outcome of the question. This form of the selection structure is called an **if-then** (A structure similar to an if-then-else, but no alternative or “else” action is necessary.) , because no alternative or `else` action is necessary.

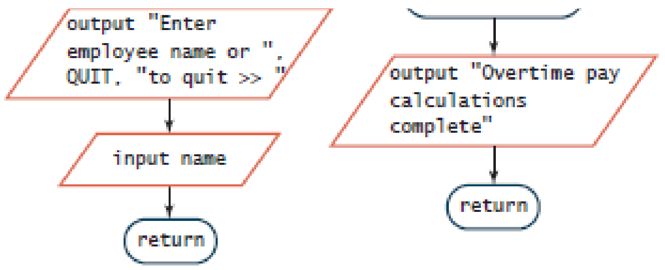
Figure 4-3 shows the flowchart and pseudocode for an interactive program that computes pay for employees. The program displays the weekly pay for each employee at the same hourly rate ($10.00) and assumes that there are no payroll deductions. The mainline logic calls `housekeeping()`, `detailLoop()`, and `finish()` modules. The `detailLoop()` module contains a typical `if-then-else` decision that determines whether an employee has worked more than a standard workweek (40 hours), and pays one and one-half times the employee’s usual hourly rate for hours worked in excess of 40 per week.

Figure 4-3

Flowchart and Pseudocode for Overtime Payroll Program



start
Declarations
string name
num hours
num RATE = 10.00
num WORK_WEEK = 40
num OVERTIME = 1.5
num pay
string QUIT = "ZZZ"
housekeeping()
name <> QUIT?
Yes
detailLoop()
No
finish()
stop
detailLoop()
output "Enter hours worked >> "
input hours
No
hours > WORK_WEEK?
Yes
pay = hours * RATE
pay = (WORK_WEEK * RATE) + (hours – WORK_WEEK) * RATE * OVERTIME
output "Pay for ", name, "is $", pay
housekeeping()
output "This program computes payroll based on"
output "overtime rate of ", OVERTIME, "after ", WORK_WEEK, " hours."
output "Enter employee name or ", QUIT, "to quit >> "
input name
return
finish()


output "Enter employee name or ", QUIT, "to quit >> "
input name
return
output "Overtime pay calculations complete"
return

```
start
   Declarations
      string name
      num hours
      num RATE = 10.00
      num WORK_WEEK = 40
      num OVERTIME = 1.5
      num pay
      string QUIT = "ZZZ"
   housekeeping()
   while name <> QUIT
      detailLoop()
   endwhile
   finish()
stop

housekeeping()
   output "This program computes payroll based on"
   output "overtime rate of ", OVERTIME, "after ", WORK_WEEK, " hours."
   output "Enter employee name or ", QUIT, "to quit >> "
   input name
return

detailLoop()
   output "Enter hours worked >> "
   input hours
   if hours > WORK_WEEK then
      pay = (WORK_WEEK * RATE) + (hours - WORK_WEEK) * RATE * OVERTIME
   else
      pay = hours * RATE
   endif
   output "Pay for ", name, "is $", pay
   output "Enter employee name or ", QUIT, "to quit >> "
   input name
return

finish()
   output "Overtime pay calculations complete"
return
```

Throughout this book, many examples are presented in both flowchart and pseudocode form. When you analyze a solution, you might find it easier to concentrate on just one of the two design tools at first. When you understand how the program works using one tool (for example, the flowchart), you can confirm that the solution is identical using the other tool.

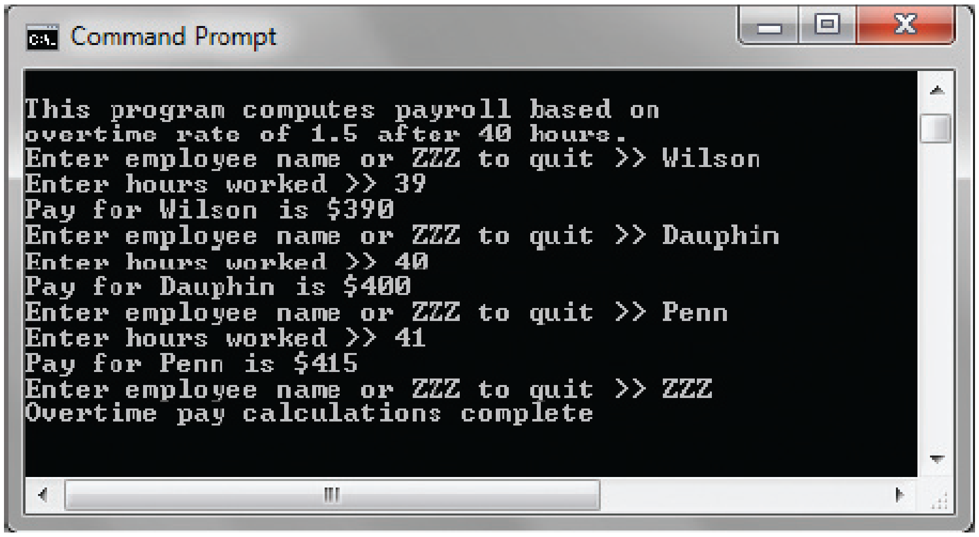
In the `detailLoop()` module of the program in Figure 4-3, the decision contains two clauses:

- The **if-then clause** (The part of a decision that holds the resulting action when the Boolean expression in the decision is true.) is the part of the decision that holds the action or actions that execute when the tested condition in the decision is true. In this example, the clause holds the longer overtime calculation.
- The **else clause** (A part of a decision that holds the action or actions that execute only when the Boolean expression in the decision is false.) of the decision is the part that executes only when the tested condition in the decision is false. In this example, the clause contains the shorter calculation.

Figure 4-4 shows a typical execution of the program in a command-line environment. Data values are entered for three employees. The first two employees do not work more than 40 hours, so their pay is displayed simply as hours times $10.00. The third employee, however, has worked one hour of overtime, and so makes $15 for the last hour instead of $10.

Figure 4-4

Typical Execution of the Overtime Payroll Program in Figure 4-3



```
This program computes payroll based on
overtime rate of 1.5 after 40 hours.
Enter employee name or ZZZ to quit >> Wilson
Enter hours worked >> 39
Pay for Wilson is $390
Enter employee name or ZZZ to quit >> Dauphin
Enter hours worked >> 40
Pay for Dauphin is $400
Enter employee name or ZZZ to quit >> Penn
Enter hours worked >> 41
Pay for Penn is $415
Enter employee name or ZZZ to quit >> ZZZ
Overtime pay calculations complete
```

Watch the video *Boolean Expressions and Decisions*.

Two Truths & A Lie

Boolean Expressions and the Selection Structure

1. The `if-then` clause is the part of a decision that executes when a tested condition in a decision is true.

   T F

2. The `else` clause is the part of a decision that executes when a tested condition in a decision is true.

   T F

3. A Boolean expression is one whose value is true or false.

   T F

4-2 Using Relational Comparison Operators

Table 4-1 describes the six **relational comparison operators** (A symbol that expresses Boolean comparisons. Examples include =, >, <, >=, <=, and <>.) supported by all modern programming languages. Each of these operators is binary—that is, each requires two operands. When you construct an expression using one of the operators described in Table 4-1, the expression evaluates to true or false. (Notice that some operators are formed using two characters without a space between them.) Usually, both operands in a comparison must be the same data type; that is, you can compare numeric values to other numeric values, and text strings to other strings. Some programming languages allow you to compare a character to a number. If you do, then a single character's numeric code value is used in the comparison. Appendix A contains more information on coding systems. In this book, only operands of the same type will be compared.

Table 4-1

Relational Comparison Operators

Operator	Name	Discussion
=	Equivalency operator	Evaluates as true when its operands are equivalent. Many languages use

		a double equal sign (==) to avoid confusion with the assignment operator.
>	Greater-than operator	Evaluates as true when the left operand is greater than the right operand.
<	Less-than operator	Evaluates as true when the left operand is less than the right operand.
>=	Greater-than or equal-to operator	Evaluates as true when the left operand is greater than or equivalent to the right operand.
<=	Less-than or equal-to operator	Evaluates as true when the left operand is less than or equivalent to the right operand.
<>	Not-equal-to operator	Evaluates as true when its operands are not equivalent. Some languages use an exclamation point followed by an equal sign to indicate not equal to (!=).

In any Boolean expression, the two values compared can be either variables or constants. For example, the expression `currentTotal = 100?` compares 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, you would never use expressions in which you compare two constants—for example, `20 = 20?` or `30 = 40?`. Such expressions are **trivial expressions** (A trivial expression is one that always evaluates to the same value.) because each will always evaluate to the same result: true for `20 = 20?` and false for `30 = 40?`.

Some languages require special operations to compare strings, but this book will assume that the standard comparison operators work correctly with strings based on their alphabetic values. For example, the comparison `"black" < "blue"?` would be evaluated as true because `"black"` precedes `"blue"` alphabetically. Usually, string 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".

Any decision can be made using combinations of just three types of comparisons: equal, greater than, and less than. You never need the three additional comparisons (greater than or equal, less than or equal, or not equal), but using them often makes decisions more convenient. For example, assume that 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 the >= operator did not exist, you could express the same logic by writing:

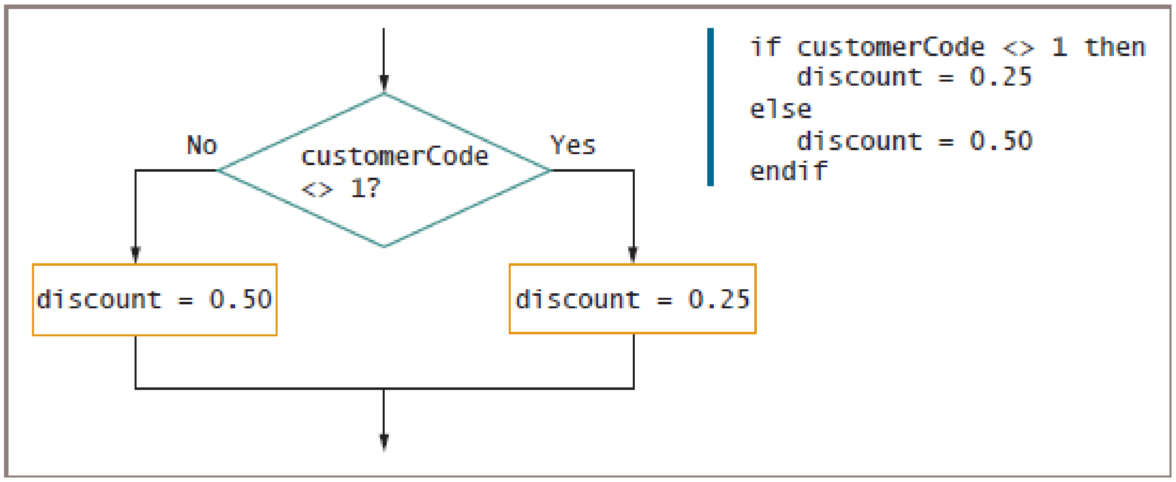
```
if customerAge < 65 then
   discount = 0
else
   discount = 0.10
endif
```

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 action that was your motivation for making the test. 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 can make you prone to including logical errors in your programs. For example, consider the flowchart segment in Figure 4-5.

Figure 4-5

Using a Negative Comparison



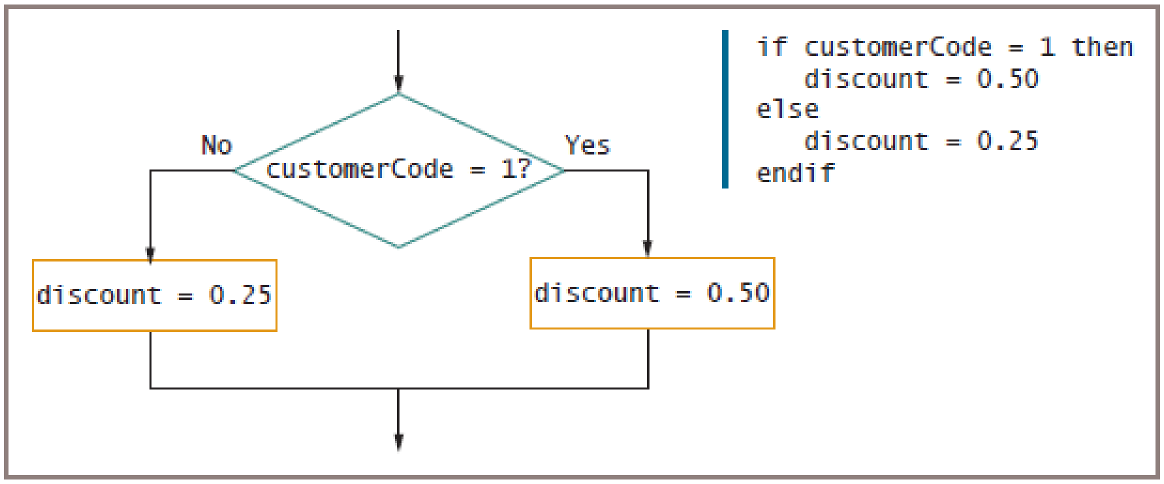
```
if customerCode <> 1 then
   discount = 0.25
else
   discount = 0.50
endif
```

In Figure 4-5, if the value of `customerCode` *is* equal to 1, the logical flow follows the false branch of the selection. If `customerCode <> 1` is true, the `discount` is 0.25; if `customerCode <> 1` is not true, it means the `customerCode` *is* 1, and the `discount` is 0.50. Even reading the phrase "if `customerCode` is not equal to 1 is not true" is awkward.

Figure 4-6 shows the same decision, this time asked using positive logic. Making the decision based on what `customerCode` *is* is clearer than trying to determine what `customerCode` is *not*.

Figure 4-6

Using the Positive Equivalent of the Negative Comparison in Figure 4-5



```
if customerCode = 1 then
   discount = 0.50
else
   discount = 0.25
endif
```

Although negative comparisons can be awkward to use, your meaning is sometimes clearest when using them. Frequently, this occurs when you use an `if` without an `else`, taking action only when some comparison is false. An example would be: `if customerZipCode <> LOCAL_ZIP_CODE then total = total + deliveryCharge`.

4-2a Avoiding a Common Error with Relational Operators

A common error that occurs when programming with relational operators is using the wrong one and missing the boundary or limit required for a selection. If you use the > symbol to make a selection when you should have used >=, all the cases that are equal will go unselected. Unfortunately, people who request programs do not always speak as precisely as a computer. If, for example, your boss says, "Write a program that selects all employees over 65," does she mean to include employees who are 65 or not? In other words, is the comparison `age > 65` or `age >= 65`? Although the phrase *over 65* indicates *greater than 65*, people do not always say what they mean, and the best course of action is to double-check the intended meaning with the person who requested the program—for example, the end user, your supervisor, or your instructor. Similar phrases that can cause misunderstandings are *no more than*, *at least*, and *not under*.

Two Truths & A Lie

Using Relational Comparison Operators

1. Usually, you can compare only values that are of the same data type.

   T F

2. A Boolean expression is defined as one that decides whether two values are equal.

   T F

3. In any Boolean expression, the two values compared can be either variables or constants.

   T F

4-3 Understanding *AND* Logic

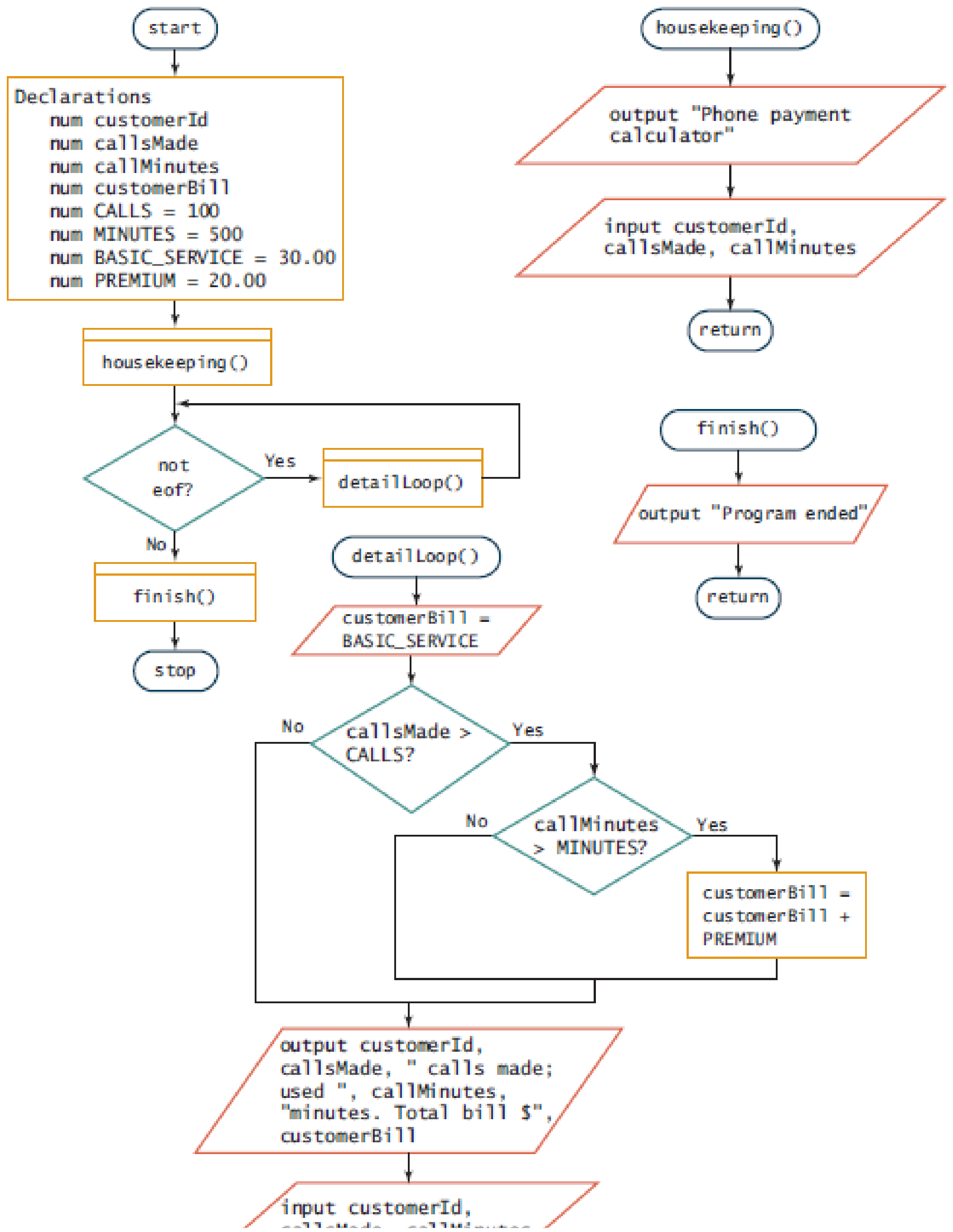
Often, you need to evaluate more than one expression to determine whether an action should take place. When you ask multiple questions before an outcome is determined, you create a **compound condition** (A condition constructed when multiple decisions are required before determining an outcome.) . For example, suppose you work for a cell phone company that charges customers as follows:

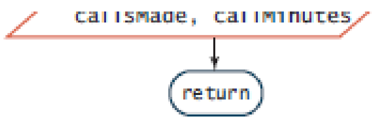
- The basic monthly service bill is $30.
- An additional $20 is billed to customers who make more than 100 calls that last for a total of more than 500 minutes.

The logic needed for this billing program includes an **AND decision** (A decision in which two conditions must both be true for an action to take place.) —a decision in which two conditions must be true for an action to take place. In this case, both a minimum number of calls must be made *and* a minimum number of minutes must be used before the customer is charged the additional amount. An AND decision can be constructed using a **nested decision** (A decision within the `if-then` or `else` clause of another decision; also called a *nested if*.) , or a **nested if** (A decision within the `if-then` or `else` clause of another decision; also called a *nested if*.) —that is, a decision within the `if`-then or `else` clause of another decision. A series of nested `if` statements is also called a **cascading if statement** (A series of nested if statements.) . The flowchart and pseudocode for the program that determines the charges for customers is shown in Figure 4-7.

Figure 4-7

Flowchart and Pseudocode for Cell Phone Billing Program





```
start
   Declarations
      num customerId
      num callsMade
      num callMinutes
      num customerBill
      num CALLS = 100
      num MINUTES = 500
      num BASIC_SERVICE = 30.00
      num PREMIUM = 20.00
   housekeeping()
   while not eof
      detailLoop()
   endwhile
   finish()
stop

housekeeping()
   output "Phone payment calculator"
   input customerId, callsMade, callMinutes
return

detailLoop()
   customerBill = BASIC_SERVICE
   if callsMade > CALLS then
      if callMinutes > MINUTES then
         customerBill = customerBill + PREMIUM
      endif
   endif
   output customerId, callsMade, " calls made; used ",
      callMinutes, " minutes. Total bill $", customerBill
   input customerId, callsMade, callMinutes
return

finish()
   output "Program ended"
return
```

You first learned about nesting structures in Chapter 3. You can always stack and nest any of the basic structures.

In the cell phone billing program, the customer data is retrieved from a file. This eliminates the need for prompts and keeps the program shorter so you can concentrate on the decision-making process. If this was an interactive program, you would use a prompt before each input statement. Chapter 7 covers file processing and explains a few additional steps you can take when working with files.

In Figure 4-7, the appropriate variables and constants are declared, and then the `housekeeping()` module displays an introductory heading and gets the first set of input data. After control returns to the mainline logic, the `eof` condition is tested, and if data entry is not complete, the `detailLoop()` module executes. In the `detailLoop()` module, the customer's bill is set to the standard fee, and then the nested decision executes. In the nested `if` structure in Figure 4-7, the expression `callsMade > CALLS?` is evaluated first. If this expression is true, only then is the second Boolean expression `(callMinutes > MINUTES?)` evaluated. If that expression is also true, then the $20 premium is added to the customer's bill. If either of the tested conditions is false, the customer's bill value is never altered, retaining the initially assigned value of $30.

Most languages allow you to use a variation of the decision structure called the *`case structure`* when you must nest a series of decisions about a single variable. Appendix F contains information about the case structure.

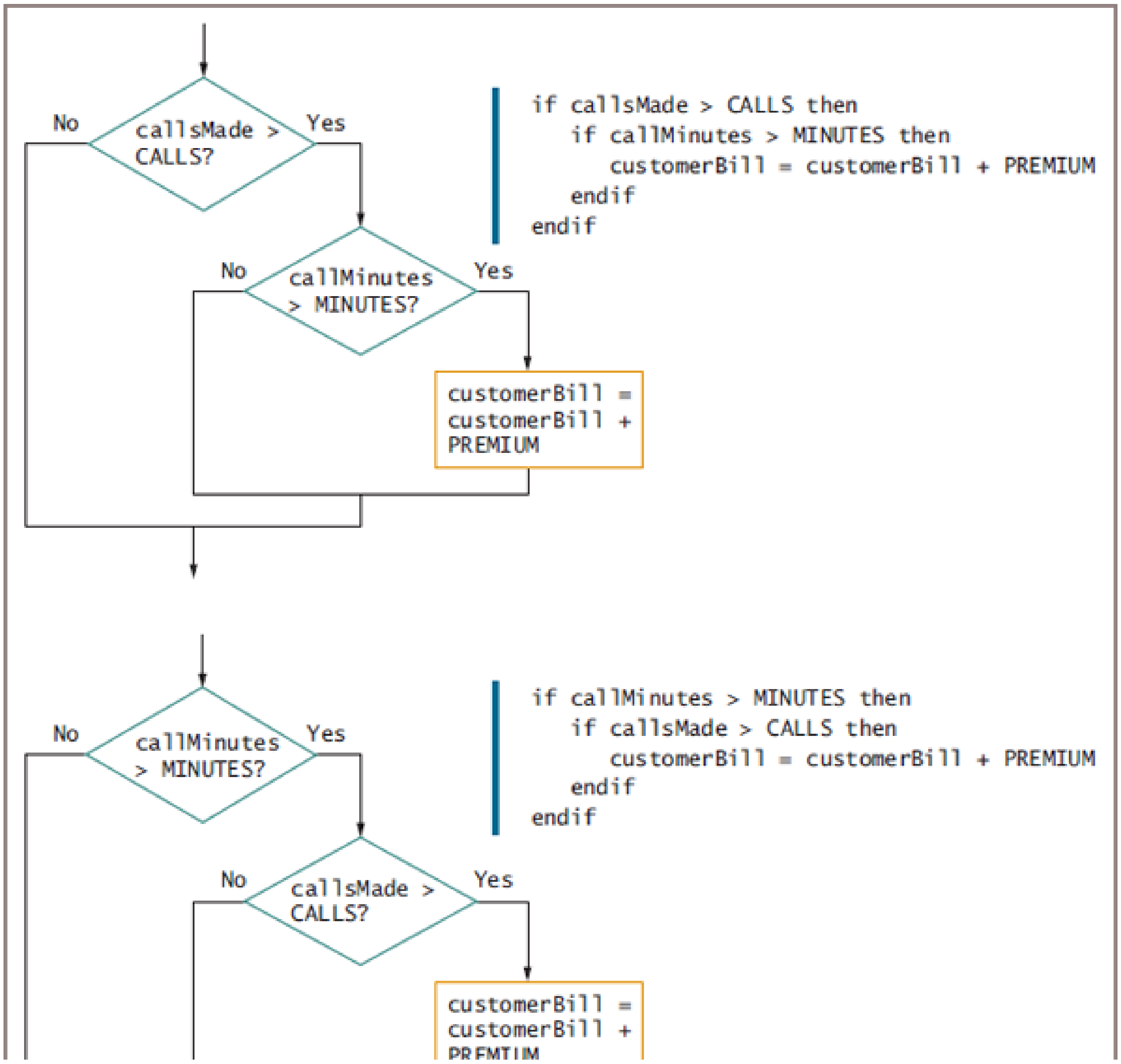
4-3a Nesting *AND* Decisions for Efficiency

When you nest two decisions, you must choose which of the decisions to make first. Logically, either expression in an AND decision can be evaluated first. However, you often can improve your program's performance by correctly choosing which of two selections to make first.

For example, Figure 4-8 shows two ways to design the nested decision structure that assigns a premium to customers' bills if they make more than 100 cell phone calls and use more than 500 minutes in a billing period. The program can ask about calls made first, eliminate customers who have not made more than the minimum, and ask about the minutes used only for customers who pass (that is, are evaluated as true on) the minimum calls test. Or, the program could ask about the minutes first, eliminate those who do not qualify, and ask about the number of calls only for customers who pass the minutes test. Either way, only customers who exceed both limits must pay the premium. Does it make a difference which question is asked first? As far as the result goes, no. Either way, the same customers pay the premium—those who qualify on the basis of both criteria. As far as program efficiency goes, however, it *might* make a difference which question is asked first.

Figure 4-8

Two Ways to Produce Cell Phone Bills Using Identical Criteria



```
if callsMade > CALLS then
   if callMinutes > MINUTES then
      customerBill = customerBill + PREMIUM
   endif
endif
```

```
if callMinutes > MINUTES then
   if callsMade > CALLS then
      customerBill = customerBill + PREMIUM
   endif
endif
```

Assume that you know that out of 1000 cell phone customers, about 90 percent, or 900, make more than 100 calls in a billing period. Assume that you also know that only about half the 1000 customers, or 500, use more than 500 minutes of call time.

If you use the logic shown first in Figure 4-8, and you need to produce 1000 phone bills, the first question, `callsMade > CALLS?`, will execute 1000 times. For approximately 90 percent of the customers, or 900 of them, the answer is true, so 100 customers are eliminated from the premium assignment, and 900 proceed to the next question about the minutes used. Only about half the customers use more than 500 minutes, so 450 of the 900 pay the premium, and it takes 1900 questions to identify them.

Using the alternate logic shown second in Figure 4-8, the first question, `callMinutes > MINUTES?`, will also be asked 1000 times—once for each customer. Because only about half the customers use the high number of minutes, only 500 will pass this test and proceed to the question for number of calls made. Then, about 90 percent of the 500, or 450 customers, will pass the second test and be billed the premium amount. It takes 1500 questions to identify the 450 premium-paying customers.

Whether you use the first or second decision order in Figure 4-8, the same 450 customers who satisfy both criteria pay the premium. The difference is that when you ask about the number of calls first, the program must ask 400 more questions than when you ask about the minutes used first.

The 400-question difference between the first and second set of decisions doesn't take much time on most computers. But it does take *some* time, and if a corporation has hundreds of thousands of customers instead of only 1000, or if many such decisions have to be made within a program, performance time can be significantly improved by asking questions in the more efficient order.

Often when you must make nested decisions, you have no idea which event is more likely to occur; in that case, you can legitimately ask either question first. 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 instances of the second decision as possible, which speeds up processing time.

Watch the video *Writing Efficient Nested Selections*.

4-3b Using the AND Operator

Most programming languages allow you to ask two or more questions in a single comparison by using a **conditional AND operator** (A symbol used to combine decisions so that two or more conditions must be true for an action to occur. Also called an *AND operator*.) , or more simply, an **AND operator** (A symbol used to combine decisions so that two or more conditions must be true for an action to occur. Also called an *AND operator*.) that joins decisions in a single statement. For example, if you want to bill an extra amount to cell phone customers who make more than 100 calls that total more than 500 minutes in a billing period, you can use nested decisions, as shown in the previous section, or you can include both decisions in a single statement by writing the following question:

```
callsMade > CALLS AND callMinutes > MINUTES?
```

When you use one or more AND operators to combine two or more Boolean expressions, each Boolean expression must be true for the entire expression to be evaluated as true. For example, if you ask, "Are you a native-born U.S. citizen and are you at least 35 years old?", the answer to both parts of the question must be *yes* before the response can be a single, summarizing *yes*. If either part of the expression is false, then the entire expression is false.

The conditional AND operator in Java, C++, and C# consists of two ampersands, with no spaces between them (&&). In Visual Basic, you use the word And.

One tool that can help you understand the AND operator is a truth table. **Truth tables** (Truth tables are diagrams used in mathematics and logic to help describe the truth of an entire expression based on the truth of its parts.) are diagrams used in mathematics and logic to help describe the truth of an entire expression based on the truth of its parts. Table 4-2 shows a truth table that lists all the possibilities with an AND decision. As the table shows, for any two expressions x and y, the expression x AND y? is true only if both x and y are individually true. If either x or y alone is false, or if both are false, then the expression x AND y? is false.

Table 4-2

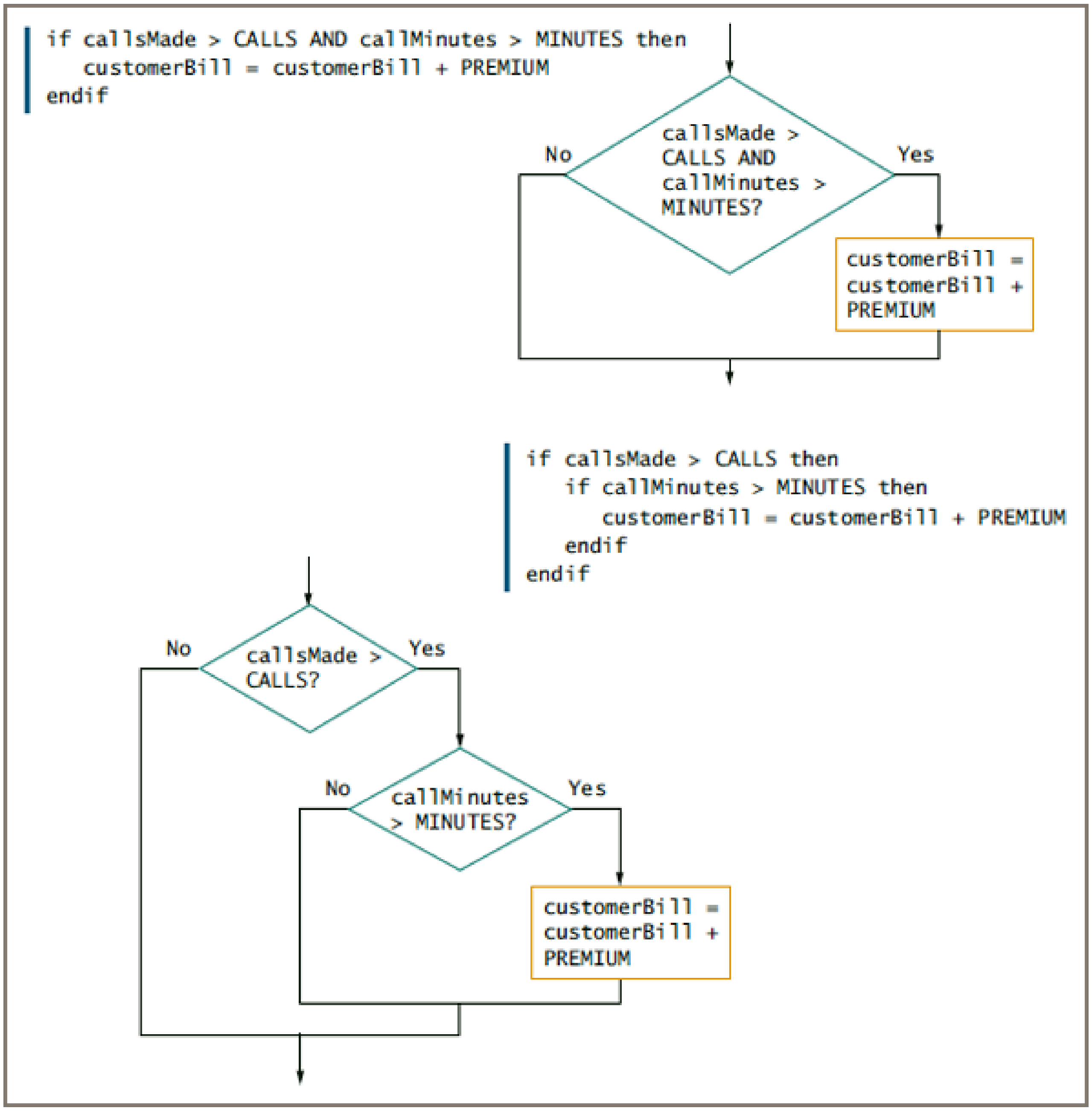
Truth Table for the AND Operator

x?	y?	x AND y?
True	True	True
True	False	False
False	True	False
False	False	False

If the programming language you use allows an AND operator, you must realize that the question you place first (to the left of the operator) is the one that will be asked first, and cases that are eliminated based on the first question will not proceed to the second question. In other words, each part of an expression that uses an AND operator is evaluated only as far as necessary to determine whether the entire expression is true or false. This feature is called **short-circuit evaluation** (A logical feature in which each part of a larger expression is evaluated only as far as necessary to determine the final outcome.) . The computer can ask only one question at a time; even when your pseudocode looks like the first example in Figure 4-9, the computer will execute the logic shown in the second example. 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 is not even tested.

Figure 4-9

Using an AND Operator and the Logic Behind It



```
if callsMade > CALLS AND callMinutes > MINUTES then
   customerBill = customerBill + PREMIUM
endif
```

```
if callsMade > CALLS then
   if callMinutes > MINUTES then
      customerBill = customerBill + PREMIUM
   endif
endif
```

No
callsMade > CALLS AND callMinutes > MINUTES?
Yes
customerBill = customerBill + PREMIUM
No
callsMade > CALLS?
Yes
No
callMinutes > MINUTES?
Yes
customerBill = customerBill + PREMIUM

You are never required to use the `AND` operator because using nested `if` statements can always achieve the same result. However, using the `AND` operator often makes your code more concise, less error-prone, and easier to understand.

4-3c Avoiding Common Errors in an *AND* Selection

When you need to 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 add a $20 premium to the bill of cell phone customers who exceed 100 calls and 500 minutes in a billing period, then the program segment shown in Figure 4-10 contains three different types of logic errors.

Figure 4-10

Incorrect Logic to Add a $20 Premium to the Bills of Cell Phone Customers Who Meet Two Criteria

The logic in Figure 4-10 shows that $20 is added to the bill of a customer who makes too many calls. This customer should not necessarily be billed extra—the customer's minutes might be below the cutoff for the $20 premium. In addition, a customer who has made few calls is not eliminated from the second question. Instead, all customers are subjected to the minutes question, and some are assigned the premium even though they might not have passed the criterion for number of calls made. Additionally, any customer who passes both tests has the premium added to his bill twice. For many reasons, the logic shown in Figure 4-10 is *not* correct for this problem.

When you use the `AND` operator in most languages, you must provide a complete Boolean expression on each side of the operator. In other words, `callMinutes > 100 AND callMinutes < 200` would be a valid expression to find `callMinutes` between 100 and 200. However, `callMinutes > 100 AND < 200` would not be valid because what follows the `AND` operator (< 200) is not a complete Boolean expression.

For clarity, you can surround each Boolean expression in a compound expression with its own set of parentheses. Use this format if it is clearer to you. For example, you might write the following:

```
if (callMinutes > MINUTES) AND (callsMade > CALLS)
   customerBill = customerBill + PREMIUM
endif
```

Two Truths & A Lie

Understanding *AND* Logic

1. When you nest decisions because the resulting action requires that two conditions be true, either decision logically can be made first and the same selections will occur.

   T F

2. When two selections are required for an action to take place, you often can improve your program's performance by appropriately choosing which selection to make first.

   T F

3. To improve efficiency in a nested selection in which two conditions must be true for some action to occur, you should first ask the question that is more likely to be true.

   T F

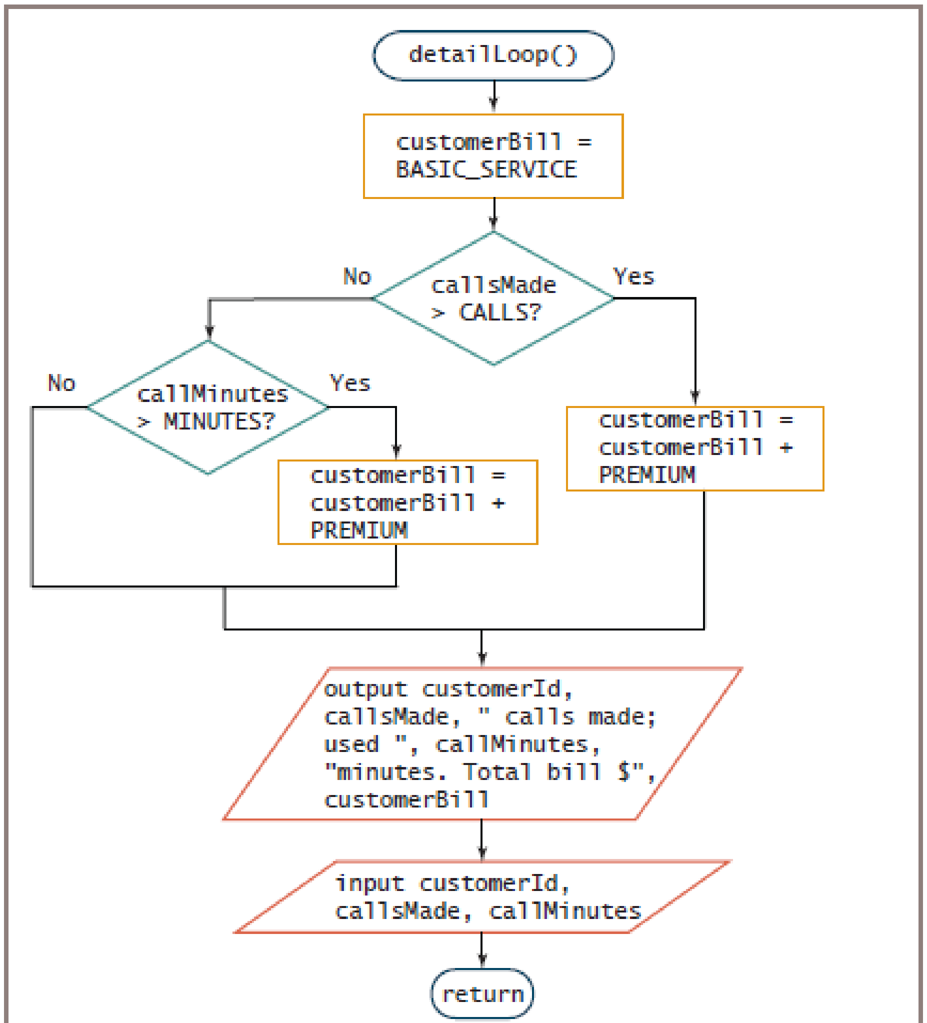
4-4 Understanding *OR* Logic

Sometimes you want to take action when one *or* the other of two conditions is true. This is called an **OR decision** (A decision that contains two or more conditions; if at least one condition is met, the resulting action takes place.) because either one condition *or* some other condition must be met in order for an event to take place. If someone asks, "Are you free for dinner Friday or Saturday?," only one of the two conditions has to be true 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.

For example, suppose you want to add $20 to the bills of cell phone customers who either make more than 100 calls or use more than 500 minutes. Figure 4-11 shows the altered `detailLoop()` module of the billing program that accomplishes this objective.

Figure 4-11

Flowchart and Pseudocode for Cell Phone Billing Program in Which a Customer Must Meet One or Both of Two Criteria to Be Billed a Premium


detailLoop()
customerBill = BASIC_SERVICE
No
callsMade > CALLS?
Yes
No
callMinutes > MINUTES?
Yes
customerBill = customerBill + PREMIUM
customerBill = customerBill + PREMIUM
output customerId, callsMade, " calls made; used ", callMinutes, "minutes. Total bill $", customerBill
input customerId, callsMade, callMinutes
return

```
detailLoop()
   customerBill = BASIC_SERVICE
   if callsMade > CALLS then
      customerBill = customerBill + PREMIUM
   else
      if callMinutes > MINUTES then
         customerBill = customerBill + PREMIUM
      endif
   endif
   output customerId, callsMade, " calls made; used ",
      callMinutes, " minutes. Total bill $", customerBill
   input customerId, callsMade, callMinutes
return
```

The `detailLoop()` in the program in Figure 4-11 asks the question `callsMade > CALLS?`, and if the result is true, the extra amount is added to the customer's bill. Because making many calls is enough for the customer to incur the premium, there is no need for further questioning. If the customer has not made more than 100 calls, only then does the program need to ask whether `callMinutes > MINUTES` is true. If the customer did not make over 100 calls, but used more than 500 minutes nevertheless, then the premium amount is added to the customer's bill.

4-4a Writing *OR* Decisions for Efficiency

As with an AND selection, when you use an OR selection, you can choose to ask either question first. For example, you can add an extra $20 to the bills of customers who meet one or the other of two criteria using the logic in either part of Figure 4-12.

Figure 4-12

Two Ways to Assign a Premium to Bills of Customers Who Meet One of Two Criteria

You might have guessed that one of these selections is superior to the other when you have some background information about the relative likelihood of each condition being tested. For example, let’s say you know that out of 1000 cell phone customers, about 90 percent, or 900, make more than 100 calls in a billing period. You also know that only about half the 1000 customers, or 500, use more than 500 minutes of call time.

When you use the logic shown in the first half of Figure 4-12, you first ask about the calls made. For 900 customers the answer is true, and you add the premium to their bills. Only about 100 sets of customer data continue to the next question regarding the call minutes, where about 50 percent of the 100, or 50, are billed the extra amount. In the end, you have made 1100 decisions to correctly add premium amounts for 950 customers.

If you use the OR logic in the second half of Figure 4-12, you ask about minutes used first—1000 times, once each for 1000 customers. The result is true for 50 percent, or 500 customers, whose bill is increased. For the other 500 customers, you ask about the number of calls made. For 90 percent of the 500, the result is true, so premiums are added for 450 additional people. In the end, the same 950 customers are billed an extra $20—but this approach requires executing 1500 decisions, 400 more decisions than when using the first decision logic.

The general rule is: *In an OR decision, first ask the question that is more likely to be true*. This approach eliminates as many executions of the second decision as possible, and the time it takes to process all the data is decreased. As with the AND situation, in an OR situation, it is more efficient to eliminate as many extra decisions as possible.

4-4b Using the OR Operator

If 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 **conditional OR operator** (A symbol used to combine decisions when any one condition can be true for an action to occur. Also called an *OR operator*.) (or simply the **OR operator** (A symbol used to combine decisions when any one condition can be true for an action to occur. Also called an *OR operator*.)). For example, you can ask the following question:

As with the AND operator, most programming languages require a complete Boolean expression on each side of the OR operator. When you use the logical OR operator, only one of the listed conditions must be met for the resulting action to take place. Table 4-3 shows the truth table for the OR operator. As you can see in the table, the entire expression x OR y? is false only when x and y each are false individually.

Table 4-3

Truth Table for the OR Operator

X?	Y?	x OR y?
True	True	True
True	False	True

False	True	True
False	False	False

If the programming language you use supports the OR operator, you still must realize that the question you place first is the question that will be asked first, and cases that pass the test of the first question will not proceed to the second question. As with the AND operator, this feature is called short-circuiting. The computer can ask only one question at a time; even when you write code as shown at the top of Figure 4-13, the computer will execute the logic shown at the bottom.

Figure 4-13

Using an OR Operator and the Logic Behind It

C#, C++, C, and Java use the symbol || as the logical `OR` operator. In Visual Basic, the operator is `Or`.

4-4c Avoiding Common Errors in an *OR* Selection

You might have noticed that the assignment statement `customerBill = customerBill + PREMIUM` appears twice in the decision-making processes in Figures 4-12 and 4-13. When you create a flowchart, the temptation is to draw the logic to look like Figure 4-14. Logically, you might argue that the flowchart in Figure 4-14 is correct because the correct customers are billed the extra $20. However, this flowchart is not structured. The second question is not a self-contained structure with one entry and exit point; instead, the flowline breaks out of the inner selection structure to join the `Yes` side of the outer selection structure.

Figure 4-14

Unstructured Flowchart for Determining Customer Cell Phone Bill

The OR selection has additional potential for errors due to the differences in the way people and computers use language. When your boss wants to add an extra amount to the bills of customers who make more than 100 calls *or* use more than 500 minutes, she is likely to say, “Add $20 to the bill of anyone who makes more than 100 calls and to anyone who has used more than 500 minutes." Her request contains the word *and* between two types of people—those who made many calls and those who used many minutes—placing the emphasis on the people. However, each decision you make is about the added $20 for a single customer who has met one criterion *or* the other *or* both. In other words, the OR condition is between each customer's attributes, and not between different customers. Instead of the manager's previous statement, it would be clearer if she said, "Add $20 to the bill of anyone who has made more than 100 calls or has used more than 500 minutes," but you can't count on people to speak like computers. As a programmer, you have the job of clarifying what really is being requested. Often, a casual request for A *and* B logically means a request for A *or* B.

The way we use English can cause another type of error when you are required to find whether a value falls between two other values. For example, a movie theater manager might say, "Provide a discount to patrons who are under 13 years old and to 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 4-15; however, this logic will not provide a discounted price for any movie patron. You must remember that every time the decision is made in Figure 4-15, it is made for a single movie patron. If `patronAge` contains a value lower than 13, then it cannot possibly contain a value over 64. Similarly, if `patronAge` contains a value over 64, there is no way it can contain a lesser value. Therefore, no value could be stored in `patronAge` for which both parts of the AND question could be true—and the price will never be set to the discounted price for any patron. Figure 4-16 shows the correct logic.

Figure 4-15

Incorrect Logic That Attempts to Provide a Discount for Young and Old Movie Patrons

Figure 4-16

Correct Logic That Provides a Discount for Young and Old Movie Patrons

A similar error can occur in your logic if the theater manager says something like, “Don’t give a discount—that is, do 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 to resemble Figure 4-17. 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 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 4-18 shows the correct logic for this decision.

Figure 4-17

Incorrect Logic That Attempts to Charge Full Price for Patrons Whose Age is Over 12 and Under 65

Figure 4-18

Correct Logic That Charges Full Price for Patrons Whose Age is Over 12 and Under 65

Besides `AND` and `OR` operators, most languages support a `NOT` operator. You use the **logical NOT operator** (A symbol that reverses the meaning of a Boolean) to reverse the meaning of a Boolean expression. For example, the statement `if NOT (age < 21) output "OK"` outputs *OK* when `age` is greater than or equal to 21. The `NOT` operator is unary instead of binary—that is, you do not use it between two expressions, but you use it in front of a single expression. In C++, Java, and C#, the exclamation point is the symbol used for the `NOT` operator. In Visual Basic, the operator is `Not`.

Watch the video *Looking in Depth at AND and OR Decisions*.

Two Truths & A Lie

Understanding *OR* Logic

1. In an OR selection, two or more conditions must be met in order for an

event to take place.

T F

2. When you use an OR selection, you can choose to ask either question first and still achieve a usable program.

   T F

3. The general rule is: In an OR decision, first ask the question that is more likely to be true.

   T F

4-5 Making Selections within Ranges

You often need to take action when a variable falls within a range of values. For example, suppose your company provides various customer discounts based on the number of items ordered, as shown in Figure 4-19.

Figure 4-19

Discount Rates Based on Items Ordered

When you write the program that determines a discount rate based on the number of items, you could make hundreds of decisions, such as `itemQuantity = 1?`, `itemQuantity = 2?`, and so on. However, it is more convenient to find the correct discount rate by using a range check.

When you use a **range check** (The comparison of a variable to a series of values that mark the limiting ends of ranges.) , you compare a variable to a series of values that mark the limiting ends of ranges. To perform a range check, make comparisons using either the lowest or highest value in each range of values. For example, to find each discount rate listed in Figure 4-19, you can use one of the following techniques:

- Make comparisons using the low ends of the ranges.
    - You can ask: Is `itemQuantity` less than 11? If not, is it less than 25? If not, is it less than 51? (If it's possible the value is negative, you would also check for a value less than 0 and take appropriate action if it is.)
    - You can ask: Is `itemQuantity` greater than or equal to 51? If not, is it greater than or equal to 25? If not, is it greater than or equal to 11? (If it's possible the value is negative, you would also check for a value greater than or equal to 0 and take appropriate action if it is not.)
- Make comparisons using the high ends of the ranges
    - You can ask: Is `itemQuantity` greater than 50? If not, is it greater than 24? If not, is it greater than 10? (If there is a maximum allowed value for `itemQuantity`, you would also check for a value greater than that limit and take appropriate action if it is.)
    - You can ask: Is `itemQuantity` less than or equal to 10? If not, is it less than or equal to 24? If not, is it less than or equal to 50? (If there is a maximum allowed value for `itemQuantity`, you would also check for a value less than or equal to that limit and take appropriate action if it is not.)

Figure 4-20 shows the flowchart and pseudocode that represent the logic for a program that determines the correct discount for each order quantity. In the decision-making process, `itemsOrdered` is compared to the high end of the lowest-range group (`RANGE1`). If `itemsOrdered` is less than or equal to that value, then you know the correct discount, `DISCOUNT1`; if not, you continue checking. If `itemsOrdered` is less than or equal to the high end of the next range (`RANGE2`), then the customer's discount is `DISCOUNT2`; if not, you continue checking, and the customer's discount eventually is set to `DISCOUNT3` or `DISCOUNT4`. In the pseudocode in Figure 4-20, notice how each associated `if`, `else`, and `endif` aligns vertically.

Figure 4-20

Flowchart and Pseudocode of Logic That Selects Correct Discount Based on Items

In computer memory, a percent sign (%) is not stored with a numeric value that represents a percentage. Instead, the mathematical equivalent is stored. For example, 15% is stored as 0.15.

For example, consider an order for 30 items. The expression `itemsOrdered <= RANGE1` evaluates as false, so the `else` clause of the decision executes. There, `itemsOrdered <= RANGE2` also evaluates to false, so its `else` clause executes. The expression `itemsOrdered <= RANGE3` is true, so `customerDiscount` becomes `DISCOUNT3`, which is 0.15. Walk through the logic with other values for `itemsOrdered` and verify for yourself that the correct discount is applied each time.

4-5a Avoiding Common Errors When Using Range Checks

When new programmers perform range checks, they are prone to including logic that has too many decisions, entailing more work than is necessary.

Figure 4-21 shows a program segment that contains a range check in which the programmer has asked one question too many—the shaded question in the figure. If you know that `itemsOrdered` is not less than or equal to `RANGE1`, not less than or equal to `RANGE2`, and not less than or equal to `RANGE3`, then `itemsOrdered` must be greater than `RANGE3`. Asking whether `itemsOrdered` is greater than `RANGE3` is a waste of time; no customer order can ever travel the logical path on the far left of the flowchart. You might say such a path is a **dead** (A logical path that can never be traveled.) or **unreachable path** (A logical path that can never be traveled.) , and that the statements written there constitute dead or unreachable code. Although a program that contains such logic will execute and assign the correct discount to customers who order more than 50 items, providing such a path is inefficient.

Figure 4-21

Inefficient Range Selection Including Unreachable Path

In Figure 4-21, it is easier to see the useless path in the flowchart than in the pseudocode representation of the same logic. However, when you use an `if` without an `else`, you are doing nothing when the question's answer is false.

When you ask questions of human beings, you sometimes already know the answers. For example, a good trial lawyer seldom asks a question in court 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 4-22 shows an inefficient range selection that asks two unneeded questions. In the figure, if `itemsOrdered` is less than or equal to `RANGE1`, `customerDiscount` is set to `DISCOUNT1`. If `itemsOrdered` is not less than or equal to `RANGE1`, then it must be greater than `RANGE1`, so the next decision (shaded in the figure) is unnecessary. The computer logic will never execute the shaded decision unless `itemsOrdered` is already greater than `RANGE1`—that is, unless the logic follows the false branch of the first selection. If you use the logic in Figure 4-22, you are wasting computer time asking a question that has previously been answered. The same logic applies to the second shaded decision in Figure 4-22. Beginning programmers sometimes justify their use of unnecessary questions as “just making really sure.” Such caution is unnecessary when writing computer logic.

Figure 4-22

Inefficient Range Selection Including Unnecessary Questions

Two Truths & A Lie

Making Selections within Ranges

1. When you perform a range check, you compare a variable to every value in a series of ranges.

T F

2. You can perform a range check by making comparisons using the lowest value in each range of values you are using.

   T F

3. You can perform a range check by making comparisons using the highest value in each range of values you are using.

   T F

4-6 Understanding Precedence When Combining AND and OR Operators

Most programming languages allow you to combine as many AND and OR operators in an expression as you need. For example, assume that you need to achieve a score of at least 75 on each of three tests to pass a course. You can declare a constant MIN_SCORE equal to 75 and test the multiple conditions with a statement like the following:

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

On the other hand, if you need to pass only one of three tests to pass a course, then the logic is as follows:

```
if score1 >= MIN_SCORE OR score2 >= MIN_SCORE OR score3 >= MIN_SCORE 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 do, the AND operators take **precedence** (The quality of an operation that determines the order in which it is evaluated.) , meaning the Boolean values of their expressions are evaluated first.

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

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

Don't Do It
The AND evaluates first, which is not the intention.

For example, assume that 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 `if` statement, the part of the expression that contains the AND operator, `age >= 65 AND rating = "G"`, is evaluated 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 then
   output "Discount applies"
endif
```

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

```
if aTrueExpression OR aFalseExpression then
   output "Discount applies"
endif
```

The combination true OR false evaluates as true. Therefore, the string "Discount applies" is output when it should not be.

Many programming languages allow you to use parentheses to correct the logic and force the OR expression to be evaluated first, as shown in the following pseudocode:

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

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

```
if aTrueExpression AND rating = "G" then
   output "Discount applies"
endif
```

In this statement, 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.

You can use the following techniques to avoid confusion when mixing AND and OR operators:

- You can use parentheses to override the default order of operations.
- You can use parentheses for clarity even though they do not change what the order of operations would be without them. For example, if a customer should be between 12

and 19 or have a school ID to receive a high school discount, you can use the expression `(age > 12 AND age < 19) OR validId = "Yes"`, even though the evaluation would be the same without the parentheses.

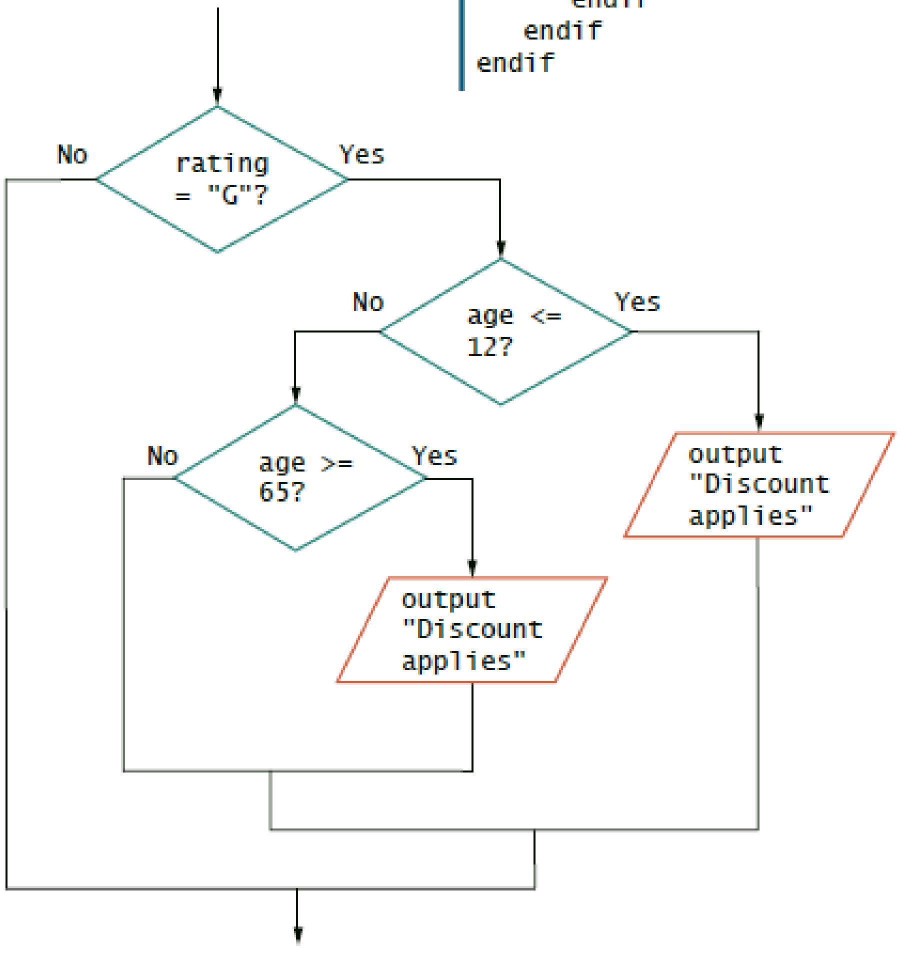
- You can use nesting `if` statements instead of using `AND` and `OR` operators. With the flowchart and pseudocode shown in Figure 4-23, it is clear which movie patrons receive the discount. In the flowchart, you can see that the `OR` is nested entirely within the Yes branch of the `rating = "G"?` selection. Similarly, in the pseudocode in Figure 4-23, 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 `age` at all.

Figure 4-23

Nested Decisions That Determine Movie Patron Discount

```
Significant declarations:
   string rating
   num age
```

```
if rating = "G" then
   if age <= 12 then
      output "Discount applies"
   else
      if age >= 65 then
         output "Discount applies"
      endif
   endif
endif
```



Two Truths & A Lie

Understanding Precedence When Combining AND and OR Operators

1. Most programming languages allow you to combine as many `AND` and `OR` operators in an expression as you need.

   T F

2. When you combine `AND` and `OR` operators, the `OR` operators take precedence, meaning their Boolean values are evaluated first.

   T F

3. You always can avoid the confusion of mixing AND and OR decisions by nesting `if` statements instead of using `AND` and `OR` operators.

   T F

4-7 Chapter Review

4-7a Chapter Summary

- Computer program decisions are made by evaluating Boolean expressions. You can use `if-then-else` or `if-then` structures to choose between two possible outcomes.
- You can use relational comparison operators to compare two operands of the same type. The standard comparison operators are =, >, <, >=, <=, and <>.
- In an AND decision, two conditions must be true for a resulting action to take place. An AND decision requires a nested decision or the use of an `AND` operator. In an AND decision, the most efficient approach is to start by asking the question that is less likely to be true.
- In an OR decision, at least one of two conditions must be true for a resulting action to take place. In an OR decision, the most efficient approach is to start by asking 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 conditional `OR` operator.

- To perform a range check, make comparisons with either the lowest or highest value in each range of comparison values. Common errors that occur when programmers perform range checks include asking unnecessary and previously answered questions.
- When you combine `AND` and `OR` operators in an expression, the `AND` operators take precedence, meaning their Boolean values are evaluated first.

Chapter Review

4-7b Key Terms

Boolean expression (An expression that represents only one of two states, usually expressed as true or false.)

if-then (A structure similar to an if-then-else, but no alternative or “else” action is necessary.)

if-then clause (The part of a decision that holds the resulting action when the Boolean expression in the decision is true.)

else clause (A part of a decision that holds the action or actions that execute only when the Boolean expression in the decision is false.)

relational comparison operators (A symbol that expresses Boolean comparisons. Examples include =, >, <, >=, <=, and <>.)

trivial expressions (A trivial expression is one that always evaluates to the same value.)

compound condition (A condition constructed when multiple decisions are required before determining an outcome.)

AND decision (A decision in which two conditions must both be true for an action to take place.)

nested decision (A decision within the `if-then` or `else` clause of another decision; also called a *nested if*.)

nested if (A decision within the `if-then` or `else` clause of another decision; also called a *nested if*.)

cascading if statement (A series of nested if statements.)

conditional AND operator (A symbol used to combine decisions so that two or more conditions must be true for an action to occur. Also called an *AND operator*.)

AND operator (A symbol used to combine decisions so that two or more conditions must be true for an action to occur. Also called an *AND operator*.)

Truth tables (Truth tables are diagrams used in mathematics and logic to help describe the truth of an entire expression based on the truth of its parts.)

short-circuit evaluation (A logical feature in which each part of a larger expression is evaluated only as far as necessary to determine the final outcome.)

OR decision (A decision that contains two or more conditions; if at least one condition is met, the resulting action takes place.)

conditional OR operator (A symbol used to combine decisions when any one condition can be true for an action to occur. Also called an *OR operator*.)

OR operator (A symbol used to combine decisions when any one condition can be true for an action to occur. Also called an *OR operator*.)

logical NOT operator (A symbol that reverses the meaning of a Boolean)

range check (The comparison of a variable to a series of values that mark the limiting ends of ranges.)

dead (A logical path that can never be traveled.)

unreachable path (A logical path that can never be traveled.)

precedence (The quality of an operation that determines the order in which it is evaluated.)

Chapter Review

4-7c Review Questions

1. The selection statement `if quantity > 100 then discountRate = RATE` is an example of a ____ .

    a. dual-alternative selection

    b. single-alternative selection

c. structured loop

d. all of the above

2. The selection statement `if dayOfWeek = "Sunday" then price = LOWER_PRICE else price = HIGHER_PRICE` is an example of a ____ .

   a. dual-alternative selection

   b. single-alternative selection

   c. unary selection

   d. all of the above

3. All selection statements must have ____ .

   a. a `then` clause

   b. an `else` clause

   c. both a. and b.

   d. none of the above

4. An expression like `amount < 10?` is a(n) ____ expression.

   a. Gregorian

   b. Edwardian

   c. Machiavellian

   d. Boolean

5. Usually, you compare only variables that have the same ____ .

   a. type

   b. size

   c. name

   d. value

6. Symbols such as > and < are known as ____ operators.

   a. arithmetic

   b. sequential

   c. relational comparison

   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. equal to, less than, and greater than

   c. less than, less than or equal to, and not equal to

   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
      output "X"
   endif
endif
```

   a. `if x > 10 OR y > 10 then output "X"`

   b. `if x > 10 AND x > y then output "X"`

   c. `if y > x then output "X"`

d. `if x > 10 AND y > 10 then output "X"`

11. 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. Either Iowa, Missouri, or Wisconsin—it does not matter which one of these three is first.

    d. Any of the five states—it does not matter which one is first.

12. The Boffo Balloon Company makes helium balloons. Large balloons cost $13.00 a dozen, medium-sized balloons cost $11.00 a dozen, and small balloons cost $8.60 a dozen. About 60 percent of the company's sales are of the smallest balloons, 30 percent are medium, and large balloons constitute only 10 percent of sales. Customer order records include customer information, quantity ordered, and size. To write a program that makes the most efficient determination of an order's price based on size ordered, you should ask first whether the size is ____ .

    a. large

    b. medium

    c. small

    d. It does not matter.

13. 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 balloons imprinted with the string `"Happy Valentine's Day"`. To most efficiently select the orders to which a discount applies, you would use ____ .

    a. nested `if` statements using OR logic

    b. nested `if` statements using AND logic

c. three completely separate unnested `if` statements

d. Not enough information is given.

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

```
if department < 3 then
   raise = SMALL_RAISE
else
   if department < 5 then
      raise = MEDIUM_RAISE
   else
      raise = BIG_RAISE
   endif
endif
```

a. `SMALL_RAISE`

b. `MEDIUM_RAISE`

c. `BIG_RAISE`

d. impossible to tell

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

```
if department < 5 then
   raise = SMALL_RAISE
else
   if department < 14 then
      raise = MEDIUM_RAISE
   else
      if department < 9 then
         raise = BIG_RAISE
      endif
   endif
endif
```

a. `SMALL_RAISE`

b. `MEDIUM_RAISE`

c. `BIG_RAISE`

d. impossible to tell

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

```
if department < 2 then
   raise = SMALL_RAISE
else
   if department < 6 then
      raise = MEDIUM_RAISE
   else
      if department < 10 then
         raise = BIG_RAISE
      endif
   endif
endif
```

a. `SMALL_RAISE`

b. `MEDIUM_RAISE`

c. `BIG_RAISE`

d. impossible to tell

17. 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

18. If `sales = 100`, `rate = 0.10`, and `expenses = 50`, which of the following expressions is true?

a. `sales >= expenses AND rate < 1`

b. `sales < 200 OR expenses < 100`

c. `expenses = rate OR sales = rate`

d. two of the above

19. If `a` is true, `b` is true, and `c` is false, which of the following expressions is true?

a. `a OR b AND c`

b. `a AND b AND c`

c. `a AND b OR c`

d. two of the above

20. If `d` is true, `e` is false, and `f` is false, which of the following expressions is true?

    a. `e OR f AND d`

    b. `f AND d OR e`

    c. `d OR e AND f`

    d. two of the above

Chapter Review

4-7d Exercises

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

```
numberBig = 300          numberMedium = 100    numberSmall = 5
wordBig = "Elephant"     wordMedium = "Horse"  wordSmall = "Bug"
```

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

```
numberBig = numberSmall?

numberBig > numberSmall?

numberMedium < numberSmall?

numberBig = wordBig?

numberBig = "Big"?

wordMedium = "Medium"?

wordBig = "Elephant"?

numberMedium <= numberBig / 3?

numberBig >= 200?
numberBig >= numberMedium + numberSmall?

numberBig > numberMedium AND numberBig < numberSmall?

numberBig = 100 OR numberBig > numberSmall?

numberBig < 10 OR numberSmall > 10?

numberBig = 30 AND numberMedium = 100 OR numberSmall = 100?
```

2. Mortimer Life Insurance Company wants several lists of salesperson data. Design a flowchart or pseudocode for the following:

    a. A program that accepts a salesperson's ID number and number of policies sold in the last month, and displays the data only if the salesperson is a high performer—a person who sells more than 25 policies in the month.

    b. A program that accepts salesperson data continuously until a sentinel value is entered and displays a list of high performers.

3. ShoppingBay is an online auction service that requires several reports. Design a flowchart or pseudocode for the following:

    A program that accepts auction data as follows: ID number, item description, length of auction in days, and minimum required bid. Display data for an auction if the minimum required bid is over $100.00.

    A program that continuously accepts auction data until a sentinel value is entered and displays a list of all data for auctions in which the minimum required bid is over $100.00.

    A program that continuously accepts auction data and displays data for every auction in which the minimum bid is $0.00 and the length of the auction is one day or less.

    A program that continuously accepts auction data and displays data for every auction in which the length is between 7 and 30 days inclusive.

    A program that prompts the user for a maximum required bid, and then continuously accepts auction data and displays data for every auction in which the minimum bid is less than or equal to the amount entered by the user.

4. The Dash Cell Phone Company charges customers a basic rate of $5 per month to send text messages. Additional rates are as follows:

- The first 60 messages per month, regardless of message length, are included in the basic bill.
- An additional five cents is charged for each text message after the 60th message, up to 180 messages.
- An additional 10 cents is charged for each text message after the 180th message.
- Federal, state, and local taxes add a total of 12 percent to each bill.

  Design a flowchart or pseudocode for the following:

a. A program that accepts the following data about one customer's bill: customer area code (three digits), customer phone number (seven digits), and number of text messages sent. Display all the data, including the month-end bill both before and after taxes are added.

b. A program that continuously accepts data about text messages until a sentinel value is entered, and displays all the details.

c. A program that continuously accepts data about text messages until a sentinel value is entered, and displays details only about customers who send more than 100 text messages.

d. A program that continuously accepts data about text messages until a sentinel value is entered, and displays details only about customers whose total bill with taxes is over $20.

e. A program that prompts the user for a three-digit area code from which to select bills. Then the program continuously accepts text message data until a sentinel value is entered, and displays data only for messages sent from the specified area code.

5. The Drive-Rite Insurance Company provides automobile insurance policies for drivers. Design a flowchart or pseudocode for the following:

   a. A program that accepts insurance policy data, including a policy number, customer last name, customer first name, age, premium due date (month, day, and year), and number of driver accidents in the last three years. If an entered policy number is not between 1000 and 9999 inclusive, set the policy number to 0. If the month is not between 1 and 12 inclusive, or the day is not correct for the month (for example, not between 1 and 31 for January or 1 and 29 for February), set the month, day, and year to 0. Display the policy data after any revisions have been made.

b. A program that continuously accepts policy holders' data until a sentinel value has been entered, and displays the data for any policy holder over 35 years old.

c. A program that accepts policy holders' data and displays the data for any policy holder who is at least 21 years old.

d. A program that accepts policy holders' data and displays the data for any policy holder no more than 30 years old.

e. A program that accepts policy holders' data and displays the data for any policy holder whose premium is due no later than March 15 any year.

f. A program that accepts policy holders' data and displays the data for any policy holder whose premium is due up to and including January 1, 2014.

g. A program that accepts policy holders' data and displays the data for any policy holder whose premium is due by April 27, 2013.

h. A program that accepts policy holders' data and displays the data for anyone who has a policy number between 1000 and 4000 inclusive, whose policy comes due in April or May of any year, and who has had fewer than three accidents.

6. The Barking Lot is a dog day care center. Design a flowchart or pseudocode for the following:

   a. A program that accepts data for an ID number of a dog's owner, and the name, breed, age, and weight of the dog. Display a bill containing all the input data as well as the weekly day care fee, which is $55 for dogs under 15 pounds, $75 for dogs from 15 to 30 pounds inclusive, $105 for dogs from 31 to 80 pounds inclusive, and $125 for dogs over 80 pounds.

   b. A program that continuously accepts dogs' data until a sentinel value is entered, and displays billing data for each dog.

   c. A program that continuously accepts dogs' data until a sentinel value is entered, and displays billing data for dog owners who owe more than $100.

7. Mark Daniels is a carpenter who creates personalized house signs. He wants

an application to compute the price of any sign a customer orders, based on the following factors:

- The minimum charge for all signs is $30.
- If the sign is made of oak, add $15. No charge is added for pine.
- The first six letters or numbers are included in the minimum charge; there is a $3 charge for each additional character.
- Black or white characters are included in the minimum charge; there is an additional $12 charge for gold-leaf lettering.

  Design a flowchart or pseudocode for the following:

  a. A program that accepts data for an order number, customer name, wood type, number of characters, and color of characters. Display all the entered data and the final price for the sign.

  b. A program that continuously accepts sign order data and displays all the relevant information for oak signs with five white letters.

  c. A program that continuously accepts sign order data and displays all the relevant information for pine signs with gold-leaf lettering and more than 10 characters.

8. Black Dot Printing 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; 30 percent live in the adjacent town of Woodstock. Black Dot wants to encourage employees who live in either town to drive to work together. Design a flowchart or pseudocode for the following:

   a. A program that accepts an employee’s data and displays it with a message that indicates whether the employee is a candidate for the carpool.

   b. A program that continuously accepts employee data until a sentinel value is entered, and displays a list of all employees who are carpool candidates.

9. Amanda Cho, a supervisor in a retail clothing store, wants to acknowledge high-achieving salespeople. Design a flowchart or pseudocode for the following:

a. A program that continuously accepts each salesperson’s first and last names, the number of shifts worked in a month, number of transactions completed this month, and the dollar value of those transactions. Display each salesperson’s name with a productivity score, which is computed by first dividing dollars by transactions and dividing the result by shifts worked. Display three asterisks after the productivity score if it is 50 or higher.

b. A program that accepts each salesperson’s data and displays the name and a bonus amount. The bonuses will be distributed as follows:

- If the productivity score is 30 or less, the bonus is $25.
- If the productivity score is 31 or more and less than 80, the bonus is $50.
- If the productivity score is 80 or more and less than 200, the bonus is $100.
- If the productivity score is 200 or higher, the bonus is $200.

c. Modify Exercise 9b to reflect the following new fact, and have the program execute as efficiently as possible:

- Sixty percent of employees have a productivity score greater than 200.

Find the Bugs

10. Your downloadable files for Chapter 4 include DEBUG04-01.txt, DEBUG04-02.txt, and DEBUG04-03.txt. Each file starts with some comments that describe the problem. Comments are lines that begin with two slashes (//). Following the comments, each file contains pseudocode that has one or more bugs you must find and correct.

Game Zone

11. In Chapter 2, you learned that many programming languages allow you to generate a random number between 1 and a limiting value named `LIMIT` by using a statement similar to `randomNumber = random(LIMIT)`. Create the

logic for a guessing game in which the application generates a random number and the player tries to guess it. Display a message indicating whether the player's guess was correct, too high, or too low. (After you finish Chapter 5, you will be able to modify the application so that the user can continue to guess until the correct answer is entered.)

12. Create a lottery game application. Generate three random numbers, each between 0 and 9. Allow the user to guess three numbers. Compare each of the user's guesses to the three random numbers and display a message that includes the user's guess, the randomly determined three digits, and the amount of money the user has won, as shown in Table 4-4.

Table 4-4

Awards for Matching Numbers in Lottery Game

Matching Numbers	**Award ($)**
Any one matching	10
Two matching	100
Three matching, not in order	1000
Three matching in exact order	1, 000, 000
No matches	0

Make certain that your application accommodates repeating digits. For example, if a user guesses 1, 2, and 3, and the randomly generated digits are 1, 1, and 1, do not give the user credit for three correct guesses—just one.

Up for Discussion

13. 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? Explain your answer.

14. 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? Explain your answer.

15. 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 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?