CHAPTER 5

Making Decisions

In this chapter, you will:

- Plan decision-making logic
- Make decisions with the `if` and `if...else` structures
- Use multiple statements in `if` and `if...else` clauses
- Nest `if` and `if...else` statements
- Use AND and OR operators
- Make accurate and efficient decisions
- Use the `switch` statement
- Use the conditional and NOT operators
- Assess operator precedence
- Add decisions and constructors to instance methods

Planning Decision-Making Logic

When computer programmers write programs, they rarely just sit down at a keyboard and begin typing. Programmers must plan the complex portions of programs using paper and pencil. Programmers often use **pseudocode**, a tool that helps them plan a program's logic by writing plain English statements. Using pseudocode requires that you write down the steps needed to accomplish a given task. You write pseudocode in everyday language, not the syntax used in a programming language. In fact, a task you write in pseudocode does not have to be computer-related. If you have ever written a list of directions to your house—for example, (1) go west on Algonquin Road, (2) turn left on Roselle Road, (3) enter expressway heading east, and so on—you have written pseudocode. A **flowchart** is similar to pseudocode, but you write the steps in diagram form, as a series of shapes connected by arrows.

Some programmers use a variety of shapes to represent different tasks in their flowcharts, but you can draw simple flowcharts that express very complex situations using just rectangles and diamonds. You use a rectangle to represent any unconditional step and a diamond to represent any decision. For example, Figure 5-1 shows a flowchart describing driving directions to a friend's house. The logic in Figure 5-1 is an example of a logical structure called a **sequence structure**—one step follows another unconditionally. A sequence structure might contain any number of steps, but when one task follows another with no chance to branch away or skip a step, you are using a sequence.

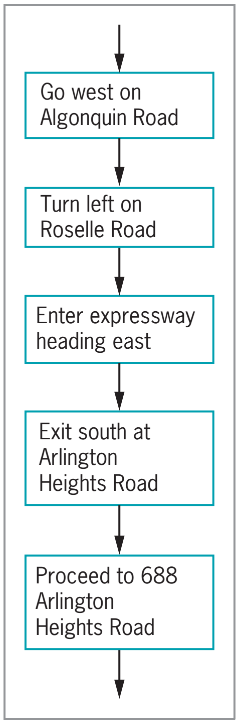

Go west on Algonquin Road
Turn left on Roselle Road
Enter expressway heading east
Exit south at Arlington Heights Road
Proceed to 688 Arlington Heights Road

Figure 5-1 Flowchart of a series of sequential steps

Sometimes, logical steps do not follow in an unconditional sequence—some tasks might or might not occur based on decisions you make. To represent a decision, flowchart creators use a diamond shape to hold a question, and they draw paths to alternative courses of action emerging from the sides of the diamonds. Figure 5-2 shows a flowchart describing directions in which the execution of some steps depends on decisions.

Figure 5-2 includes a **decision structure**—one that involves choosing between alternative courses of action based on some value within a program. For example, the program that produces your paycheck can make decisions about the proper amount to withhold for taxes, the program that guides a missile can alter its course, and a program that monitors your blood pressure during surgery can determine when to sound an alarm. Making decisions is what makes computer programs seem “smart.”

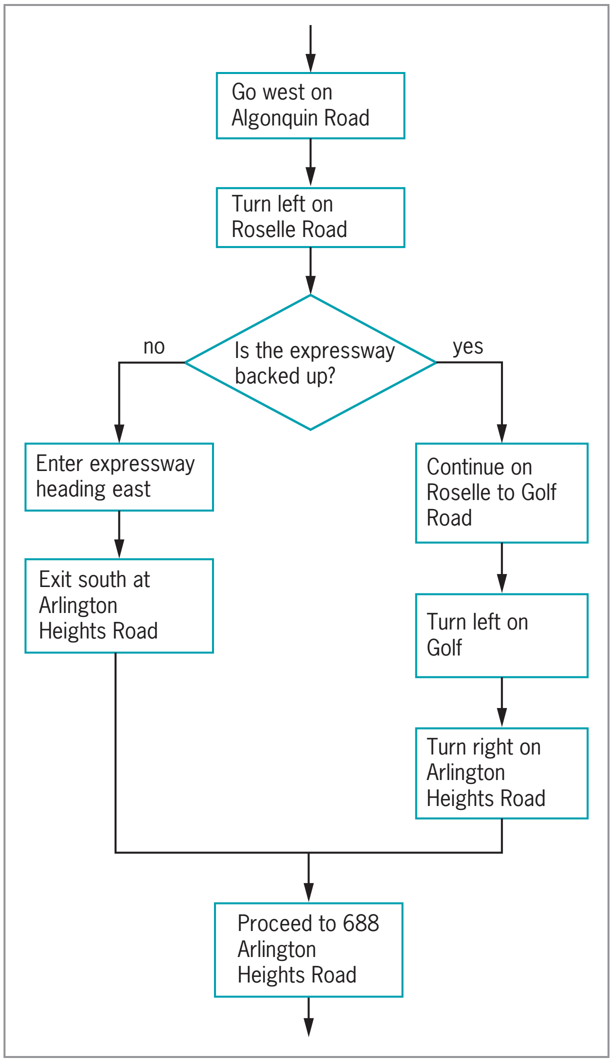

Go west on Algonquin Road
Turn left on Roselle Road
Is the expressway backed up?
no
yes
Enter expressway heading east
Exit south at Arlington Heights Road
Continue on Roselle to Golf Road
Turn left on Golf
Turn right on Arlington Heights Road
Proceed to 688 Arlington Heights Road

Figure 5-2 Flowchart including a decision

When reduced to their most basic form, all computer decisions are yes-or-no decisions. That is, the answer to every computer question is yes or no (or true or false, or on or off). This is because computer circuitry consists of millions of tiny switches that are either on or off, and the result of every decision sets one of these switches in memory. As you learned in Chapter 2, the values `true` and `false` are **Boolean values**; every computer decision results in a Boolean value. Thus, internally, a program never asks, for example, “What number did the user enter?” Instead, the decisions might be “Did the user enter a *1*?” “If not, did the user enter a *2*?” “If not, did the user enter a *3*?”

Sir George Boole lived from 1815 to 1864. He developed a type of linguistic algebra, based on *0*s and *1*s, the three most basic operations of which were (and still are) AND, OR, and NOT. All computer logic is based on his discoveries.

TWO TRUTHS & A LIE

Planning Decision-Making Logic

1. Pseudocode and flowcharts are both tools that are used to check the syntax of computer programs.
2. In a sequence structure, one step follows another unconditionally.
3. In a decision structure, alternative courses of action are chosen based on a Boolean value.

The false statement is #1. Pseudocode and flowcharts are both tools that help programmers plan a program's logic.

The `if` and `if...else` Structures

In Java, the simplest statement you can use to make a decision is the **`if` statement**. For example, suppose you have declared an integer variable named `quizScore`, and you want to display a message when the value of `quizScore` is 10. The `if` statement in Figure 5-3 makes the decision whether to produce output. Note that the double equal sign (==) is used to determine equality; it is Java's **equivalency operator**.

```
if(quizScore == 10)
   System.out.println("The score is perfect");
```

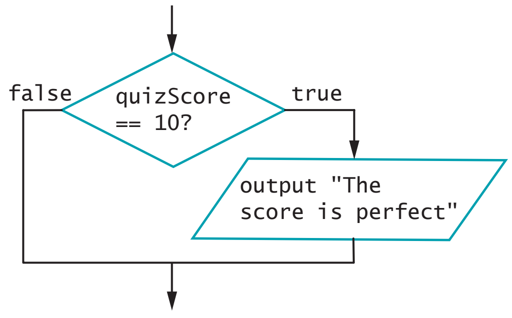


Figure 5-3 A Java `if` statement

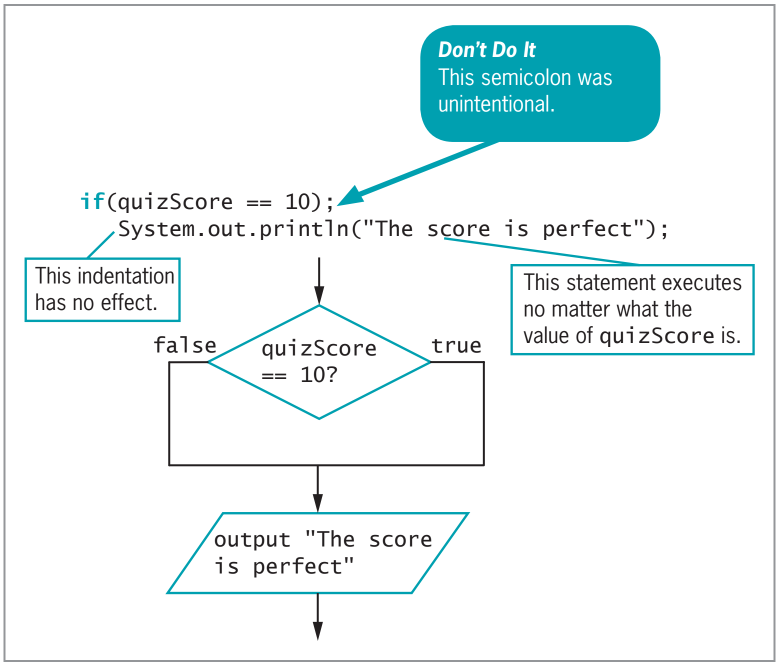
In the example in Figure 5-3, if `quizScore` holds the value 10, the Boolean value of the expression `quizScore == 10` is `true`, and the subsequent output statement executes. If the value of the expression `quizScore == 10` is `false`, the output statement does not execute.

As the flowchart segment shows, whether the tested expression is true or false, the program continues and executes any statements that follow the `if` statement.

An `if` statement always includes parentheses. Within the parentheses, you can place any Boolean expression. Most often you use a comparison that includes one of the relational operators you learned about in Chapter 2 (==, <, >, <=, >=, or !=). However, you can use any expression that evaluates as true or false, such as a simple `boolean` variable or a call to a method that returns a `boolean` value.

Pitfall: Misplacing a Semicolon in an `if` Statement

In a Java `if` statement, the Boolean expression, such as `(quizScore == 10)`, must appear within parentheses. In Figure 5-3, there is no semicolon at the end of the first line of the `if` statement following the parentheses because the statement does not end there. The statement ends after the `println()` call, so that is where you type the semicolon. You could type the entire `if` statement on one line and it would execute correctly; however, the two-line format for the `if` statement is more conventional and easier to read, so you usually type `if` and the Boolean expression on one line, press Enter, and then indent a few spaces before coding the action that occurs if the Boolean expression evaluates as `true`. Be careful—if you use the two-line format and type a semicolon at the end of the first line, as in the example shown in Figure 5-4, the results might not be what you intended.



```
if(quizScore == 10);
   System.out.println("The score is perfect");
```

Don't Do It
This semicolon was unintentional.
This indentation has no effect.
This statement executes no matter what the value of quizScore is.
false
quizScore == 10?
true
output "The score is perfect"

Figure 5-4 Logic that executes when an extra semicolon is inserted in an `if` statement

When the Boolean expression in Figure 5-4 is `true`, an **empty statement** that contains only a semicolon executes. Whether the tested expression evaluates as `true` or `false`, the decision is over immediately, and execution continues with the next independent statement that displays a message. In this case, because of the incorrect semicolon, the `if` statement accomplishes nothing.

Pitfall: Using the Assignment Operator Instead of the Equivalency Operator

Another common programming error occurs when a programmer uses a single equal sign rather than the double equal sign when attempting to determine equivalency. The expression `quizScore = 10` does not compare `quizScore` to 10; instead, it attempts to assign the value 10 to `quizScore`. When the expression `quizScore = 10` is used in the `if` statement, the assignment is illegal because only Boolean expressions are allowed. The confusion arises in part because the single equal sign is used within Boolean expressions in `if` statements in several older programming languages, such as COBOL, Pascal, and BASIC. Adding to the confusion, Java programmers use the word *equals* when speaking of equivalencies. For example, you might say, "If `quizScore` *equals* 10…".

The expression `if(x = true)` will compile only if `x` is a `boolean` variable, because it would be legal to assign `true` to `x`. However, such a statement would be useless because the value of such an expression could never be `false`.

An alternative to using a Boolean expression in an `if` statement, such as `quizScore == 10`, is to store the Boolean expression's value in a Boolean variable. For example, if `isPerfectScore` is a Boolean variable, then the following statement compares `quizScore` to 10 and stores `true` or `false` in `isPerfectScore`:

```
isPerfectScore = (quizScore == 10);
```

Then, you can write the `if` statement as:

```
if(isPerfectScore)
   System.out.println("The score is perfect");
```

This adds an extra step to the program, but makes the `if` statement more similar to an English-language statement.

When comparing a variable to a constant, some programmers prefer to place the constant to the left of the comparison operator, as in `10 == quizScore`. This practice is a holdover from other programming languages, such as C++, in which an accidental assignment might be made when the programmer types the assignment operator (a single equal sign) instead of the comparison operator (the double equal sign). In Java, the compiler does not allow you to make a mistaken assignment in a Boolean expression.

Pitfall: Attempting to Compare Objects Using the Relational Operators

You can use the standard relational operators (==, <, >, <=, >=, and !=) to compare the values of primitive data types such as `int` and `double`. However, you cannot use <, >, <=, or >= to compare objects; a program containing such comparisons does not compile. You can use the equals and not equals comparisons (== and !=) with objects, but when you use them, you compare the objects' memory addresses instead of their values. Recall that every object name is a reference; the equivalency operators compare objects' references. In other words, == only yields `true` for two objects when they refer to the same object in memory, not when they are different objects with the same value. To compare the values of objects, you should write specialized methods. Remember, `String`s are objects, so do not use == to compare `String`s. You will learn more about this in the chapter *Characters, Strings, and the `StringBuilder`*.

Although object names are references, their field names are not references if they represent simple, primitive data types. You can compare the values between objects' fields by using public accessor methods. For example, suppose you have created a class named `Student` with a `double` grade point average field and a nonstatic public method named `getGpa()`. After instantiating two objects named `student1` and `student2`, you can write a statement such as the following:

```
if(student1.getGpa() > student2.getGpa())
   System.out.println("The first student has a higher gpa");
```

The values represented by `student1.getGpa()` and `student2.getGpa()` are both `double`s, so they can be compared using any of the relational operators.

The if…else Structure

Consider the following statement:

```
if(quizScore == 10)
   System.out.println("The score is perfect");
```

Such a statement is sometimes called a **single-alternative `if`** because the program only performs an action, or not, based on one alternative; in this example, a statement is displayed when `quizScore` is 10. Often, you require two options for the course of action following a decision. A **dual-alternative `if`** is the decision structure you use when you need to take one or the other of two possible courses of action. For example, you would use a dual-alternative `if` structure if you wanted to display one message when the value of `quizScore` is 10 and a different message when it is not. In Java, the **`if…else` statement** provides the mechanism to perform one action when a Boolean expression evaluates as `true` and to perform a different action when a Boolean expression evaluates as `false`.

The code in Figure 5-5 displays one of two messages. In this example, when the value of `quizScore` is 10, the **`if` clause** of the statement executes, displaying the message "The score is perfect". When `quizScore` is any other value, the **`else` clause** of the statement executes and the program displays the message "No, it's not". You can code an `if` without an `else`, but it is illegal to code an `else` without an `if` that precedes it.

```
if(quizScore == 10)
   System.out.println("The score is perfect");
else
   System.out.println("No, it's not");
```

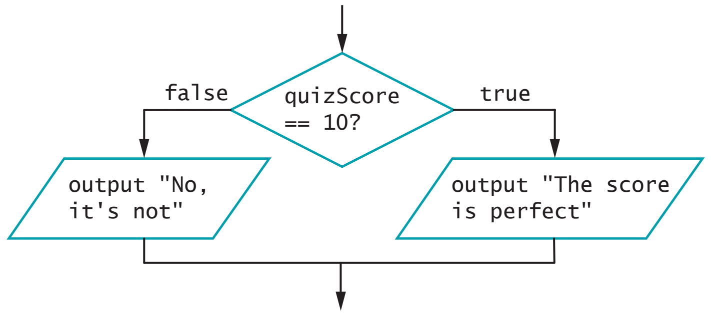


Figure 5-5 An `if...else` structure

The indentation shown in the example code in Figure 5-5 is not required but is standard usage. You vertically align the keyword `if` with the keyword `else`, and then indent the action statements that depend on the evaluation.

When you execute an `if...else` statement, only one of the resulting actions takes place depending on the evaluation of the Boolean expression. Each statement, the one following the `if` and the one following the `else`, is a complete statement, so each ends with a semicolon.

Watch the video *Making Decisions*.

TWO TRUTHS & A LIE

The `if` and `if...else` Structures

1. In a Java `if` statement, the keyword `if` is followed by a Boolean expression within parentheses.
2. In a Java `if` statement, a semicolon follows the Boolean expression.
3. When determining equivalency in Java, you use a double equal sign.

The false statement is #2. In a Java `if` statement, a semicolon ends the statement. It is used following the action that should occur if the Boolean expression is true. If a semicolon follows the Boolean expression, then the body of the `if` statement is empty.

You Do It

Using an if...else Statement

In this section, you start writing a program for Sacks Fifth Avenue, a nonprofit thrift shop. The program determines which volunteer to assign to price a donated item. To begin, you prompt the user to answer a question about whether a donation is clothing or some other type, and then the program displays the name of the volunteer who handles such donations. Clothing donations are handled by Regina, and other donations are handled by Marco.

1. Open a new text file, and then enter the first lines of code to create a class named `AssignVolunteer`. You import the `Scanner` class so that you can use keyboard input. The class contains a `main()` method that performs all the work of the class:

```
import java.util.Scanner;
public class AssignVolunteer
{
   public static void main(String[] args)
   {
```

2. On new lines, declare the variables and constants this application uses. The user will be prompted to enter one of the values stored in the two constants. That value will then be assigned to the integer `donationType` and compared to the `CLOTHING_CODE` constant. Then, based on the results of that comparison, the program will assign the value of one of the `PRICER` constants to the `String` variable `volunteer`.

```
int donationType;
String volunteer;
final int CLOTHING_CODE = 1;
final int OTHER_CODE = 2;
final String CLOTHING_PRICER = "Regina";
final String OTHER_PRICER = "Marco";
```

3. Define the input device, and then add the code that prompts the user to enter a *1* or *2* for the donation type. Accept the response, and assign it to `donationType`:

```
Scanner input = new Scanner(System.in);
System.out.println("What type of donation is this?");
System.out.print("Enter " + CLOTHING_CODE + " for clothing, " +
    OTHER_CODE + " for anything else... ");
donationType = input.nextInt();
```

(continues)

(continued)

4. Use an `if...else` statement to choose the name of the volunteer to be assigned to the `volunteer String`, as follows:

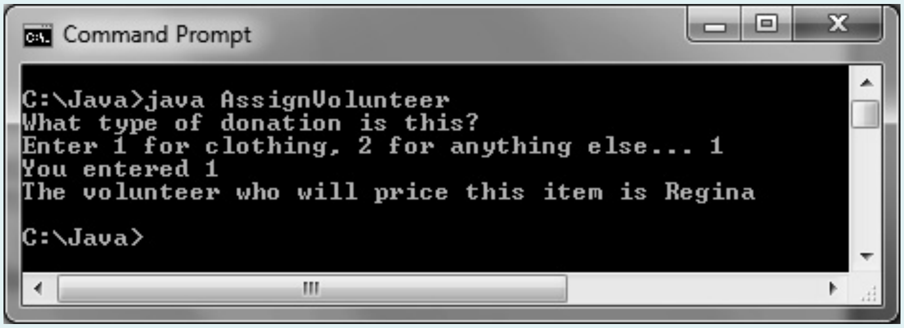
```
if(donationType == CLOTHING_CODE)
    volunteer = CLOTHING_PRICER;
else
    volunteer = OTHER_PRICER;
```

5. Display the chosen code and corresponding volunteer's name:

```
System.out.println("You entered " + donationType);
System.out.println("The volunteer who will price this item is " +
    volunteer);
```

6. Type the two closing curly braces to end the `main()` method and the `AssignVolunteer` class.

7. Save the program as **AssignVolunteer.java**, and then compile and run the program. Confirm that the program selects the correct volunteer when you choose *1* for a clothing donation or *2* for any other donation type. For example, Figure 5-6 shows a typical execution of the program when the user enters *1* for a clothing donation.

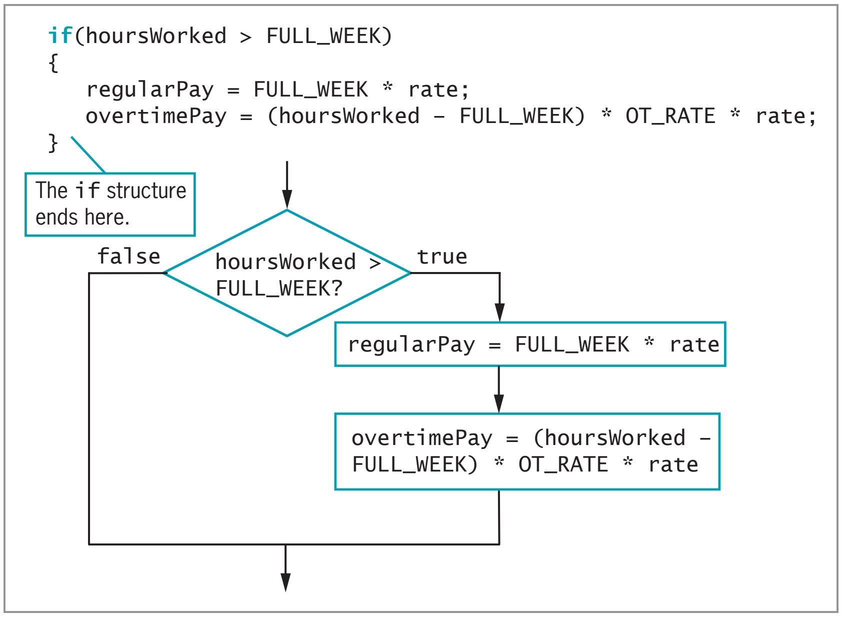


```
C:\Java>java AssignVolunteer
What type of donation is this?
Enter 1 for clothing, 2 for anything else... 1
You entered 1
The volunteer who will price this item is Regina

C:\Java>
```

Figure 5-6 Typical execution of the `AssignVolunteer` application

Using Multiple Statements in `if` and `if...else` Clauses

Often, you want to take more than one action following the evaluation of a Boolean expression within an `if` statement. For example, you might want to display several separate lines of output or perform several mathematical calculations. To execute more than one statement that depends on the evaluation of a Boolean expression, you use a pair of curly braces to place the dependent statements within a block. For example, the program segment shown in Figure 5-7 determines whether an employee has worked more than the value of a FULL_WEEK constant; if so, the program computes regular and overtime pay.



```
if(hoursWorked > FULL_WEEK)
{
   regularPay = FULL_WEEK * rate;
   overtimePay = (hoursWorked - FULL_WEEK) * OT_RATE * rate;
}
```

The if structure ends here.
false
hoursWorked > FULL_WEEK?
true
regularPay = FULL_WEEK * rate
overtimePay = (hoursWorked - FULL_WEEK) * OT_RATE * rate

Figure 5-7 An if structure that determines pay

When you place a block within an if statement, it is crucial to place the curly braces correctly. For example, in Figure 5-8, the curly braces have been omitted. Within the code segment in Figure 5-8, when hoursWorked > FULL_WEEK is true, regularPay is calculated and the if expression ends. The next statement that computes overtimePay executes every time the program runs, no matter what value is stored in hoursWorked. This last statement does not depend on the if statement; it is an independent, stand-alone statement. The indentation might be deceiving; it looks as though two statements depend on the if statement, but indentation does not cause statements following an if statement to be dependent. Rather, curly braces are required if multiple statements must be treated as a block.

When you create a block, you do not have to place multiple statements within it. It is perfectly legal to place curly braces around a single statement. For clarity, some programmers always use curly braces to surround the actions in an if statement, even when there is only one statement in the block.

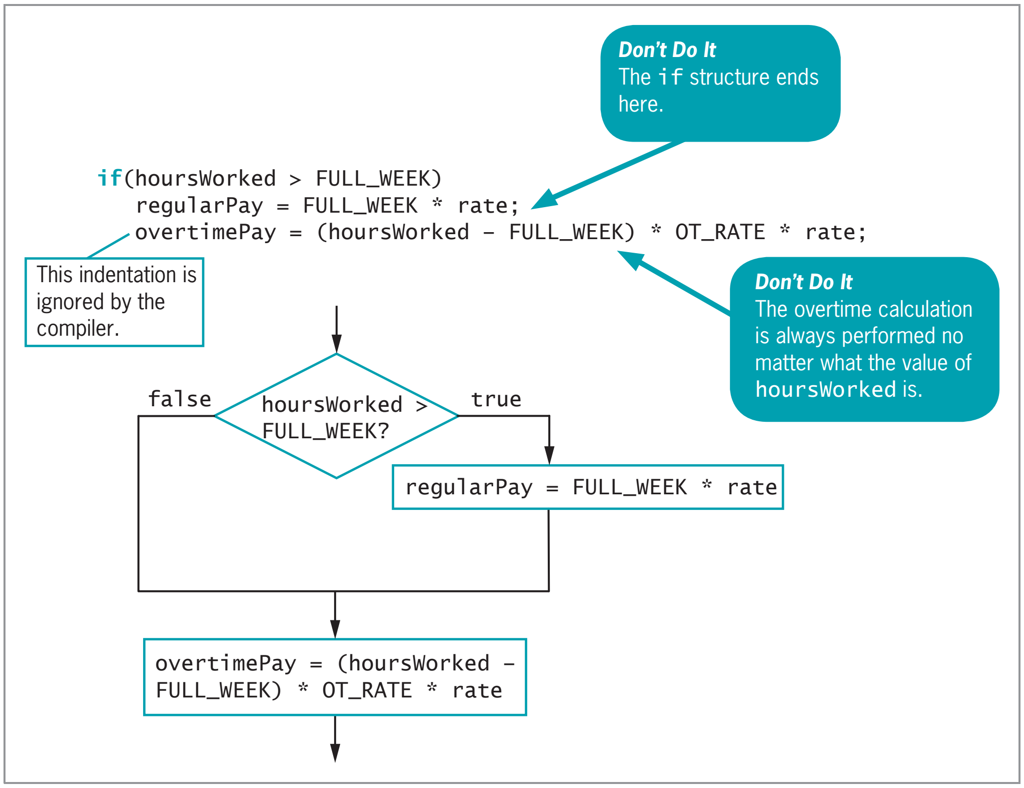

Don't Do It
The if structure ends here.
if(hoursWorked > FULL_WEEK)
regularPay = FULL_WEEK * rate;
overtimePay = (hoursWorked – FULL_WEEK) * OT_RATE * rate;
This indentation is ignored by the compiler.
Don't Do It
The overtime calculation is always performed no matter what the value of hoursWorked is.
false
hoursWorked > FULL_WEEK?
true
regularPay = FULL_WEEK * rate
overtimePay = (hoursWorked – FULL_WEEK) * OT_RATE * rate

Figure 5-8 Erroneous overtime pay calculation with missing curly braces

Because the curly braces are missing, regardless of whether `hoursWorked` is more than `FULL_WEEK`, the last statement in Figure 5-8 is a new stand-alone statement that is not part of the `if`, and so it always executes. If `hoursWorked` is 30, for example, and `FULL_WEEK` is 40, then the program calculates the value of `overtimePay` as a negative number (because 30 minus 40 results in –10). Therefore, the output is incorrect. Correct blocking is crucial to achieving valid output.

When you fail to block statements that should depend on an `if`, and you also use an `else` clause, the program will not compile. For example, consider the following code:

```
if(hoursWorked > FULL_WEEK)
   regularPay = FULL_WEEK * rate;
   overtimePay = (hoursWorked – FULL_WEEK) * OT_RATE * rate;
else
   regularPay = FULL_WEEK * rate;
```

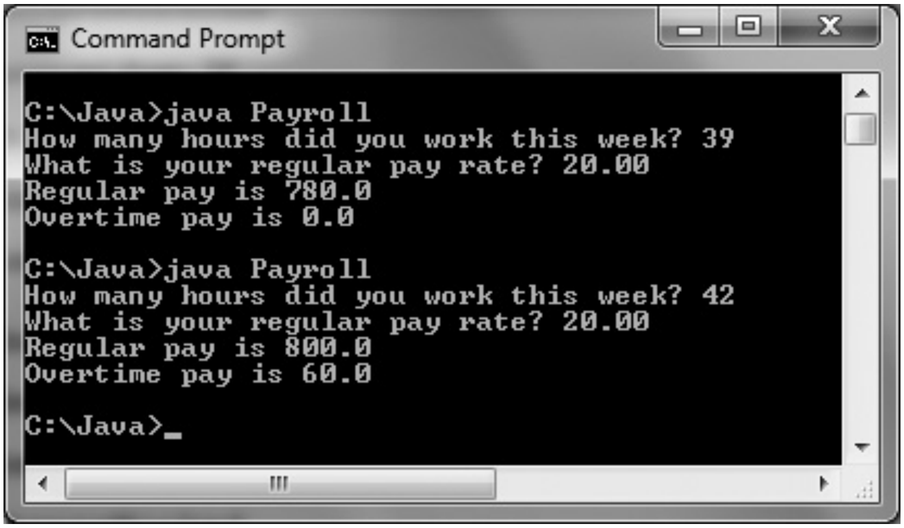
Don't Do It
These statements should be blocked.

In this case, the `if` statement ends after the first `regularPay` calculation, and the second complete stand-alone statement performs the `overtimePay` calculation. The third statement in this code starts with `else`, which is illegal. An error message will indicate that the program contains "`else` without `if`".

Just as you can block statements to depend on an if, you can also block statements to depend on an else. Figure 5-9 shows an application that contains an if structure with two dependent statements and an else with two dependent statements. The program executes the final println() statement without regard to the hoursWorked variable's value; it is not part of the if structure. Figure 5-10 shows the output from two executions of the program. In the first execution, the user entered 39 for the hoursWorked value and 20.00 for rate; in the second execution, the user entered 42 for hoursWorked and 20.00 for rate.

```java
import java.util.Scanner;
public class Payroll
{
  public static void main(String[] args)
  {
    double rate;
    double hoursWorked;
    double regularPay;
    double overtimePay;
    final int FULL_WEEK = 40;
    final double OT_RATE = 1.5;
    Scanner keyboard = new Scanner(System.in);
    System.out.print("How many hours did you work this week? ");
    hoursWorked = keyboard.nextDouble();
    System.out.print("What is your regular pay rate? ");
    rate = keyboard.nextDouble();
    if(hoursWorked > FULL_WEEK)
    {
      regularPay = FULL_WEEK * rate;
      overtimePay = (hoursWorked - FULL_WEEK) * OT_RATE * rate;
    }
    else
    {
      regularPay = hoursWorked * rate;
      overtimePay = 0.0;
    }
    System.out.println("Regular pay is " +
       regularPay + "\nOvertime pay is " + overtimePay);
  }
}
```

Figure 5-9 Payroll application containing an if and else clause with blocks



```
C:\Java>java Payroll
How many hours did you work this week? 39
What is your regular pay rate? 20.00
Regular pay is 780.0
Overtime pay is 0.0

C:\Java>java Payroll
How many hours did you work this week? 42
What is your regular pay rate? 20.00
Regular pay is 800.0
Overtime pay is 60.0

C:\Java>_
```

Figure 5-10 Output of the `Payroll` application

When you block statements, you must remember that any variable you declare within a block is local to that block. For example, the following code segment contains a variable named `sum` that is local to the block following the `if`. The last `println()` statement causes an error because the `sum` variable is not recognized:

```
if(a == b)
{
   int sum = a + b;
   System.out.println
      ("The two variables are equal");
}
System.out.println("The sum is " + sum);
```

The `sum` variable is declared in this block...

...so it is not recognized here.

TWO TRUTHS & A LIE

Using Multiple Statements in `if` and `if...else` Clauses

1. To execute more than one statement that depends on the evaluation of a Boolean expression, you use a pair of curly braces to place the dependent statements within a block.
2. Indentation can be used to cause statements following an `if` statement to depend on the evaluation of the Boolean expression.
3. When you declare a variable within a block, it is local to that block.

The false statement is #2. Indentation does not cause statements following an `if` statement to be dependent; curly braces are required if multiple statements must be treated as a block.

You Do It

Using Multiple Statements in if *and* else *Clauses*

In this section, you use a block of code to add multiple actions to an `if…else` statement.

1. Open the `AssignVolunteer` application from the previous “You Do It” section. Change the class name to **`AssignVolunteer2`**, and immediately save the file as **AssignVolunteer2.java**.
2. Add a `String` to the variables. This `String` will be assigned a message that displays the donation type:

```
String message;
```

3. In place of the existing `if…else` statement in the program, insert the following statement that takes two blocked actions for each donation type. It assigns a volunteer and assigns a value to the `message String`.

```
if(donationType == CLOTHING_CODE)
{
   volunteer = CLOTHING_PRICER;
   message = "a clothing donation";
}
else
{
   volunteer = OTHER_PRICER;
   message = "another donation type";
}
```

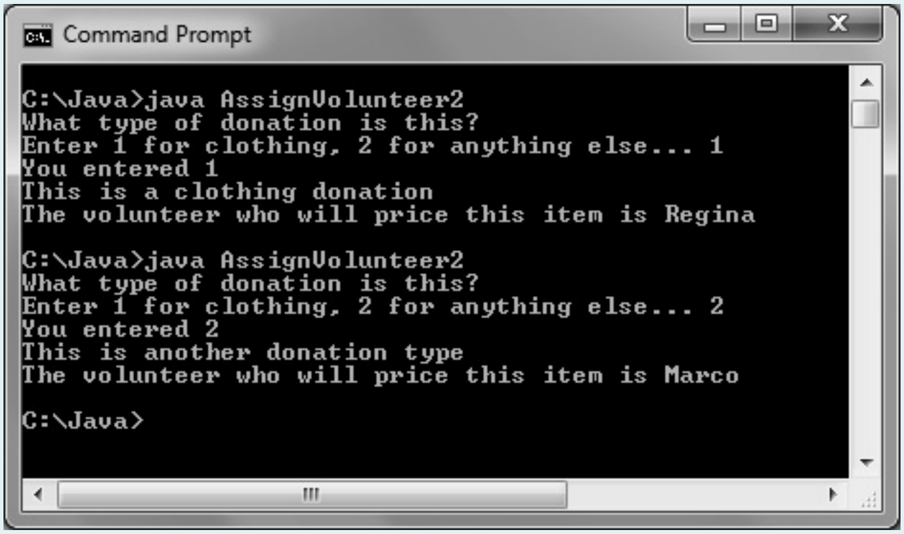
4. Following the output statement that displays the donation type, add the following statement that displays the assigned message:

```
System.out.println("This is " + message);
```

(continues)

(continued)

5. Save the file, and compile and execute the program. Figure 5-11 shows two executions.



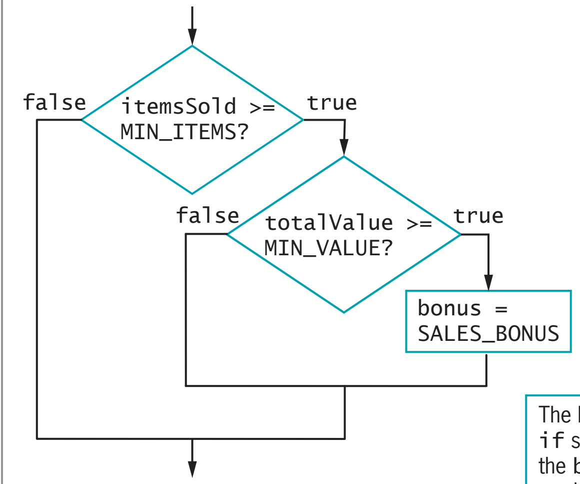
```
C:\Java>java AssignVolunteer2
What type of donation is this?
Enter 1 for clothing, 2 for anything else... 1
You entered 1
This is a clothing donation
The volunteer who will price this item is Regina

C:\Java>java AssignVolunteer2
What type of donation is this?
Enter 1 for clothing, 2 for anything else... 2
You entered 2
This is another donation type
The volunteer who will price this item is Marco

C:\Java>
```

Figure 5-11 Two executions of the `AssignVolunteer2` program

Nesting `if` and `if...else` Statements

Within an `if` or an `else` clause, you can code as many dependent statements as you need, including other `if` and `else` structures. Statements in which an `if` structure is contained inside another `if` structure are commonly called **nested `if` statements**. Nested `if` statements are particularly useful when two conditions must be met before some action is taken.

For example, suppose you want to pay a $50 bonus to a salesperson only if the salesperson sells at least three items that total at least $1,000 in value during a specified time. Figure 5-12 shows the logic and the code to solve the problem.



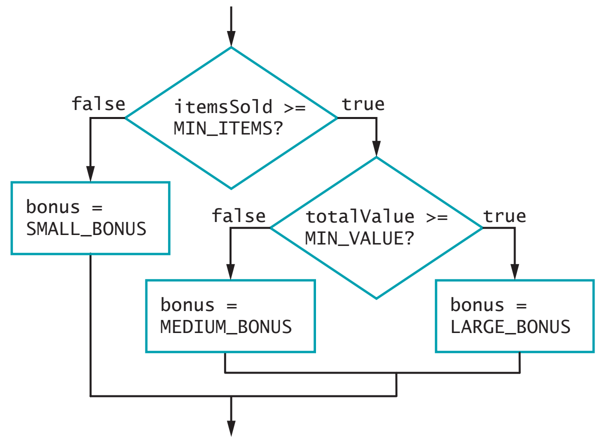
```
final int MIN_ITEMS = 3;
final int MIN_VALUE = 1000;
final int SALES_BONUS = 50;
int bonus = 0;

if(itemsSold >= MIN_ITEMS)
   if(totalValue >= MIN_VALUE)
      bonus = SALES_BONUS;
```

The Boolean expression in each `if` statement must be true for the `bonus` assignment to be made.

Figure 5-12 Determining whether to assign a bonus using nested `if` statements

Notice there are no semicolons in the `if` statement code shown in Figure 5-12 until after the `bonus = SALES_BONUS;` statement. The expression `itemsSold >= MIN_ITEMS` is evaluated first. Only if this expression is `true` does the program evaluate the second Boolean expression, `totalValue >= MIN_VALUE`. If that expression is also `true`, the bonus assignment statement executes, and the `if` structure ends.

When you use nested `if` statements, you must pay careful attention to placement of any `else` clauses. For example, suppose you want to distribute bonuses on a revised schedule. If the salesperson does not sell at least three items, you want to give a $10 bonus. If the salesperson sells at least three items whose combined value is less than $1,000, the bonus is $25. If the salesperson sells at least three items whose combined value is at least $1,000, the bonus is $50. Figure 5-13 shows the logic.



```
final int MIN_ITEMS = 3;
final int MIN_VALUE = 1000;
final int LARGE_BONUS= 50;
final int MEDIUM_BONUS = 25;
final int SMALL_BONUS = 10;

int bonus = 0;

if(itemsSold >= MIN_ITEMS)
   if(totalValue >= MIN_VALUE)
      bonus = LARGE_BONUS;
   else
      bonus = MEDIUM_BONUS;
else
   bonus = SMALL_BONUS;
```

The last `else` goes with the first `if`.

Figure 5-13 Determining one of three bonuses using nested `if` statements

As Figure 5-13 shows, when one `if` statement follows another, the first `else` clause encountered is paired with the most recent `if` encountered. In this figure, the complete nested `if…else` structure fits entirely within the `if` portion of the outer `if…else` statement. No matter how many levels of `if…else` statements are needed to produce a solution, the `else` statements are always associated with their `if`s on a "first in-last out" basis. In Figure 5-13, the indentation of the lines of code helps to show which `else` statement is paired with which `if` statement. Remember, the compiler does not take indentation into account, but consistent indentation can help readers understand a program's logic.

TWO TRUTHS & A LIE

Nesting `if` and `if…else` Statements

1. Statements in which an `if` structure is contained inside another `if` structure commonly are called nested `if` statements.
2. When one `if` statement follows another, the first `else` clause encountered is paired with the first `if` that occurred before it.
3. A complete nested `if…else` structure always fits entirely within either the `if` portion or the `else` portion of its outer `if…else` statement.

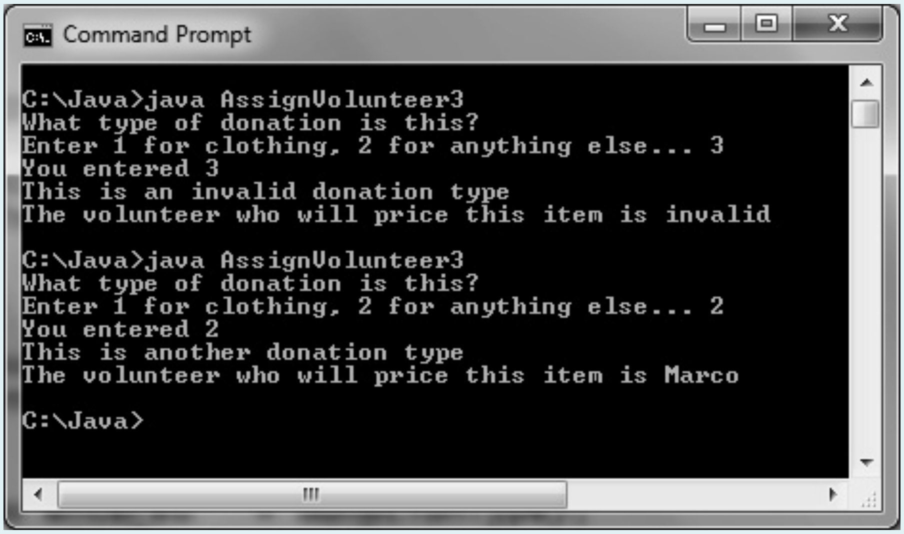
The false statement is #2. When one `if` statement follows another, the first `else` clause encountered is paired with the most recent `if` encountered.

You Do It

Using a Nested `if` Statement

In this section, you add a nested `if` statement to the `AssignVolunteer2` application.

1. Rerun the `AssignVolunteer2` program, and enter an invalid code, such as *3*. The selected volunteer is Marco because the program tests only for an entered value of *1* or not *1*. Modify the program to display the entered code, volunteer, and donation type message only when the entered value is *1* or *2*, and to display the entered code and an error message otherwise. Rename the class **`AssignVolunteer3`**, and save the file as **AssignVolunteer3.java**. Figure 5-14 shows two typical executions of the program.



```
C:\Java>java AssignVolunteer3
What type of donation is this?
Enter 1 for clothing, 2 for anything else... 3
You entered 3
This is an invalid donation type
The volunteer who will price this item is invalid

C:\Java>java AssignVolunteer3
What type of donation is this?
Enter 1 for clothing, 2 for anything else... 2
You entered 2
This is another donation type
The volunteer who will price this item is Marco

C:\Java>
```

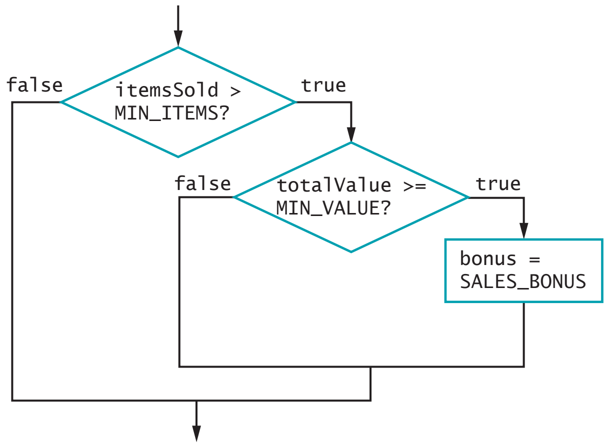
Figure 5-14 Two executions of the `AssignVolunteer3` program

Using Logical AND and OR Operators

In Java, you can combine two Boolean tests into a single expression using the logical AND and OR operators.

The AND Operator

For an alternative to some nested `if` statements, you can use the **logical AND operator** between two Boolean expressions to determine whether both are `true`. In Java, the AND operator is written as two ampersands (&&). For example, the two statements shown in Figure 5-15 work exactly the same way. In each case, both the `itemsSold` variable must be at least the minimum number of items required for a bonus and the `totalValue` variable must be at least the minimum required value for the bonus to be set to `SALES_BONUS`.



```
if(itemsSold > MIN_ITEMS)
   if(totalValue >= MIN_VALUE)
      bonus = SALES_BONUS;
```

```
if(itemsSold > MIN_ITEMS && totalValue >= MIN_VALUE)
   bonus = SALES_BONUS;
```

Figure 5-15 Code for bonus-determining decision using nested `if`s and using the && operator

It is important to note that when you use the && operator, you must include a complete Boolean expression on each side. In other words, like many arithmetic operators, the && operator is a binary operator, meaning it requires an operand on each side. If you want to set a bonus to $400 when a `saleAmount` is both over $1,000 and under $5,000, the correct statement is:

```
if(saleAmount > 1000 && saleAmount < 5000)
   bonus = 400;
```

Even though the `saleAmount` variable is intended to be used in both parts of the AND expression, the following statement is incorrect and does not compile because there is not a complete expression on both sides of the binary && operator:

```
if(saleAmount > 1000 && < 5000)
   bonus = 400;
```

Don't Do It
This statement will not compile because it does not have a Boolean expression on each side of the && operator.

For clarity, many programmers prefer to surround each Boolean expression that is part of a compound Boolean expression with its own set of parentheses, as in the following example:

```
if((saleAmount > 1000) && (saleAmount < 5000))
   bonus = 400;
```

Use the extra parentheses if doing so makes the compound expression clearer to you.

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

The OR Operator

When you want some action to occur even if only one of two conditions is `true`, you can use nested `if` statements, or you can use the **logical OR operator**, which is written as `||`.

For example, if you want to give a discount to any customer who satisfies at least one of two conditions—buying a minimum number of items or buying any number of items that total a minimum value—you can write the code using either of the ways shown in Figure 5-16.

The two vertical lines used in the OR operator are sometimes called "pipes." The pipe appears on the same key as the backslash on your keyboard.

```
if(itemsBought >= MIN_ITEMS)
   discountRate = DISCOUNT;
else
    if(itemsValue >= MIN_VALUE)
       discountRate = DISCOUNT;

if(itemsBought >= MIN_ITEMS || itemsValue >= MIN_VALUE)
   discountRate = DISCOUNT;
```

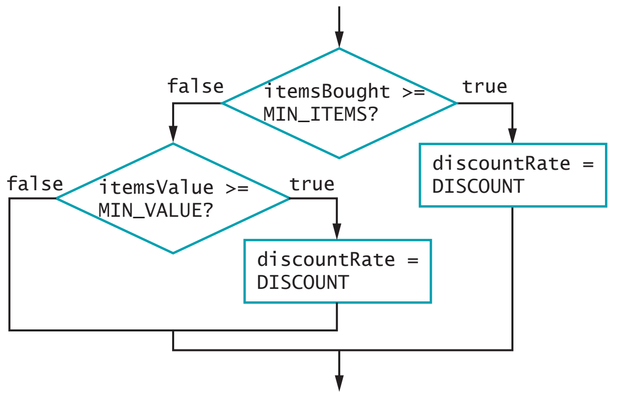


Figure 5-16 Determining customer discount when customer needs to meet only one of two criteria

As with the && operator, you are never required to use the `||` operator because using nested `if` statements always achieves the same result. However, using the `||` operator often makes your code more concise, less error-prone, and easier to understand.

Short-Circuit Evaluation

The expressions on each side of the && and || operators are evaluated only as far as necessary to determine whether the entire expression is `true` or `false`. This feature is called **short-circuit evaluation**. With the && operator, both Boolean expression operands must be `true` before the action in the result statement can occur. (The same is true for nested `if`s, as you can see in Figure 5-15.) When you use the && operator, if the first tested expression is `false`, the second expression is never evaluated because its value does not matter.

The || operator also uses short-circuit evaluation. In other words, because only one of the Boolean expressions in an || expression must be true to cause the dependent statements to execute, if the expression to the left of the || is true, then there is no need to evaluate the expression to the right of the ||. (The same is true for nested `if`s, as you can see in Figure 5-16.) When you use the || operator, if the first tested expression is `true`, the second expression is never evaluated because its value does not matter.

If you are using simple comparisons as the operands for the && or || operators, as in the examples in Figures 5-15 and 5-16, you won't notice that short-circuit evaluation is occurring. However, suppose that you have created two methods that return Boolean values and you use calls to those methods in an `if` statement, as in the following:

```
if(method1() && method2())
   System.out.println("OK");
```

Depending on the actions performed within the methods, it might be important to understand that in this case, if `method1()` is false, then `method2()` will not execute.

Watch the video *Using && and ||*.

TWO TRUTHS & A LIE

Using Logical AND and OR Operators

1. The AND operator is written as two ampersands (&&), and the OR operator is written as two pipes (||).
2. When you use the && and || operators, you must include a complete Boolean expression on each side.
3. When you use an && or || operator, each Boolean expression that surrounds the operator is always tested in order from left to right.

The false statement is #3. The expressions in each part of an AND or OR expression are evaluated only as much as necessary to determine whether the entire expression is `true` or `false`. For example, in an AND expression, if the first Boolean value is false, the second expression is not tested. In an OR expression, if the first Boolean value is true, the second expression is not tested. This feature is called short-circuit evaluation.

You Do It

Using the && Operator

This section helps you create a program that demonstrates how short-circuiting works with the `&&` operator.

1. Open a new file in your text editor, and type the header and curly braces for a class named `ShortCircuitTestAnd`:

```
public class ShortCircuitTestAnd
{
}
```

2. Between the curly braces for the class, type the header and braces for a `main()` method:

```
public static void main(String[] args)
{
}
```

3. Within the `main()` method, insert an `if…else` statement that tests the return values of two method calls. If both methods are `true`, then "Both are true" is displayed. Otherwise, "Both are not true" is displayed.

```
if(trueMethod() && falseMethod())
   System.out.println("Both are true");
else
   System.out.println("Both are not true");
```

4. Following the closing curly brace for the `main()` method, but before the closing curly brace for the class, insert a method named `trueMethod()`. The method displays the message "True method" and returns a `true` value.

```
public static boolean trueMethod()
{
   System.out.println("True method");
   return true;
}
```

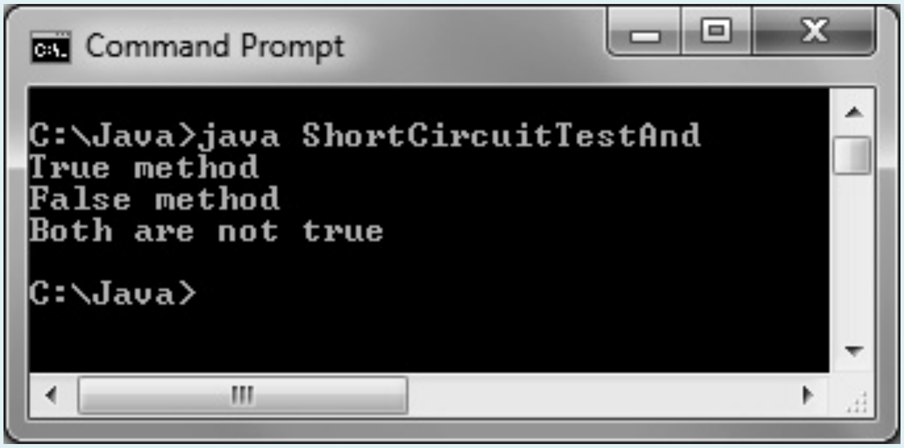
5. Following the closing curly brace of `trueMethod()`, insert a method named `falseMethod()` that displays the message "False method" and returns a `false` value.

```
public static boolean falseMethod()
{
   System.out.println("False method");
   return false;
}
```

(continues)

(continued)

6. Save the file as **ShortCircuitTestAnd.java**, and then compile and execute it. Figure 5-17 shows the output. First, "True method" is displayed because `trueMethod()`was executed as the first half of the Boolean expression in the program's `if` statement. Then, the second half of the Boolean expression calls `falseMethod()`. Finally, "Both are not true" is displayed because both halves of the tested expression were not true.



```
C:\Java>java ShortCircuitTestAnd
True method
False method
Both are not true

C:\Java>
```

Figure 5-17 Execution of `ShortCircuitTestAnd` program

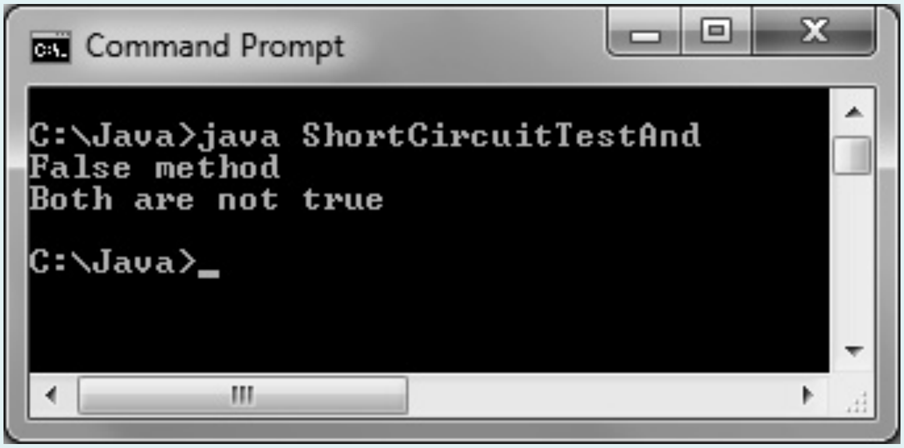
7. Change the position of the method calls in the `if` statement so that the statement becomes the following:

```
if(falseMethod() && trueMethod())
   System.out.println("Both are true");
else
   System.out.println("Both are not true");
```

8. Save the file, compile it, and execute it. Now the output looks like Figure 5-18. The `if` statement makes a call to `falseMethod()`, and its output is displayed. Because the first half of the Boolean expression is false, there is no need to test the second half, so `trueMethod()` never executes, and the program proceeds directly to the statement that displays "Both are not true."

(continues)

(continued)



```
C:\Java>java ShortCircuitTestAnd
False method
Both are not true

C:\Java>_
```

Figure 5-18 Output of `ShortCircuitTestAnd` after reversing Boolean expressions

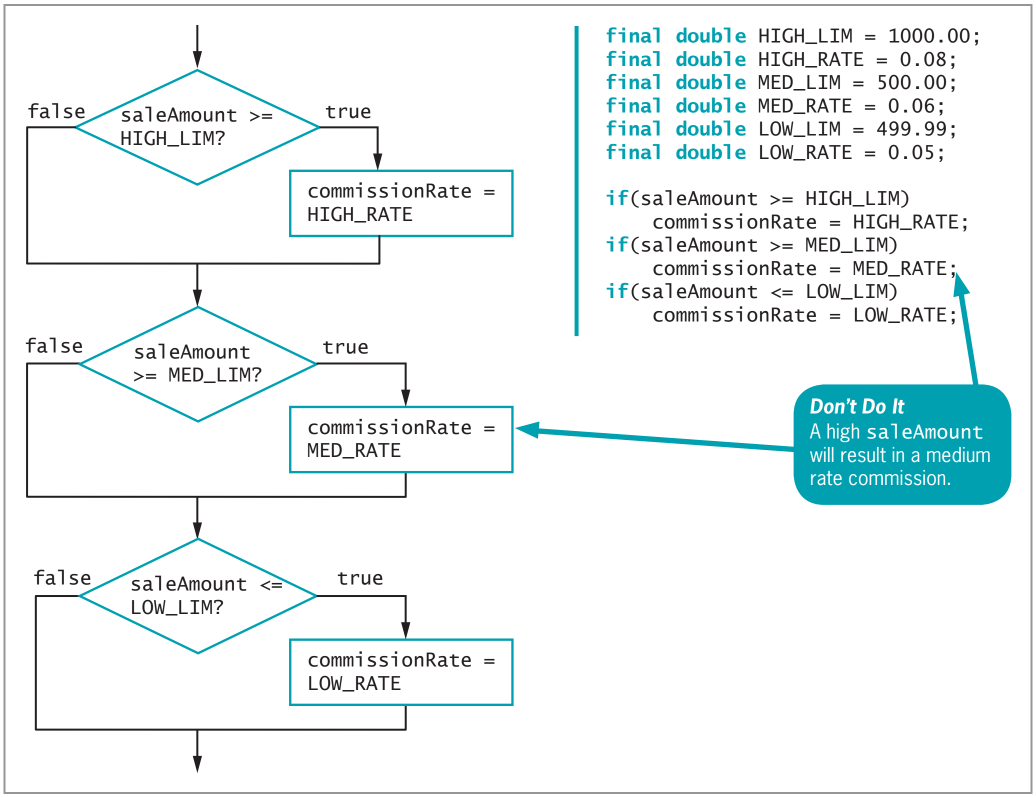
9. Change the class name to **`ShortCircuitTestOr`**, and immediately save the file as **ShortCircuitTestOr.java**. Replace the `&&` operator with the `||` operator. Compile and execute the program with `trueMethod()`to the right of the `||` operator and `falseMethod()`to its left. Then, reverse the positions of the methods, and compile and execute the program again. Make sure that you understand the output each way.

Making Accurate and Efficient Decisions

When new programmers must make a range check, they often introduce incorrect or inefficient code into their programs. In this section, you learn how to make accurate and efficient range checks, and you also learn how to use the `&&` and `||` operators appropriately.

Making Accurate Range Checks

A **range check** is a series of statements that determine to which of several consecutive series of values another value falls. Consider a situation in which salespeople can receive one of three possible commission rates based on their sales. For example, a sale totaling $1,000 or more earns the salesperson an 8% commission, a sale totaling $500 through $999 earns 6% of the sale amount, and any sale totaling $499.99 or less earns 5%. Using three separate `if` statements to test single Boolean expressions might result in some incorrect commission assignments. For example, examine the code shown in Figure 5-19.

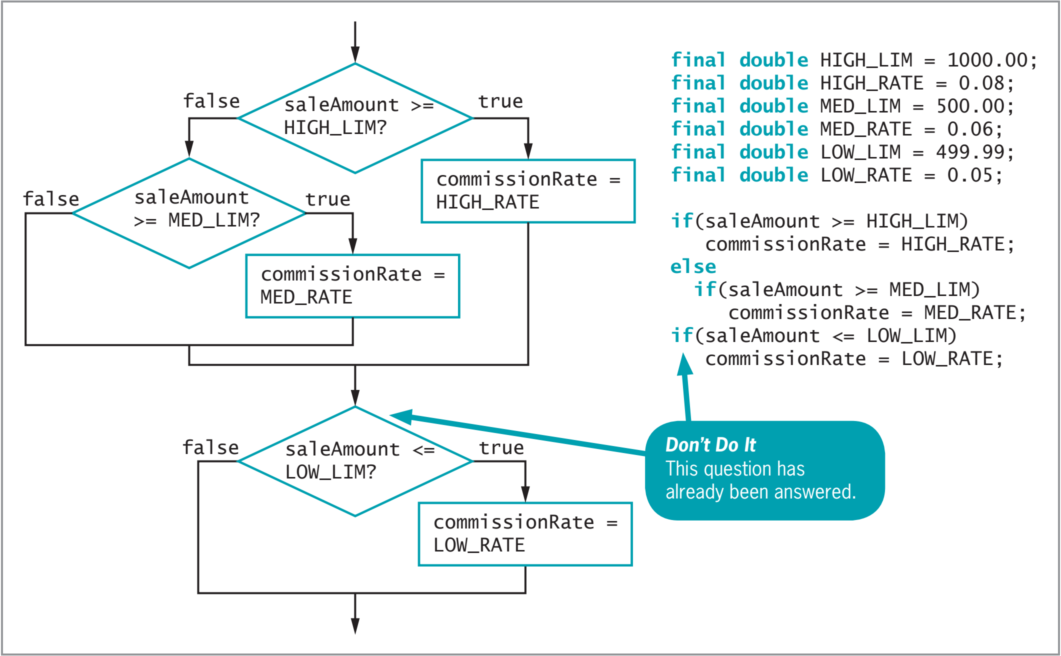

false
saleAmount >= HIGH_LIM?
true
commissionRate = HIGH_RATE
false
saleAmount >= MED_LIM?
true
commissionRate = MED_RATE
false
saleAmount <= LOW_LIM?
true
commissionRate = LOW_RATE
Don't Do It
A high saleAmount will result in a medium rate commission.

```
final double HIGH_LIM = 1000.00;
final double HIGH_RATE = 0.08;
final double MED_LIM = 500.00;
final double MED_RATE = 0.06;
final double LOW_LIM = 499.99;
final double LOW_RATE = 0.05;

if(saleAmount >= HIGH_LIM)
    commissionRate = HIGH_RATE;
if(saleAmount >= MED_LIM)
    commissionRate = MED_RATE;
if(saleAmount <= LOW_LIM)
    commissionRate = LOW_RATE;
```

Figure 5-19 Incorrect commission-determining code

Using the code shown in Figure 5-19, when a `saleAmount` is $5,000, for example, the first `if` statement executes, and the Boolean expression `(saleAmount >= HIGH_LIM)` evaluates as `true`, so `HIGH_RATE` is correctly assigned to `commissionRate`. However, the next `if` expression, `(saleAmount >= MED_LIM)`, also evaluates as `true`, so the `commissionRate`, which was just set to `HIGH_RATE`, is incorrectly reset to `MED_RATE`.

A partial solution to this problem is to use an `else` statement following the first evaluation, as shown in Figure 5-20.


false
saleAmount >= HIGH_LIM?
true
commissionRate = HIGH_RATE
false
saleAmount >= MED_LIM?
true
commissionRate = MED_RATE
false
saleAmount <= LOW_LIM?
true
commissionRate = LOW_RATE
Don't Do It
This question has already been answered.

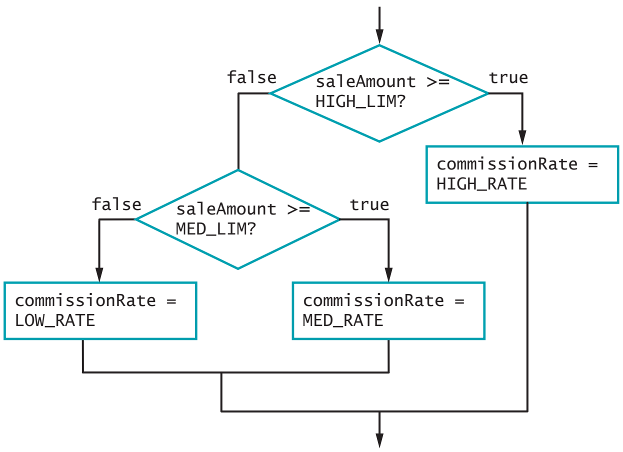
```
final double HIGH_LIM = 1000.00;
final double HIGH_RATE = 0.08;
final double MED_LIM = 500.00;
final double MED_RATE = 0.06;
final double LOW_LIM = 499.99;
final double LOW_RATE = 0.05;

if(saleAmount >= HIGH_LIM)
   commissionRate = HIGH_RATE;
else
   if(saleAmount >= MED_LIM)
      commissionRate = MED_RATE;
if(saleAmount <= LOW_LIM)
   commissionRate = LOW_RATE;
```

Figure 5-20 Improved, but inefficient, commission-determining code

With the new code in Figure 5-20, when the `saleAmount` is $5,000, the expression `(saleAmount >= HIGH_LIM)` is `true` and the `commissionRate` becomes `HIGH_RATE`; then the entire `if` structure ends. When the `saleAmount` is not greater than or equal to $1,000 (for example, $800), the first `if` expression is `false`, and the `else` statement executes and correctly sets the `commissionRate` to `MED_RATE`.

The code shown in Figure 5-20 works, but it is somewhat inefficient. When the `saleAmount` is any amount over `LOW_RATE`, either the first `if` sets `commissionRate` to `HIGH_RATE` for amounts that are at least $1,000, or its `else` sets `commissionRate` to `MED_RATE` for amounts that are at least $500. In either of these two cases, the Boolean value tested in the next statement, `if(saleAmount <= LOW_LIM)`, is always `false`, so `commissionRate` retains its correct value. However, it was unnecessary to ask the `LOW_LIM` question.

After you know that `saleAmount` is not at least `MED_LIM`, rather than asking `if(saleAmount <= LOW_LIM)`, it's easier, more efficient, and less error-prone to use an `else`. If the `saleAmount` is not at least `HIGH_LIM` and is also not at least `MED_LIM`, it must by default be less than or equal to `LOW_LIM`. Figure 5-21 shows this improved logic. Notice that the `LOW_LIM` constant is no longer declared because it is not needed anymore—if a `saleAmount` is not greater than or equal to `MED_LIMIT`, the `commissionRate` must receive the `LOW_RATE`.

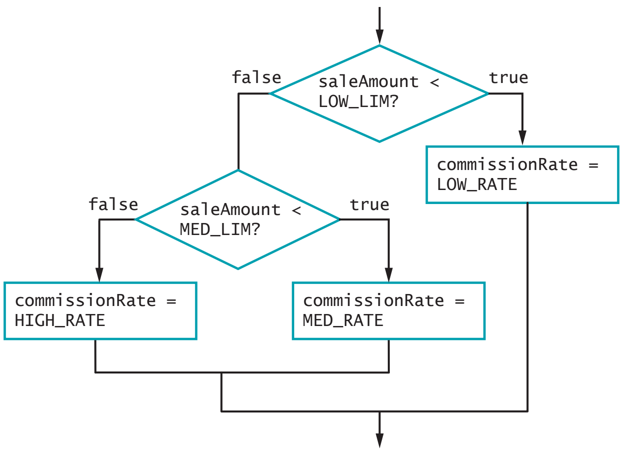

saleAmount >= HIGH_LIM?
false
true
commissionRate = HIGH_RATE
saleAmount >= MED_LIM?
false
true
commissionRate = LOW_RATE
commissionRate = MED_RATE

```
final double HIGH_LIM = 1000.00;
final double HIGH_RATE = 0.08;
final double MED_LIM = 500.00;
final double MED_RATE = 0.06;
final double LOW_RATE = 0.05;

if(saleAmount >= HIGH_LIM)
   commissionRate = HIGH_RATE;
else
   if(saleAmount >= MED_LIM)
      commissionRate = MED_RATE;
   else
      commissionRate = LOW_RATE;
```

Figure 5-21 Improved and efficient commission-determining logic

Making Efficient Range Checks

Within a nested `if…else`, like the one shown in Figure 5-21, it is most efficient to ask the question that is most likely to be true first. In other words, if you know that most `saleAmount` values are high, compare `saleAmount` to `HIGH_LIM` first. That way, you most frequently avoid asking multiple questions. If, however, you know that most `saleAmounts` are small, you should ask `if(saleAmount < LOW_LIM)` first. The code shown in Figure 5-22 results in the same commission value for any given `saleAmount`, but this sequence of decisions is more efficient when most `saleAmount` values are small.


saleAmount < LOW_LIM?
false
true
commissionRate = LOW_RATE
saleAmount < MED_LIM?
false
true
commissionRate = HIGH_RATE
commissionRate = MED_RATE

```
final double HIGH_RATE = 0.08;
final double MED_LIM = 1000.00;
final double MED_RATE = 0.06;
final double LOW_LIM = 500.00;
final double LOW_RATE = 0.05;

if(saleAmount < LOW_LIM)
   commissionRate = LOW_RATE;
else
   if(saleAmount < MED_LIM)
      commissionRate = MED_RATE;
   else
      commissionRate = HIGH_RATE;
```

Figure 5-22 Commission-determining code asking about smallest `saleAmount` first

In Figure 5-22, notice that the comparisons use the < operator instead of <=. That's because a `saleAmount` of $1,000.00 should result in a `HIGH_RATE`, and a `saleAmount` of $500.00 should result in a `MED_RATE`. If you wanted to use <= comparisons, then you could change the `MED_LIM` and `LOW_LIM` cutoff values to 999.99 and 499.99, respectively.

Using && and || Appropriately

Beginning programmers often use the && operator when they mean to use ||, and often use || when they should use &&. Part of the problem lies in the way we use the English language. For example, your boss might request, "Display an error message when an employee's hourly pay rate is under $5.85 and when an employee's hourly pay rate is over $60." You define $5.85 as a named constant `LOW` and $60 as `HIGH`. However, because your boss used the word *and* in the request, you might be tempted to write a program statement like the following:

```
if(payRate < LOW && payRate > HIGH)
   System.out.println("Error in pay rate");
```

This message can never be output because the Boolean expression can never be true.

However, as a single variable, no `payRate` value can ever be both below 5.85 *and* over 60 at the same time, so the output statement can never execute, no matter what value the `payRate` has. In this case, you must write the following code that uses the || operator to display the error message under the correct circumstances:

```
if(payRate < LOW || payRate > HIGH)
   System.out.println("Error in pay rate");
```

Similarly, your boss might request, "Display the names of those employees in departments 1 and 2." Because the boss used the word *and* in the request, you might be tempted to write the following:

```
if(department == 1 && department == 2)
   System.out.println("Name is: " + name);
```

However, the variable `department` can never contain both a *1* and a *2* at the same time, so no employee name will ever be output, no matter what the value of `department` is. The correct statement chooses employees whose `department` is 1 or 2, as follows:

```
if(department == 1 || department == 2)
   System.out.println("Name is: " + name);
```

Another type of mistake occurs if you use a single ampersand or pipe when you try to indicate a logical AND or OR. Both & and | are valid Java operators, but they have two different functions. When you use a single & or | with integer operands, it operates on bits. When you use a single & or | with Boolean expressions, it always evaluates both expressions instead of using short-circuitry.

TWO TRUTHS & A LIE

Making Accurate and Efficient Decisions

1. A range check is a series of statements that determine within which of a set of ranges a value falls.
2. When you must make a series of decisions in a program, it is most efficient to first ask the question that is most likely to be true.
3. The statement `if(payRate < 6.00 && payRate > 50.00)` can be used to select `payRate` values that are higher or lower than the specified limits.

The false statement is #3. The statement `if(payRate < 6.00 && payRate > 50.00)` cannot be used to make a selection because no value for `payRate` can be both below 6.00 *and* above 50.00 at the same time.

Using the `switch` Statement

By nesting a series of `if` and `else` statements, you can choose from any number of alternatives. For example, suppose you want to display a student's class year based on a stored number. Figure 5-23 shows one possible implementation of the program.

```
if(year == 1)
   System.out.println("Freshman");
else
   if(year == 2)
      System.out.println("Sophomore");
   else
      if(year == 3)
         System.out.println("Junior");
      else
         if(year == 4)
            System.out.println("Senior");
         else
            System.out.println("Invalid year");
```

Figure 5-23 Determining class status using nested `if` statements

In program segments like the one in Figure 5-23, many programmers (particularly those familiar with the Visual Basic programming language) would code each `else` and the `if` clause that follows it on the same line, and refer to the format as an **`else`…`if` clause**. Because Java ignores whitespace, the logic is the same whether each `else` and the subsequent `if` are on the same line or different lines.

An alternative to using the series of nested if statements shown in Figure 5-23 is to use the switch statement. The **switch statement** is useful when you need to test a single variable against a series of exact integer (including int, byte, and short types), character, or string values. The ability to use strings as the tested values in a switch statement is a new feature in Java 7.

The switch statement uses four keywords:

- switch starts the structure and is followed immediately by a test expression enclosed in parentheses.
- case is followed by one of the possible values for the test expression and a colon.
- break optionally terminates a switch statement at the end of each case.
- default optionally is used prior to any action that should occur if the test variable does not match any case.

Figure 5-24 shows the switch statement used to display the four school years based on an integer named year.

```
switch(year)
{
   case 1:
      System.out.println("Freshman");
      break;
   case 2:
      System.out.println("Sophomore");
      break;
   case 3:
      System.out.println("Junior");
      break;
   case 4:
      System.out.println("Senior");
      break;
   default:
      System.out.println("Invalid year");
}
```

Figure 5-24 Determining class status using a switch statement

You are not required to list the case values in ascending order, as shown in Figure 5-24, although doing so often makes a statement easier to understand. For efficiency, you might want to list the most likely case first.

The switch statement shown in Figure 5-24 begins by evaluating the year variable shown in the first line. If year is equal to the first case value, which is 1, the statement that displays “Freshman” executes. The break statement bypasses the rest of the switch structure, and execution continues with any statement after the closing curly brace of the switch structure.

If the `year` variable is not equivalent to the first `case` value of 1, the next `case` value is compared, and so on. If the `year` variable does not contain the same value as any of the `case` statements, the `default` statement or statements execute.

You can leave out the `break` statements in a `switch` structure. However, if you omit the `break` and the program finds a match for the test variable, all the statements within the `switch` statement execute from that point forward. For example, if you omit each `break` statement in the code shown in Figure 5-24, when the year is 3, the first two cases are bypassed, but *Junior*, *Senior*, and *Invalid year* all are output. You should intentionally omit the `break` statements if you want all subsequent cases to execute after the test variable is matched.

You do not need to write code for each case in a `switch` statement. For example, suppose that the supervisor for departments 1, 2, and 3 is *Jones*, but other departments have different supervisors. In that case, you might use the code in Figure 5-25.

```
int department;
String supervisor;
// Statements to get department go here
switch(department)
{
   case 1:
   case 2:
   case 3:
      supervisor = "Jones";
      break;
   case 4:
      supervisor = "Staples";
      break;
   case 5:
      supervisor = "Tejano";
      break:
   default:
      System.out.println("Invalid department code");
}
```

Figure 5-25 Using empty `case` statements so the same result occurs in multiple cases

On the other hand, you might use strings in a `switch` structure to determine whether a supervisor name is valid, as shown in the method in Figure 5-26.

```
public static boolean isValidSupervisor(String name)
{
   boolean isValid;
   switch(name)
   {
      case "Jones":
      case "Staples":
      case "Tejano":
         isValid = true;
      break;
      default:
         isValid = false;
   }
   return isValid;
}
```

Figure 5-26 A method that uses a `switch` structure with string values

When several `char` variables must be checked and you want to ignore whether they are uppercase or lowercase, one frequently used technique employs empty `case` statements, as in Figure 5-27.

```
switch(departmentCode)
{
   case 'a':
   case 'A':
      departmentName = "Accounting";
      break;
   case 'm':
   case 'M':
      departmentName = "Marketing";
      break;
   // and so on
}
```

Figure 5-27 Using a `switch` structure to ignore character case

You are never required to use a `switch` structure; you can always achieve the same results with nested `if` statements. The `switch` structure is simply convenient to use when there are several alternative courses of action that depend on a single integer, character, or string value. In addition, it makes sense to use `switch` only when a reasonable number of specific matching values need to be tested.

Watch the video *Using the `switch` Statement.*

TWO TRUTHS & A LIE

Using the `switch` Statement

1. When you must make more decisions than Java can support, you use a `switch` statement instead of nested `if…else` statements.
2. The `switch` statement is useful when you need to test a single variable against a series of exact integer or character values.
3. A `break` statement bypasses the rest of its `switch` structure, and execution continues with any statement after the closing curly brace of the `switch` structure.

The false statement is #1. By nesting a series of `if` and `else` statements, you can choose from any number of alternatives. The `switch` statement is just a more convenient way of expressing nested `if…else` statements when the tested value is an integer or character.

You Do It

Using the `switch` Statement

In this section, you alter the `AssignVolunteer3` program to add more options for donation types, and then use a `switch` statement to assign the appropriate volunteer.

1. Open the **AssignVolunteer3.java** file that you created in a "You Do It" section earlier in this chapter. Change the class name to **`AssignVolunteer4`**, and immediately save the file as **AssignVolunteer4.java**.
2. Keep the declaration `CLOTHING_CODE`, but replace the `OTHER_CODE` declaration with three new ones:

```
final int FURNITURE_CODE = 2;
final int ELECTRONICS_CODE = 3;
final int OTHER_CODE = 4;
```

3. Retain the two pricing volunteer declarations, but add two new ones:

```
final String FURNITURE_PRICER = "Walter";
final String ELECTRONICS_PRICER = "Lydia";
```

(continues)

(continued)

4. Replace the output statement that asks the user to enter 1 or 2 with the following simpler statement:

```
System.out.print("Enter an integer... ");
```

   In a professional program, you might want to present the user with details about all the options, but this example keeps the prompt simple to save you from excessive typing.

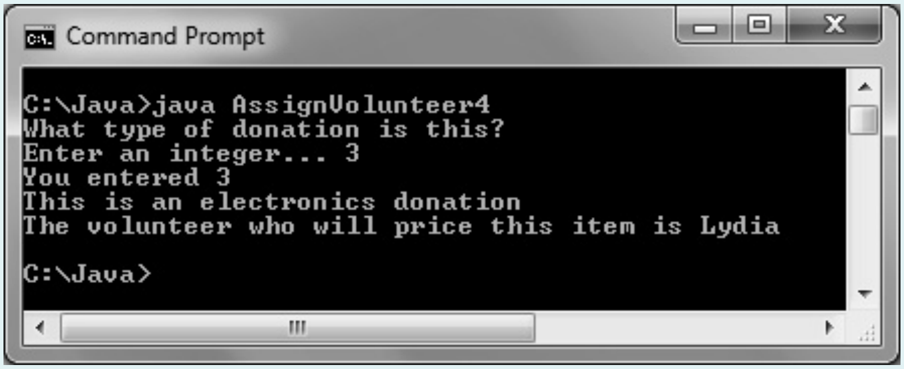
5. Replace the existing `if...else` structure with the following `switch` structure:

```
switch(donationType)
{
    case(CLOTHING_CODE):
        volunteer = CLOTHING_PRICER;
        message = "a clothing donation";
        break;
    case(FURNITURE_CODE):
        volunteer = FURNITURE_PRICER;
        message = "a furniture donation";
        break;
    case(ELECTRONICS_CODE):
        volunteer = ELECTRONICS_PRICER;
        message = "an electronics donation";
        break;
    case(OTHER_CODE):
        volunteer = OTHER_PRICER;
        message = "another donation type";
        break;
    default:
        volunteer = "invalid";
        message = "an invalid donation type";
}
```

6. Save the file, and then compile and execute it. Figure 5-28 shows a typical execution.

(continues)

(continued)



```
C:\Java>java AssignVolunteer4
What type of donation is this?
Enter an integer... 3
You entered 3
This is an electronics donation
The volunteer who will price this item is Lydia

C:\Java>
```

Figure 5-28 Typical execution of the `AssignVolunteer4` program

Using the Conditional and NOT Operators

Besides using `if` statements and `switch` structures, Java provides one more way to make decisions. The **conditional operator** requires three expressions separated with a question mark and a colon and is used as an abbreviated version of the `if…else` structure. As with the `switch` structure, you are never required to use the conditional operator; it is simply a convenient shortcut. The syntax of the conditional operator is:

```
testExpression ? trueResult : falseResult;
```

The first expression, `testExpression`, is a Boolean expression that is evaluated as `true` or `false`. If it is `true`, the entire conditional expression takes on the value of the expression following the question mark (`trueResult`). If the value of the `testExpression` is `false`, the entire expression takes on the value of `falseResult`.

You have seen many examples of binary operators such as == and &&. The conditional operator is a **ternary operator**—one that needs three operands. Through Java 6, the conditional operator is the only ternary operator in Java, so it is sometimes referred to as “the” ternary operator. Java 7 introduces a collapsed version of the ternary operator that checks for `null` values assigned to objects. The new operator is called *the Elvis operator* because it uses a question mark and colon together (?:); if you view it sideways, it reminds you of Elvis Presley.

For example, suppose you want to assign the smallest price to a sale item. Let the variable `a` be the advertised price and the variable `b` be the discounted price on the sale tag. The expression for assigning the smallest cost is:

```
smallerNum = (a < b) ? a : b;
```

When evaluating the expression `a < b`, where `a` is less than `b`, the entire conditional expression takes the value to the left of the colon, `a`, which then is assigned to `smallerNum`. If `a` is not less than `b`, the expression assumes the value to the right of the colon, `b`, and `b` is assigned to `smallerNum`.

You could achieve the same results with the following `if`…`else` structure:

```
if(a < b)
   smallerNum = a;
else
   smallerNum = b;
```

The advantage of using the conditional operator is the conciseness of the statement.

Using the NOT Operator

You use the **NOT operator**, which is written as the exclamation point (`!`), to negate the result of any Boolean expression. Any expression that evaluates as `true` becomes `false` when preceded by the NOT operator, and accordingly, any `false` expression preceded by the NOT operator becomes `true`.

For example, suppose a monthly car insurance premium is $200 if the driver is age 25 or younger and $125 if the driver is age 26 or older. Each of the `if`…`else` statements in Figure 5-29 correctly assigns the premium values.

```
if(age <= 25)
   premium = 200;
else
   premium = 125;
```

```
if(!(age <= 25))
   premium = 125;
else
   premium = 200;
```

```
if(age >= 26)
   premium = 125;
else
   premium = 200;
```

```
if(!(age >= 26))
   premium = 200;
else
   premium = 125;
```

Figure 5-29 Four `if`…`else` statements that all do the same thing

In Figure 5-29, the statements with the ! operator are somewhat harder to read, particularly because they require the double set of parentheses, but the result of the decision-making process is the same in each case. Using the ! operator is clearer when the value of a Boolean variable is tested. For example, a variable initialized as `boolean oldEnough = (age >= 25);` can become part of the relatively easy-to-read expression `if(!oldEnough)`….

TWO TRUTHS & A LIE

Using the Conditional and NOT Operators

1. The conditional operator is used as an abbreviated version of the `if…else` structure and requires two expressions separated with an exclamation point.
2. The NOT operator is written as the exclamation point (`!`).
3. The value of any false expression becomes true when preceded by the NOT operator.

The false statement is #1. The conditional operator requires three expressions separated with a question mark and a colon.

Understanding Operator Precedence

You can combine as many `&&` or `||` operators as you need to make a decision. For example, if you want to award bonus points (defined as `BONUS`) to any student who receives a perfect score on any of four quizzes, you might write a statement like the following:

```
if(score1 == PERFECT || score2 == PERFECT ||
   score3 == PERFECT || score4 == PERFECT)
      bonus = BONUS;
else
   bonus = 0;
```

In this case, if at least one of the score variables is equal to the `PERFECT` constant, the student receives the bonus points.

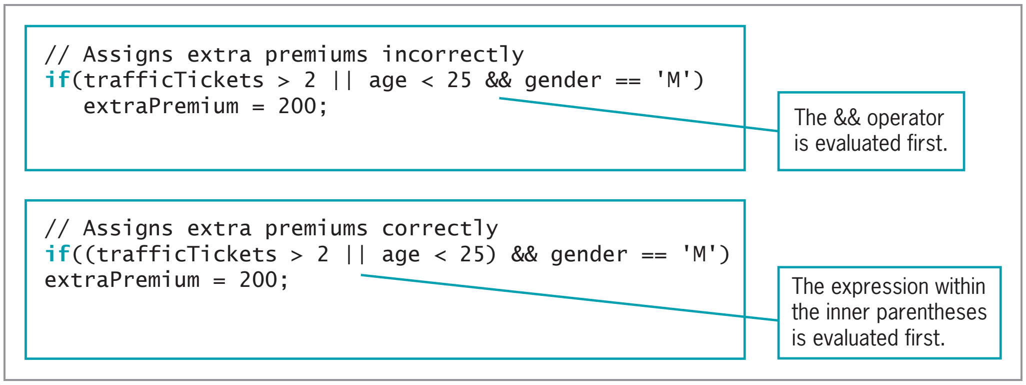
Although you can combine any number of `&&` or `||` operations in an expression, special care must be taken when you mix them. You learned in Chapter 2 that arithmetic operations have higher and lower precedences, and an operator's precedence makes a difference in how an expression is evaluated. For example, within an arithmetic expression, multiplication and division are always performed prior to addition or subtraction. Table 5-1 shows the precedence of the operators you have used so far.

Precedence	Operator(s)	Symbol(s)
Highest	Logical NOT	!
Intermediate	Multiplication, division, modulus	* / %
	Addition, subtraction	+ -
	Relational	> < >= <=
	Equality	== !=
	Logical AND	&&
	Logical OR	||
	Conditional	?:
Lowest	Assignment	=

Table 5-1 Operator precedence for operators used so far

In general, the order of precedence agrees with common algebraic usage. For example, in any mathematical expression, such as `x = a + b`, the arithmetic is done first and the assignment is done last, as you would expect. The relationship of && and || might not be as obvious. The && operator has higher precedence than the || operator. For example, consider the program segments shown in Figure 5-30. These code segments are intended to be part of an insurance company program that determines whether an additional premium should be charged to a driver who meets both of the following criteria:

- Has more than two traffic tickets or is under 25 years old
- Is male



```
// Assigns extra premiums incorrectly
if(trafficTickets > 2 || age < 25 && gender == 'M')
   extraPremium = 200;
```

```
// Assigns extra premiums correctly
if((trafficTickets > 2 || age < 25) && gender == 'M')
extraPremium = 200;
```

The && operator is evaluated first.
The expression within the inner parentheses is evaluated first.

Figure 5-30 Two comparisons using && and ||

One way to remember the precedence of the AND and OR operators is to remember that they are evaluated in alphabetical order.

Consider a 30-year-old female driver with three traffic tickets; according to the stated criteria, she should not be assigned the extra premium because she is not male. With the first `if` statement in Figure 5-30, the && operator takes precedence, so `age < 25 && gender == 'M'` is evaluated first. The value is `false` because `age` is not less than 25, so the expression is reduced to `trafficTickets > 2` or `false`. Because the value of the tickets variable is greater than 2, the entire expression is `true`, and $200 is assigned to `extraPremium`, even though it should not be.

In the second `if` statement shown in Figure 5-30, parentheses have been added so the `||` operator is evaluated first. The expression `trafficTickets > 2 || age < 25` is `true` because the value of `trafficTickets` is 3. So the expression evolves to `true && gender == 'M'`. Because gender is not 'M', the value of the entire expression is `false`, and the `extraPremium` value is not assigned 200, which is correct. Even when an expression would be evaluated as you intend without adding extra parentheses, you can always add them to help others more easily understand your programs.

The following two conventions are important to keep in mind:

- The order in which you use operators makes a difference.
- You can always use parentheses to change precedence or make your intentions clearer.

TWO TRUTHS & A LIE

Understanding Operator Precedence

1. Assume `p`, `q`, and `r` are all Boolean variables that have been assigned the value `true`. After the following statement executes, the value of `p` is still `true`.
   ```
   p = !q || r;
   ```
2. Assume `p`, `q`, and `r` are all Boolean variables that have been assigned the value `true`. After the following statement executes, the value of `p` is still `true`.
   ```
   p = !(!q && !r);
   ```
3. Assume `p`, `q`, and `r` are all Boolean variables that have been assigned the value `true`. After the following statement executes, the value of `p` is still `true`.
   ```
   p = !(q || !r);
   ```

The false statement is #3. If `p`, `q`, and `r` are all Boolean variables that have been assigned the value `true`, then after `p = !(q || !r);` executes, the value of `p` is `false`. First `q` is evaluated as `true`, so the entire expression within the parentheses is `true`. The leading NOT operator reverses that result to `false` and assigns it to `p`.

Adding Decisions and Constructors to Instance Methods

You frequently will want to use what you have learned about decision making to control the allowed values in objects' fields. Whether values are assigned to objects by constructors or by mutator methods, you often will need to use decisions to control values.

For example, suppose that you create an `Employee` class as shown in Figure 5-31. The class contains two fields that hold an employee ID number and pay rate. The constructor accepts values for these fields as parameters, but instead of simply assigning the parameters to the fields, the code determines whether each value is within the allowed limits for the field. Similar logic could be used in any set methods created for the class. Using decisions helps you ensure that fields have acceptable values.

```
public class Employee
{
   private int empNum;
   private double payRate;
   public int MAX_EMP_NUM = 9999;
   public double MAX_RATE = 60.00;
   Employee(int num, double rate)
   {
      if(num <= MAX_EMP_NUM)
         empNum = num;
      else
         empNum = MAX_EMP_NUM;
      if(payRate <= MAX_RATE)
         payRate = rate;
      else
         payRate = 0;
   }
   public int getEmpNum()
   {
      return empNum;
   }
   public double getPayRate()
   {
      return payRate;
   }
}
```

Figure 5-31 The `Employee` class that contains a constructor that makes decisions

You Do It

Adding Decisions to Constructors and Instance Methods

In this section, you modify the `DogTriathlonParticipant` class you created in Chapter 4. Because some points are awarded for participation in each event, a score of 0 is not possible unless a dog did not participate. In the existing class, the constructor accepts the number of events in which a dog participated and the participant's score in each event. Currently, there is no way to check whether these values are in agreement. Now, you can modify the class so that the number of events matches the number of valid scores supplied to the constructor.

1. Open the **DogTriathlonParticipant.java** file that you created in Chapter 4. Change the class name to **`DogTriathlonParticipant2`**, and immediately save the file as **DogTriathlonParticipant2.java**.
2. Change the constructor name to **`DogTriathlonParticipant2`**.
3. If 0 is assigned to the number of events in the existing program, computing the average score produces a nonnumeric result. Now that you know how to use decisions, you can fix this problem. In place of the arithmetic statement that produces the average score using division, use the following `if…else` structure:

```
if(NUM_EVENTS == 0)
   avg = 0;
else
   avg = (double) total / NUM_EVENTS;
```

4. Add a Boolean field to the list of class fields. This field holds `true` if the number of events reported matches the number of nonzero scores. Otherwise, the field holds `false`:

```
private boolean scoresAgree;
```

5. There are several ways to ensure that the number of events passed to the constructor matches the number of nonzero scores passed. One way is to add 1 to a total for each nonzero score and then determine whether that total equals the passed number of events. To accomplish this, first add the following code to the constructor immediately after the statements that assign values to the name and number of events. These statements declare a variable that holds the number of nonzero scores passed to the constructor, and then add 1 to the variable for each nonzero event score:

(continues)

(continued)

```
int totalNot0 = 0;
if(score1 != 0)
   totalNot0 = totalNot0 + 1;
if(score2 != 0)
   totalNot0 = totalNot0 + 1;
if(score3 != 0)
   totalNot0 = totalNot0 + 1;
```

6. Compare the number of events to the total of nonzero scores, and set the Boolean variable scores Agree:

```
if(numEvents == totalNot0)
   scoresAgree = true;
else
   scoresAgree = false;
```

7. Replace the statements that unconditionally assigned values to `obedienceScore`, `conformationScore`, and `agilityScore` with the following `if…else` structure, which assigns the constructor's parameters to the three scores only when `scoresAgree` is `true`.

```
if(scoresAgree)
{
   obedienceScore = score1;
   conformationScore = score2;
   agilityScore = score3;
}
else
{
   obedienceScore = 0;
   conformationScore = 0;
   agilityScore = 0;
}
```

8. In the `display()` method for the `DogTriathlonParticipant2` class, add the following statement that displays a special notice if an error occurred in the number of events value.

```
if(!scoresAgree)
   System.out.println("Notice! Number of events for " +
name + " does not agree with scores reported.");
```

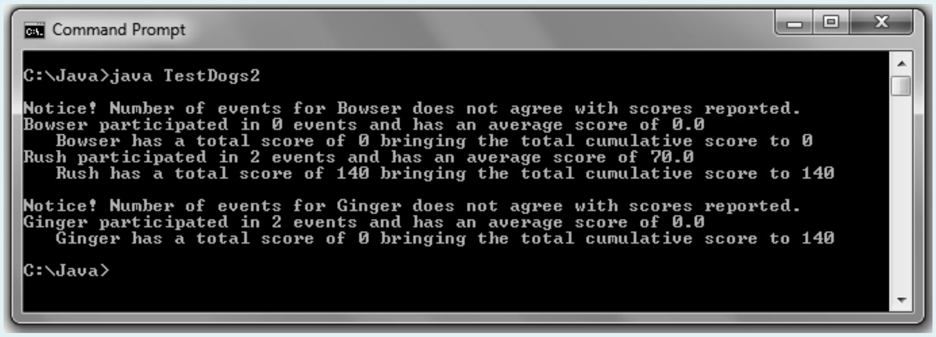
9. Save the file and compile it.

10. Open the **TestDogs.java** file that you created in a "You Do It" section in Chapter 4. Rename the class `TestDogs2`, and immediately save the file as **TestDogs2.java**.

(continues)

(continued)

11. Change `DogTriathlonParticipant` to **`DogTriathlonParticipant2`** in the six places it occurs in the three object declarations.
12. Change the object declarations so that the number of events and the number of nonzero scores used as constructor arguments agree for some objects but not for others.
13. Save the file, and then compile and execute it. Figure 5-32 shows a typical execution in which one participant's entries are valid but the other two contain errors.



```
C:\Java>java TestDogs2

Notice! Number of events for Bowser does not agree with scores reported.
Bowser participated in 0 events and has an average score of 0.0
   Bowser has a total score of 0 bringing the total cumulative score to 0
Rush participated in 2 events and has an average score of 70.0
   Rush has a total score of 140 bringing the total cumulative score to 140

Notice! Number of events for Ginger does not agree with scores reported.
Ginger participated in 2 events and has an average score of 0.0
   Ginger has a total score of 0 bringing the total cumulative score to 140

C:\Java>
```

Figure 5-32 Execution of `TestDogs2` program

14. Change the values in the `TestDogs2` program. Recompile and reexecute the program several times to ensure that using various combinations of number of events and event scores produces appropriate results.
15. On your own, modify the `DogTriathlonParticipant2` class and rename it **`DogTriathlon3`**. In this version, do not use a count of the nonzero score parameters to determine whether the number of events matches the number of valid scores used as arguments. Instead, use only decisions to ensure that the parameters are in agreement. Save the file as **DogTriathlon3.java**, and create a file named **TestDogs3.java** that you can use to test the class. Be sure to test every possible combination of constructor parameters in the `TestDogs3` class—for example, when the events parameter is 2, it is correct whether the nonzero scores are the first and second, the first and third, or the second and third.

Don't Do It

- Don't ignore subtleties in boundaries used in decision making. For example, selecting employees who make less than \$20 an hour is different from selecting employees who make \$20 an hour or less.
- Don't use the assignment operator instead of the comparison operator when testing for equality.
- Don't insert a semicolon after the Boolean expression in an `if` statement; insert the semicolon after the entire statement is completed.
- Don't forget to block a set of statements with curly braces when several statements depend on the `if` or the `else` statement.
- Don't forget to include a complete Boolean expression on each side of an `&&` or `||` operator.
- Don't try to use a `switch` structure to test anything other than an integer, character, or string value.
- Don't forget a `break` statement if one is required by the logic of your `switch` structure.
- Don't use the standard relational operators to compare objects; use them only with the built-in Java types. In the chapter *Characters, Strings, and the `StringBuilder`*, you will learn how to compare `String`s correctly, and in the chapter *Advanced Inheritance Concepts* you will learn to compare other objects.

Key Terms

Pseudocode is a tool that helps programmers plan a program's logic by writing plain English statements.

A **flowchart** is a tool that helps programmers plan a program's logic by writing the steps in diagram form, as a series of shapes connected by arrows.

A **sequence structure** is a logical structure in which one step follows another unconditionally.

A **decision structure** is a logical structure that involves choosing between alternative courses of action based on some value within a program.

`True` or `false` values are **Boolean values**; every computer decision results in a Boolean value.

In Java, the simplest statement you can use to make a decision is the **`if` statement**; you use it to write a single-alternative decision.

The **equivalency operator** (==) compares values and returns `true` if they are equal.

An **empty statement** contains only a semicolon.

A **single-alternative `if`** is a decision structure that performs an action, or not, based on one alternative.

A **dual-alternative `if`** is a decision structure that takes one of two possible courses of action.

In Java, the **`if`…`else` statement** provides the mechanism to perform one action when a Boolean expression evaluates as `true` and to perform a different action when a Boolean expression evaluates as `false`.

The **`if` clause** of an `if`…`else` statement is the part that executes when the evaluated Boolean expression is true.

The **`else` clause** of an `if`…`else` statement is the part that executes when the evaluated Boolean expression is false.

A **nested `if` statement** contains an `if` structure within another `if` structure.

The **logical AND operator** uses two Boolean expressions as operands and evaluates to `true` if both operands are `true`. The AND operator is written as two ampersands (&&).

The **logical OR operator** uses two Boolean expressions as operands and evaluates to `true` if either operand is `true`. The OR operator is written as two pipes (||).

Short-circuit evaluation describes the feature of the AND and OR operators in which evaluation is performed only as far as necessary to make a final decision.

A **range check** is a series of statements that determine within which of a set of ranges a value falls.

An **`else`…`if` clause** is a format used in nested `if` statements in which each instance of `else` and its subsequent `if` are placed on the same line.

The **`switch` statement** uses up to four keywords to test a single variable against a series of exact integer or character values. The keywords are `switch`, `case`, `break`, and `default`.

The **conditional operator** requires three expressions separated with a question mark and a colon and is used as an abbreviated version of the `if`…`else` structure.

A **ternary operator** is one that needs three operands.

You use the **NOT operator**, which is written as the exclamation point (`!`), to negate the result of any Boolean expression.

Chapter Summary

- Making a decision involves choosing between two alternative courses of action based on some value within a program.
- You can use the `if` statement to make a decision based on a Boolean expression that evaluates as `true` or `false`. If the Boolean expression enclosed in parentheses within an `if` statement is `true`, the subsequent statement or block executes. A single-alternative `if` performs an action based on one alternative; a dual-alternative `if`, or `if…else`, provides the mechanism for performing one action when a Boolean expression is `true` and a different action when the expression is `false`.
- To execute more than one statement that depends on the evaluation of a Boolean expression, you use a pair of curly braces to place the dependent statements within a block. Within an `if` or an `else` statement, you can code as many dependent statements as you need, including other `if` and `else` statements.
- Nested `if` statements are particularly useful when two conditions must be met before some action occurs.
- You can use the AND operator (`&&`) within a Boolean expression to determine whether two expressions are both `true`. You use the OR operator (`||`) when you want to carry out some action even if only one of two conditions is `true`.
- New programmers frequently cause errors in their `if` statements when they perform a range check incorrectly or inefficiently, or when they use the wrong operator while trying to make an AND or OR decision.
- You use the `switch` statement to test a single variable against a series of exact integer, character, or string values.
- The conditional operator requires three expressions, a question mark, and a colon and is used as an abbreviated version of the `if…else` statement. The NOT operator (`!`) negates the result of any Boolean expression.
- Operator precedence makes a difference in how expressions are evaluated. You can always use parentheses to change precedence or make your intentions clearer.
- Decisions are frequently used to control field values in constructors and mutator methods.

Review Questions

1. The logical structure in which one instruction occurs after another with no branching is a ________________.
   a. sequence
   b. selection
   c. loop
   d. case

2. Which of the following is typically used in a flowchart to indicate a decision?

   a. square c. diamond
   b. rectangle d. oval

3. Which of the following is not a type of `if` statement?

   a. single-alternative `if` c. reverse `if`
   b. dual-alternative `if` d. nested `if`

4. A decision is based on a(n) ________________ value.

   a. Boolean c. definitive
   b. absolute d. convoluted

5. In Java, the value of `(4 > 7)` is ________________.

   a. 4 c. `true`
   b. 7 d. `false`

6. Assuming the variable `q` has been assigned the value 3, which of the following statements displays XXX?

   a. `if(q > 0) System.out.println("XXX");`
   b. `if(q > 7); System.out.println("XXX");`
   c. Both of the above statements display XXX.
   d. Neither of the above statements displays XXX.

7. What is the output of the following code segment?

```
t = 10;
if(t > 7)
{
   System.out.print("AAA");
   System.out.print("BBB");
}
```

   a. AAA c. AAABBB
   b. BBB d. nothing

8. What is the output of the following code segment?

```
t = 10;
if(t > 7)
   System.out.print("AAA");
   System.out.print("BBB");
```

   a. AAA c. AAABBB
   b. BBB d. nothing

9. What is the output of the following code segment?

```
t = 7;
if(t > 7)
   System.out.print("AAA");
   System.out.print("BBB");
```

a. AAA
b. BBB
c. AAABBB
d. nothing

10. When you code an `if` statement within another `if` statement, as in the following, then the `if` statements are ______________.

```
if(a > b)
   if(c > d)x = 0;
```

a. notched
b. nestled
c. nested
d. sheltered

11. The operator that combines two conditions into a single Boolean value that is `true` only when both of the conditions are `true`, but is `false` otherwise, is ______________.

a. `$$`
b. `!!`
c. `||`
d. `&&`

12. The operator that combines two conditions into a single Boolean value that is `true` when at least one of the conditions is `true` is ______________.

a. `$$`
b. `!!`
c. `||`
d. `&&`

13. Assuming a variable `f` has been initialized to 5, which of the following statements sets `g` to 0?

a. `if(f > 6 || f == 5) g = 0;`
b. `if(f < 3 || f > 4) g = 0;`
c. `if(f >= 0 || f < 2) g = 0;`
d. All of the above statements set `g` to 0.

14. Which of the following groups has the lowest operator precedence?

a. relational
b. equality
c. addition
d. logical OR

15. Which of the following statements correctly outputs the names of voters who live in district 6 and all voters who live in district 7?

    a.
    ```
    if(district == 6 || 7)
    System.out.println("Name is " + name);
    ```
    b.
    ```
    if(district == 6 || district == 7)
    System.out.println("Name is " + name);
    ```
    c.
    ```
    if(district = 6 && district == 7)
    System.out.println("Name is " + name);
    ```
    d. two of these

16. Which of the following displays "Error" when a student ID is less than 1000 or more than 9999?

    a.
    ```
    if(stuId < 1000) if(stuId > 9999)
    System.out.println("Error");
    ```
    b.
    ```
    if(stuId < 1000 && stuId > 9999)
    System.out.println("Error");
    ```
    c.
    ```
    if(stuId < 1000)
      System.out.println("Error");
    else
      if(stuId > 9999)
        System.out.println("Error");
    ```
    d. Two of these are correct.

17. You can use the ______________ statement to terminate a `switch` structure.

    a. `switch`
    b. `end`
    c. `case`
    d. `break`

18. The argument tested in a `switch` structure can be any of the following *except* a(n) ______________.

    a. `int`
    b. `char`
    c. `double`
    d. `String`

19. Assuming a variable `w` has been assigned the value 15, what does the following statement do?

    ```
    w == 15 ? x = 2 : x = 0;
    ```

    a. assigns 15 to `w`
    b. assigns 2 to `x`
    c. assigns 0 to `x`
    d. nothing

20. Assuming a variable `y` has been assigned the value 6, the value of `!(y < 7)` is ________________ .

    a. 6
    b. 7
    c. `true`
    d. `false`

Exercises

Programming Exercises

1. Write an application that asks a user to enter an integer. Display a statement that indicates whether the integer is even or odd. Save the file as **EvenOdd.java**.

2. Write an application that prompts the user for the day's high and low temperatures. If the high is greater than or equal to 90 degrees, display the message, "Heat warning." If the low is less than 32 degrees, display the message "Freeze warning." If the difference between the high and low temperatures is more than 40 degrees, display the message, "Large temperature swing." Save the file as **Temperatures.java**.

3. a. Write an application for the Summerdale Condo Sales office; the program determines the price of a condominium. Ask the user to choose *1* for park view, *2* for golf course view, or *3* for lake view. The output is the name of the chosen view as well as the price of the condo. Park view condos are $150,000, condos with golf course views are $170,000, and condos with lake views are $210,000. If the user enters an invalid code, set the price to 0. Save the file as **CondoSales.java**.

   b. Add a prompt to the `CondoSales` application to ask the user to specify a (1) garage or a (2) parking space, but only if the condo view selection is valid. Add $5,000 to the price of any condo with a garage. If the parking value is invalid, display an appropriate message and assume that the price is for a condo with no garage. Save the file as **CondoSales2.java**.

4. a. The Williamsburg Women's Club offers scholarships to local high school students who meet any of several criteria. Write an application that prompts the user for a student's numeric high school grade point average (for example, 3.2), the student's number of extracurricular activities, and the student's number of service activities. Display the message "Scholarship candidate" if the student has any of the following:

   - A grade point average of 3.8 or above and at least one extracurricular activity and one service activity
   - A grade point average below 3.8 but at least 3.4 and a total of at least three extracurricular and service activities
   - A grade point average below 3.4 but at least 3.0 and at least two extracurricular activities and three service activities

If the student does not meet any of the qualification criteria, display “Not a candidate.” Save the file as **Scholarship.java**.

b. Modify the `Scholarship` application so that if a user enters a grade point average under 0 or over 4.0, or a negative value for either of the activities, an error message appears. Save the file as **Scholarship2.java**.

5. Write an application that displays a menu of three items for the Jivin’ Java Coffee Shop as follows:

(1) American	1.99
(2) Espresso	2.50
(3) Latte	2.15

Prompt the user to choose an item using the number (1, 2, or 3) that corresponds to the item, or to enter *0* to quit the application. After the user makes the first selection, if the choice is *0*, display a total bill of $0. Otherwise, display the menu again. The user should respond to this prompt with another item number to order or *0* to quit. If the user types *0*, display the cost of the single requested item. If the user types *1*, *2*, or *3*, add the cost of the second item to the first, and then display the menu a third time. If the user types *0* to quit, display the total cost of the two items; otherwise, display the total for all three selections. Save the file as **Coffee.java**.

6. Barnhill Fastener Company runs a small factory. The company employs workers who are paid one of three hourly rates depending on skill level:

Skill Level	Hourly Pay Rate ($)
1	17.00
2	20.00
3	22.00

Each factory worker might work any number of hours per week; any hours over 40 are paid at one and one-half times the usual rate.

In addition, workers in skill levels 2 and 3 can elect the following insurance options:

Option	Explanation	Weekly Cost to Employee ($)
1	Medical insurance	32.50
2	Dental insurance	20.00
3	Long-term disability insurance	10.00

Also, workers in skill level 3 can elect to participate in the retirement plan at 3% of their gross pay.

Write an interactive Java payroll application that calculates the net pay for a factory worker. The program prompts the user for skill level and hours worked, as well as appropriate insurance and retirement options for the employee's skill level category. The application displays: (1) the hours worked, (2) the hourly pay rate, (3) the regular pay for 40 hours, (4) the overtime pay, (5) the total of regular and overtime pay, and (6) the total itemized deductions. If the deductions exceed the gross pay, display an error message; otherwise, calculate and display (7) the net pay after all the deductions have been subtracted from the gross. Save the file as **Pay.java**.

7. Create a class for Shutterbug's Camera Store, which is having a digital camera sale. The class is named `DigitalCamera`, and it contains data fields for a brand, the number of megapixels in the resolution, and price. Include a constructor that takes arguments for the brand and megapixels. If the megapixel parameter is greater than 10, the constructor sets it to 10. The sale price is set based on the resolution; any camera with 6 megapixels or fewer is $99, and all other cameras are $129. Also include a method that displays `DigitalCamera` details. Write an application named `TestDigitalCamera` in which you instantiate at least four objects, prompt the user for values for the camera brand and number of megapixels, and display all the values. Save the files as **DigitalCamera.java** and **TestDigitalCamera.java**.

8. Create a class for the Parks Department in Cloverdale. The class is named `Park`, and it contains a `String` field for the name of the park, an integer field for the size in acres, and four Boolean fields that hold whether the park has each of these features: picnic facilities, a tennis court, a playground, and a swimming pool. Include get and set methods for each field. Include code in the method that sets the number of acres and does not allow a negative number or a number over 400. Save the file as **Park.java**.

   Then, create a program with methods that do the following:

   - Accepts a `Park` parameter and returns a Boolean value that indicates whether the `Park` has both picnic facilities and a playground.
   - Accepts a `Park` parameter and four Boolean parameters that represent requests for the previously mentioned `Park` features. The method returns `true` if the `Park` has all the requested features.

- Accepts a `Park` parameter and four Boolean parameters that represent requests for the previously mentioned `Park` features. The method returns `true` if the `Park` has exactly all the requested features and no others.
- Accepts a `Park` parameter and returns the number of facilities that the `Park` features.

- Accepts two `Park` parameters and returns the larger `Park`.

Declare at least three `Park` objects, and demonstrate that all the methods work correctly. Save the program as **TestPark.java**.

9. a. Create a class named `Invoice` that holds an invoice number, balance due, and three fields representing the month, day, and year when the balance is due. Create a constructor that accepts values for all five data fields. Within the constructor, assign each argument to the appropriate field with the following exceptions:

   - If an invoice number is less than 1000, force the invoice number to 0.
   - If the month field is less than 1 or greater than 12, force the month field to 0.
   - If the day field is less than 1 or greater than 31, force the day field to 0.
   - If the year field is less than 2011 or greater than 2017, force the year field to 0.

   In the `Invoice` class, include a display method that displays all the fields on an `Invoice` object. Save the file as **Invoice.java**.

   b. Write an application containing a `main()` method that declares several `Invoice` objects, proving that all the statements in the constructor operate as specified. Save the file as **TestInvoice.java**.

   c. Modify the constructor in the `Invoice` class so that the day is not greater than 31, 30, or 28, depending on the month. For example, if a user tries to create an invoice for April 31, force it to April 30. Also, if the month is invalid, and thus forced to 0, also force the day to 0. Save the modified `Invoice` class as **Invoice2.java**. Then modify the `TestInvoice` class to create `Invoice2` objects. Create enough objects to test every decision in the constructor. Save this file as **TestInvoice2.java**.

10. Use the Web to locate the lyrics to the traditional song “The Twelve Days of Christmas.” The song contains a list of gifts received for the holiday. The list is cumulative so that as each “day” passes, a new verse contains all the words of the previous verse, plus a new item. Write an application that displays the words to the song starting with any day the user enters. (*Hint*: Use a `switch` statement with `case`s in descending day order and without any `break` statements so that the lyrics for any day repeat all the lyrics for previous days.) Save the file as **TwelveDays.java**.

Debugging Exercises

1. Each of the following files in the Chapter05 folder of your downloadable student files has syntax and/or logic errors. In each case, determine the problem and fix the program. After you correct the errors, save each file using the same filename preceded with *Fix*. For example, save DebugFive1.java as FixDebugFive1.java.

   a. DebugFive1.java
   b. DebugFive2.java
   c. DebugFive3.java
   d. DebugFive4.java

Game Zone

1. In Chapter 1, you created a class called `RandomGuess`. In this game, players guess a number, the application generates a random number, and players determine whether they were correct. Now that you can make decisions, modify the application so it allows a player to enter a guess before the random number is displayed and then displays a message indicating whether the player's guess was correct, too high, or too low. Save the file as **RandomGuess2.java**. (After you finish the next chapter, you will be able to modify the application so that the user can continue to guess until the correct answer is entered.)

2. Create a lottery game application. Generate three random numbers (see Appendix D for help in doing so), 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-digit number, and the amount of money the user has won as follows:

Matching Numbers	Award ($)
Any one matching	10
Two matching	100
Three matching, not in order	1,000
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. Save the file as **Lottery.java**.

3. In Chapter 3, you created a `Card` class. Modify the `Card` class so the `setValue()` method does not allow a `Card`'s value to be less than 1 or higher than 13. If the argument to `setValue()` is out of range, assign 1 to the `Card`'s value.

   In Chapter 3, you also created a `PickTwoCards` application that randomly selects two playing cards and displays their values. In that application, all `Card` objects arbitrarily were assigned a suit represented by a single character, but they could have different values, and the player observed which of two `Card` objects had the higher value. Now, modify the application so the suit and the value both are chosen randomly. Using two `Card` objects, play a very simple version of the card game War. Deal two `Cards`—one for the computer and one for the player—and determine the higher card, then display a message indicating whether the cards are equal, the computer won, or the player won. (Playing cards are considered equal when they have the same value, no matter what their suit is.) For this game, assume the Ace (value 1) is low. Make sure that the two `Cards` dealt are not the same `Card`. For example, a deck cannot contain more than one `Card` representing the 2 of spades. If two cards are chosen to have the same value, change the suit for one of them. Save the application as **War.java**. (After you study the chapter *Arrays*, you will be able to create a more sophisticated War game in which you use an entire deck without repeating cards.)

4. In Chapter 4, you created a `Die` class from which you could instantiate an object containing a random value from 1 through 6. You also wrote an application that randomly "throws" two dice and displays their values. Modify the application so it determines whether the two dice are the same, the first has a higher value, or the second has a higher value. Save the application as **TwoDice2.java**.

5. In the game Rock Paper Scissors, two players simultaneously choose one of three options: rock, paper, or scissors. If both players choose the same option, then the result is a tie. However, if they choose differently, the winner is determined as follows:

   - Rock beats scissors, because a rock can break a pair of scissors.
   - Scissors beats paper, because scissors can cut paper.
   - Paper beats rock, because a piece of paper can cover a rock.

   Create a game in which the computer randomly chooses rock, paper, or scissors. Let the user enter a number 1, 2, or 3, each representing one of the three choices. Then, determine the winner. Save the application as **RockPaperScissors.java**. (In the chapter *Characters, Strings, and the `StringBuilder`*, you will modify the game so that the user enters a string for *rock*, *paper*, and *scissors*, rather than just entering a number.)

Case Problems

1. a. Carly's Catering provides meals for parties and special events. In Chapters 3 and 4, you created an `Event` class for the company. Now, make the following changes to the class:

    - Currently, the class contains a field that holds the price for an `Event`. Now add another field that holds the price per guest, and add a `public` method to return its value.
    - Currently, the class contains a constant for the price per guest. Replace that field with two fields—a lower price per guest that is $32, and a higher price per guest that is $35.
    - Add a new method named `isLargeEvent()` that returns `true` if the number of guests is 50 or greater and otherwise returns `false`.
    - Modify the method that sets the number of guests so that a large `Event` (over 50 guests) uses the lower price per guest to set the new `pricePerGuest` field and calculate the total `Event` price. A small `Event` uses the higher price.

    Save the file as **Event.java**.

    b. In Chapter 4, you modified the `EventDemo` class to demonstrate two `Event` objects. Now, modify that class again as follows:

    - Instantiate three `Event` objects, and prompt the user for values for each object.
    - Change the method that displays `Event` details to use the new `isLargeEvent()` method and the new price per guest value. Use the display method with all three objects.
    - Create a method that accepts two `Event` objects and returns the larger one based on number of guests. (If the `Events` have the same number of guests, you can return either object.) Call this method three times—once with each pair of instantiated `Events`—and display the event number and number of guests for each argument as well as the event number and number of guests for the larger `Event`.

    Save the file as **EventDemo.java**.

2. a. Sammy's Seashore Supplies rents beach equipment such as kayaks, canoes, beach chairs, and umbrellas to tourists. In Chapters 3 and 4, you created a `Rental` class for the company.

   Now, make the following change to the class:

   - Currently, a rental price is calculated as $40 per hour plus $1 for each minute over a full hour. This means that a customer who rents equipment for 41 or more minutes past an hour pays more than a customer who waits until the next hour to return the equipment. Change the price calculation so that a customer pays $40 for each full hour and $1 for each extra minute up to and including 40 minutes.

   Save the file as **Rental.java**.

   b. In Chapter 4, you modified the `RentalDemo` class to demonstrate a `Rental` object. Now, modify that class again as follows:

   - Instantiate three `Rental` objects, and prompt the user for values for each object. Display the details for each object to verify that the new price calculation works correctly.
   - Create a method that accepts two `Rental` objects and returns the one with the longer rental time. (If the `Rental`s have the same time, you can return either object.) Call this method three times—once with each pair of instantiated `Rental`s—and display the contract number and time in hours and minutes for each argument as well as the contract number of the longer `Rental`.

   Save the file as **RentalDemo.java**.