CHAPTER 4

Making Decisions

In this chapter you will:

- Understand logic-planning tools and decision making
- Learn how to make decisions using the `if` statement
- Learn how to make decisions using the `if-else` statement
- Use compound expressions in `if` statements
- Make decisions using the `switch` statement
- Use the conditional operator
- Use the NOT operator
- Learn to avoid common errors when making decisions
- Learn about decision-making issues in GUI programs

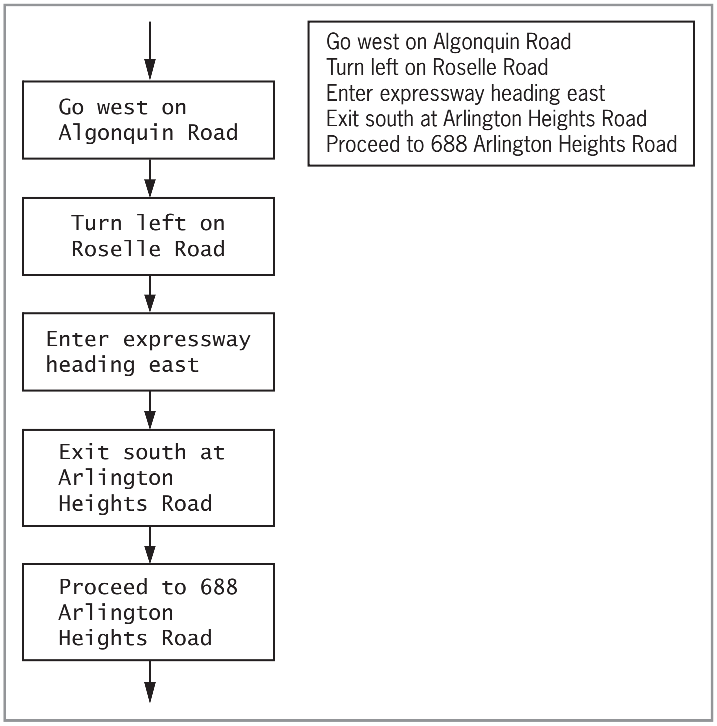
Computer programs are powerful because of their ability to make decisions. Programs that decide which travel route will offer the best weather conditions, which Web site will provide the closest match to search criteria, or which recommended medical treatment has the highest probability of success all rely on a program's decision making. In this chapter you will learn to make decisions in C# programs.

Understanding Logic-Planning Tools and Decision Making

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.

You learned the difference between a program's logic and its syntax in Chapter 1.

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 4-1 shows a flowchart and pseudocode describing driving directions to a friend's house. Notice how the actions illustrated in the flowchart and the pseudocode statements correspond. The logic in Figure 4-1 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 4-1 Flowchart and pseudocode for 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. Flowchart creators use diamond shapes to indicate alternative courses of action, which are drawn starting from the sides of the diamonds. Figure 4-2 shows a flowchart describing directions in which the execution of some steps depends on decisions.

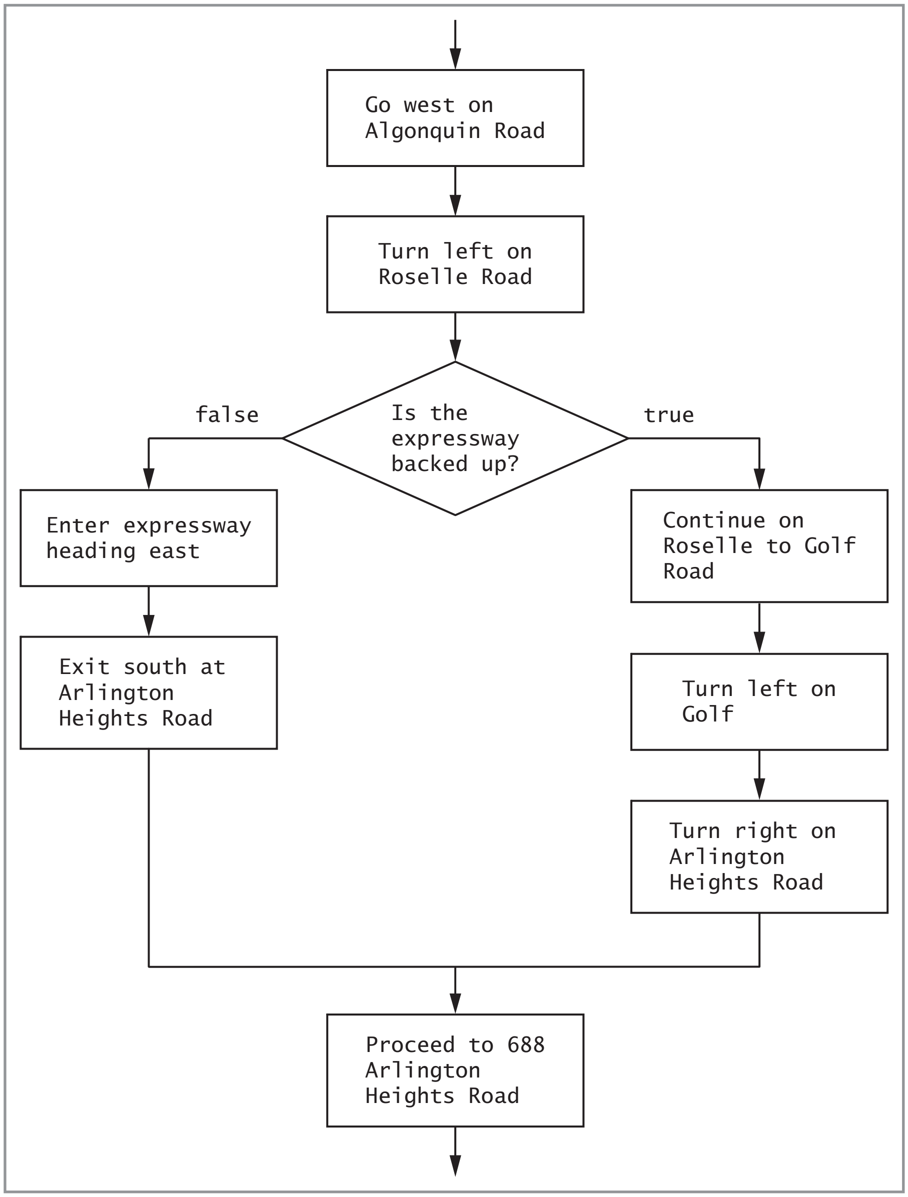

Go west on Algonquin Road
Turn left on Roselle Road
Is the expressway backed up?
false
true
Enter expressway heading east
Exit south at Arlington Heights Road
Continue on Roselle to Golf Road
Turn left on Golf
Turn right on Arlington Heights Road
Proceed to 688 Arlington Heights Road

Figure 4-2 Flowchart including a decision

Figure 4-2 shows a **decision structure**—one that involves choosing between alternative courses of action based on some value within a program. When reduced to their most basic form, all computer decisions are true-or-false decisions. 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. The values `true` and `false` are Boolean values; every computer decision results in a Boolean value. Thus, internally, a program you write 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?"

TWO TRUTHS & A LIE

Understanding Logic-Planning Tools and Decision Making

1. A sequence structure has three or more alternative logical paths.
2. A decision structure involves choosing between alternative courses of action based on some value within a program.
3. When reduced to their most basic form, all computer decisions are yes-or-no decisions.

The false statement is #1. In a sequence structure, one step follows another unconditionally.

Making Decisions Using the `if` Statement

You learned about Boolean expressions and the `bool` data type in Chapter 2. Table 2-4 summarizes how you use the comparison operators.

The `if` and `if-else` statements are the two most commonly used decision-making statements in C#. You use an **`if` statement** to make a single-alternative decision. In other words, you use an `if` statement to determine whether an action will occur. The `if` statement takes the following form:

```
if(testedExpression)
   statement;
```

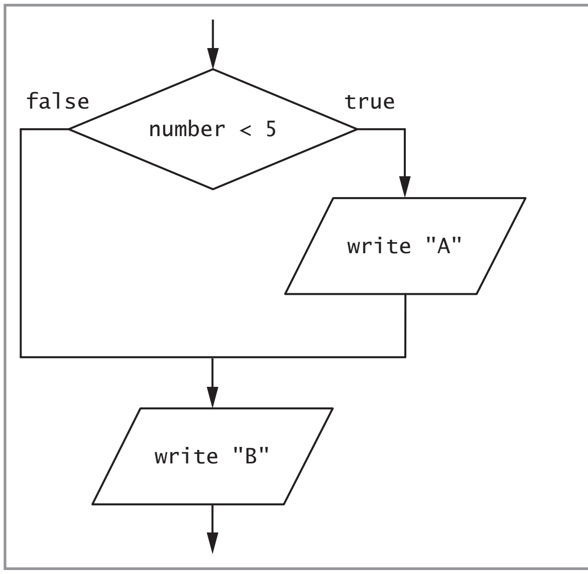
where *testedExpression* represents any C# expression that can be evaluated as `true` or `false` and *statement* represents the action that will take place if the expression evaluates as `true`. You must place the `if` statement's evaluated expression between parentheses.

In the chapter *Introduction to Methods*, you will learn to write methods that return values. A method call that returns a Boolean value also can be used as the tested expression in an `if` statement.

Usable expressions in an `if` statement include Boolean expressions such as `amount > 5` and `month == "May"` as well as the value of `bool` variables such as `isValidIDNumber`. If the expression evaluates as `true`, then the statement executes. Whether the expression evaluates as `true` or `false`, the program continues with the next statement following the complete `if` statement.

In some programming languages, such as C++, nonzero numbers evaluate as true and 0 evaluates as false. However, in C#, only Boolean expressions evaluate as true and false.

For example, the code segment written and diagrammed in Figure 4-3 displays "A" and "B" when number holds a value less than 5. The expression number < 5 evaluates as true, so the statement that displays "A" executes. Then the independent statement that displays "B" executes.



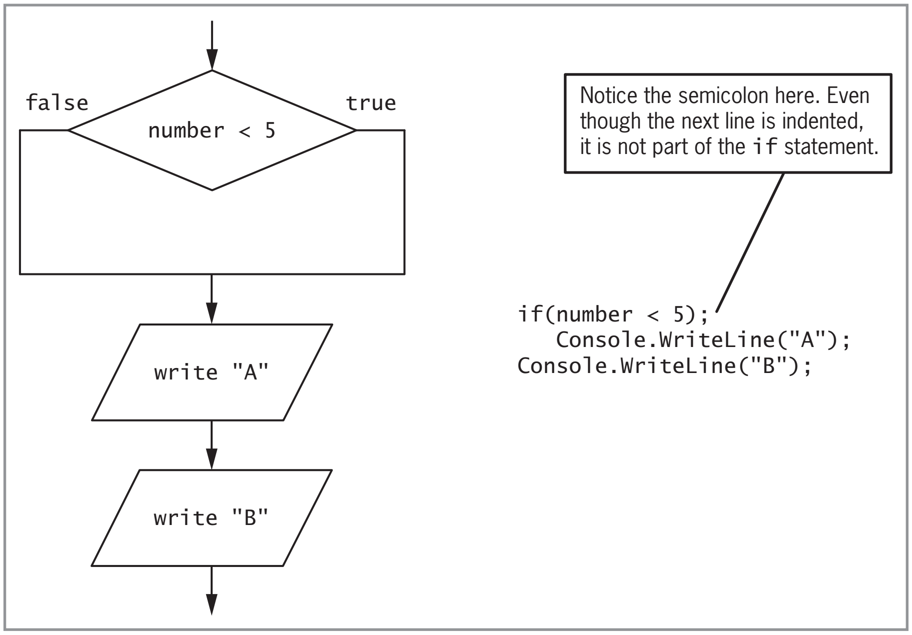
```
if(number < 5)
   Console.WriteLine("A");
Console.WriteLine("B");
```

Figure 4-3 Flowchart and code including a typical if statement followed by a separate statement

You can leave a space between the keyword if and the opening parenthesis if you think that format is easier to read.

When an evaluated expression is false, the rest of the statement does not execute. For example, when number is 5 or greater in Figure 4-3, only "B" is displayed. Because the expression number < 5 is false, the statement that displays "A" never executes.

In Figure 4-3, notice there is no semicolon at the end of the line that contains if(number < 5). The statement does not end at that point; it ends after Console.WriteLine("A");. If you incorrectly insert a semicolon at the end of if(number < 5), then the statement says, "If number is less than 5, do nothing; then, no matter what the value of `number` is, the next independent statement displays "A". Figure 4-4 shows the flowchart logic that matches the code when a semicolon is placed at the end of the `if` expression.


false
true
number < 5
write "A"
write "B"
Notice the semicolon here. Even though the next line is indented, it is not part of the if statement.

```
if(number < 5);
   Console.WriteLine("A");
Console.WriteLine("B");
```

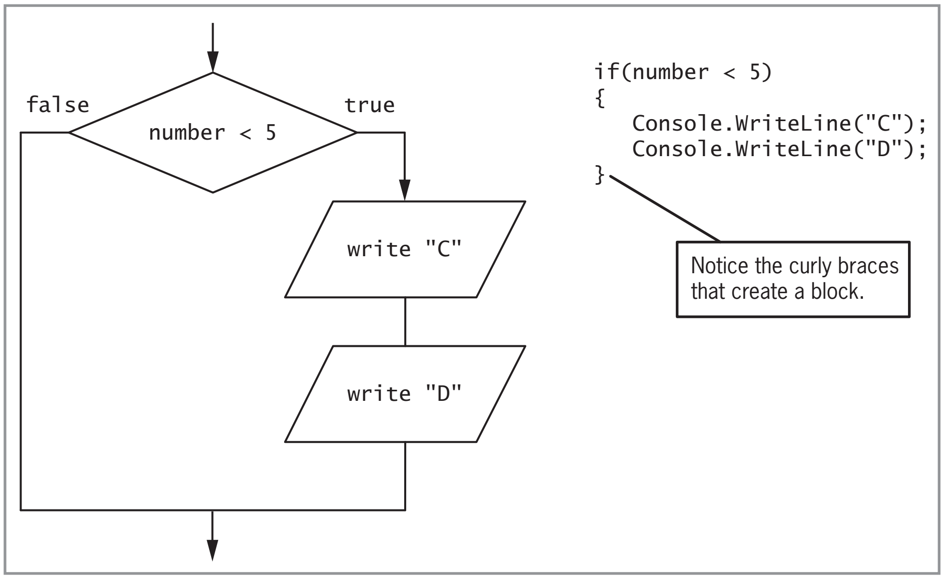
Figure 4-4 Flowchart and code including an `if` statement with a semicolon following the `if` expression

Although it is customary, and good style, to indent any statement that executes when an `if` Boolean expression evaluates as `true`, the C# compiler does not pay any attention to the indentation. Each of the following `if` statements displays "A" when `number` is less than 5. The first shows an `if` written on a single line; the second shows an `if` on two lines but with no indentation. The third uses conventional indentation.

Although these first two formats work for `if` statements, they are not conventional and using them makes a program harder to understand.

```
if(number < 5) Console.WriteLine("A");
if(number < 5)
Console.WriteLine("A");
if(number < 5)
   Console.WriteLine("A");
```

When you want to execute two or more statements conditionally, you must place the statements within a block. A **block** is a collection of one or more statements contained within a pair of curly braces. For example, the code segment written and diagrammed in Figure 4-5 displays both "C" and "D" when `number` is less than 5, and it displays neither when `number` is not less than 5.



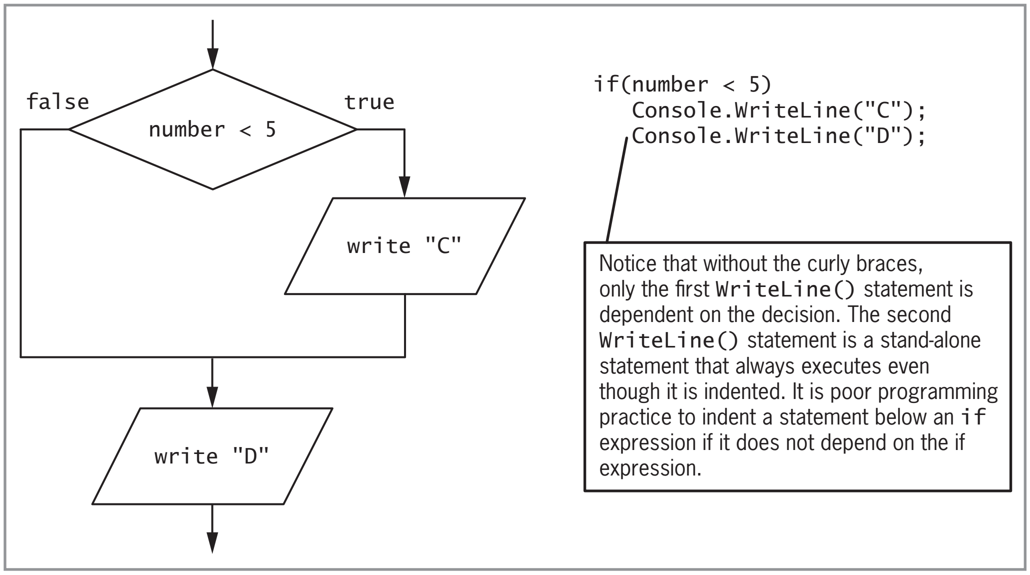
```
if(number < 5)
{
   Console.WriteLine("C");
   Console.WriteLine("D");
}
```

Notice the curly braces that create a block.

Figure 4-5 Flowchart and code including a typical `if` statement containing a block

Indenting alone does not cause multiple statements to depend on the evaluation of a Boolean expression in an `if`. For multiple statements to depend on an `if`, they must be blocked with braces. For example, Figure 4-6 shows two statements that are indented below an `if` expression. When you glance at the code, it might first appear that both statements depend on the `if`; in fact, however, only the first one does, as shown in the flowchart, because the statements are not blocked.

The `if` expression that precedes the block is the **control statement** for the decision structure.



```
if(number < 5)
   Console.WriteLine("C");
   Console.WriteLine("D");
```

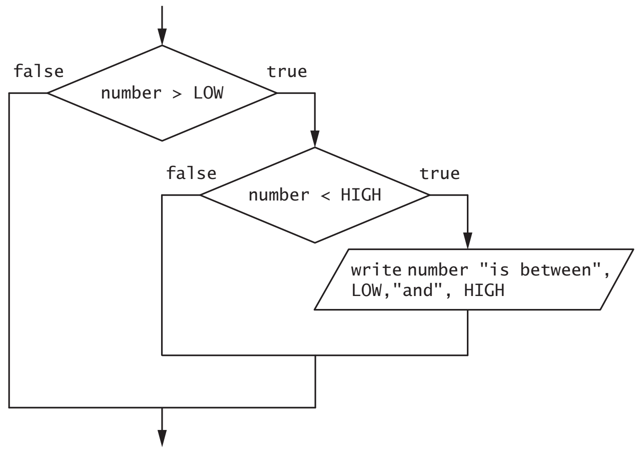
Notice that without the curly braces, only the first `WriteLine()` statement is dependent on the decision. The second `WriteLine()` statement is a stand-alone statement that always executes even though it is indented. It is poor programming practice to indent a statement below an `if` expression if it does not depend on the if expression.

Figure 4-6 Flowchart and code including an `if` statement that is missing curly braces or that has inappropriate indenting

In C#, it is customary to align opening and closing braces in a block. Some programmers prefer to place the opening brace on the same line as the `if` expression instead of giving the brace its own line. This style is called the **K & R style**, named for Brian Kernighan and Dennis Ritchie, who wrote the first book on the C programming language.

When you create a block using curly braces, you do not have to place multiple statements within it. It is perfectly legal to block a single statement. Blocking a single statement can be a useful technique to help prevent future errors. When a program later is modified to include multiple statements that depend on the `if`, it is easy to forget to add curly braces. You will naturally place the additional statements within the block if the braces are already in place. It also is legal to create a block that contains no statements. You usually do so only when starting to write a program, as a reminder to yourself to add statements later.

You can place any number of statements within the block contained by the curly braces, including other `if` statements. Of course, if a second `if` statement is the only statement that depends on the first `if`, then no braces are required. Figure 4-7 shows the logic for a **nested `if`** statement—one in which one decision structure is contained within another. With a nested `if` statement, a second `if`'s Boolean expression is tested only when the first `if`'s Boolean expression evaluates as `true`.



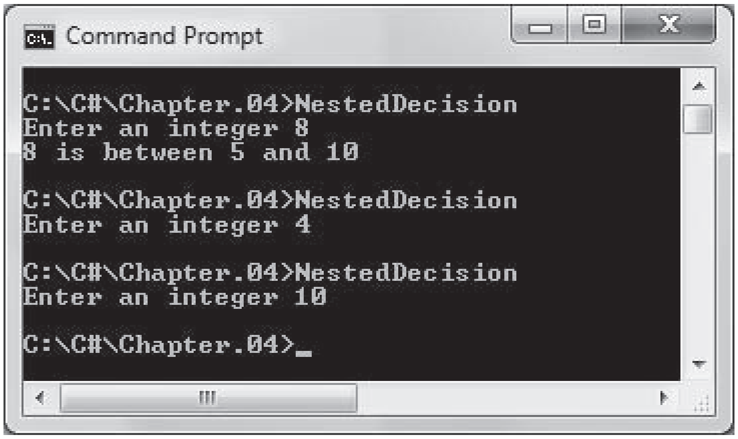
```
if(number > LOW)
   if(number < HIGH)
      Console.WriteLine("{0} is between {1} and {2}",
         number, LOW, HIGH);
```

Figure 4-7 Flowchart and code showing the logic of a nested `if`

Figure 4-8 shows a program that contains the logic. When a user enters a number greater than 5 in the program in Figure 4-8, the first `if` expression evaluates as `true` and the `if` statement that tests whether the number is less than 10 executes. When the second `if` evaluates as `true`, the `Console.WriteLine()` statement executes. However, if the second `if` is `false`, no output occurs. When the user enters a number less than or equal to 5, the first if expression is false and the second if expression is never tested, and again no output occurs. Figure 4-9 shows the output after the program is executed three times using three different input values. Notice that when the value input by the user is not between 5 and 10, no output message appears; the message is displayed only when both if expressions are true.

```
using System;
public class NestedDecision
{
   public static void Main()
   {
      const int HIGH = 10, LOW = 5;
      string numberString;
      int number;
      Console.Write("Enter an integer ");
      numberString = Console.ReadLine();
      number = Convert.ToInt32(numberString);
        if(number > LOW)
          if(number < HIGH)
            Console.WriteLine("{0} is between {1} and {2}",
              number, LOW, HIGH);
   }
}
```

Figure 4-8 Program using nested if



```
C:\C#\Chapter.04>NestedDecision
Enter an integer 8
8 is between 5 and 10

C:\C#\Chapter.04>NestedDecision
Enter an integer 4

C:\C#\Chapter.04>NestedDecision
Enter an integer 10

C:\C#\Chapter.04>_
```

Figure 4-9 Output of three executions of the NestedDecision program

A Note on Equivalency Comparisons

Often, programmers mistakenly use a single equal sign rather than the double equal sign when attempting to determine equivalency. For example, the following expression does not compare `number` to `HIGH`:

```
number = HIGH
```

Instead, it attempts to assign the value `HIGH` to the variable `number`. When it is part of an `if` statement, this assignment is illegal.

The only condition under which the assignment operator would work as part of the tested expression in an `if` statement is when the assignment is made to a `bool` variable. For example, suppose a payroll program contains a `bool` variable named `doesEmployeeHaveDependents`, and then uses the following statement:

```
if(doesEmployeeHaveDependents = numDependents > 0)...
```

In this case, `numDependents` would be compared to 0, and the result, `true` or `false`, would be assigned to `doesEmployeeHaveDependents`. Then the decision would be made based on the result.

TWO TRUTHS & A LIE

Making Decisions Using the `if` Statement

1. In C#, you must place an `if` statement's evaluated expression between parentheses.
2. In C#, for multiple statements to depend on an `if`, they must be indented.
3. In C#, you can place one `if` statement within a block that depends on another `if` statement.

The false statement is #2. Indenting alone does not cause multiple statements to depend on the evaluation of a Boolean expression in an `if`. For multiple statements to depend on an `if`, they must be blocked with braces.

Making Decisions Using the `if-else` Statement

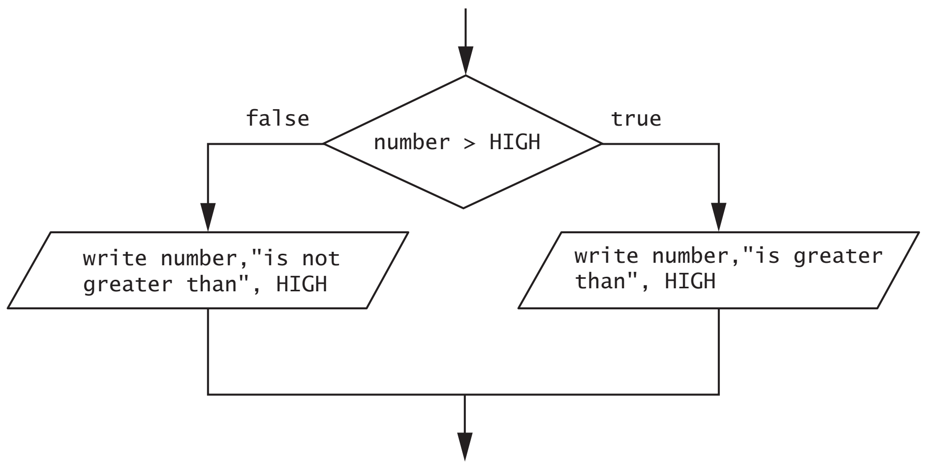
Some decisions you make are **dual-alternative decisions**; they have two possible resulting actions. If you want to perform one action when a Boolean expression evaluates as `true` and an alternate action when it evaluates as false, you can use an **if-else statement**. The if-else statement takes the following form:

```
if(expression)
   statement1;
else
   statement2;
```

You can code an if without an else, but it is illegal to code an else without an if.

Just as you can block several statements so they all execute when an expression within an if is true, you can block multiple statements after an else so that they will all execute when the evaluated expression is false.

For example, Figure 4-10 shows the logic for an if-else statement, and Figure 4-11 shows a program that contains the statement. With every execution of the program, one or the other of the two WriteLine() statements executes. Figure 4-12 shows two executions of the program.


false
true
number > HIGH
write number,"is not greater than", HIGH
write number,"is greater than", HIGH

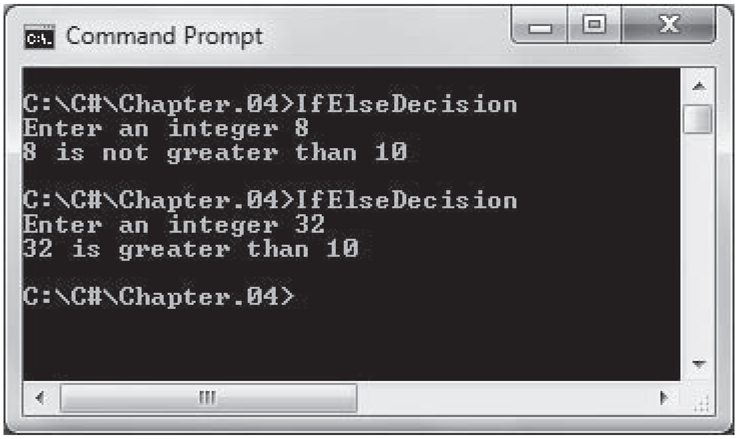
```
if(number > HIGH)
   Console.WriteLine("{0} is greater than {1}",
      number, HIGH);
else
   Console.WriteLine("{0} is not greater than {1}",
      number, HIGH);
```

Figure 4-10 Flowchart and code showing the logic of a dual-alternative if-else statement

```
using System;
public class IfElseDecision
{
   public static void Main()
   {
      const int HIGH = 10;
      string numberString;
      int number;
      Console.Write("Enter an integer ");
      numberString = Console.ReadLine();
      number = Convert.ToInt32(numberString);
      if(number > HIGH)
        Console.WriteLine("{0} is greater than {1}",
          number, HIGH);
      else
        Console.WriteLine("{0} is not greater than {1}",
          number, HIGH);
   }
}
```

The indentation shown in the `if-else` example in Figure 4-11 is not required, but is standard. You vertically align the keyword `if` with the keyword `else`, and then indent the action statements that depend on the evaluation.

Figure 4-11 Program with a dual-alternative `if-else` statement



```
C:\C#\Chapter.04>IfElseDecision
Enter an integer 8
8 is not greater than 10

C:\C#\Chapter.04>IfElseDecision
Enter an integer 32
32 is greater than 10

C:\C#\Chapter.04>
```

Figure 4-12 Output of two executions of the `IfElseDecision` program

When `if-else` statements are nested, each `else` always is paired with the most recent unpaired `if`. For example, in the following code, the `else` is paired with the second `if`.

```
if(saleAmount > 1000)
   if(saleAmount < 2000)
      bonus = 100;
   else
      bonus = 50;
```

In this example, the following bonuses are assigned:

- If `saleAmount` is between \$1000 and \$2000, `bonus` is \$100 because both evaluated expressions are `true`.
- If `saleAmount` is \$2000 or more, `bonus` is \$50 because the first evaluated expression is `true` and the second one is `false`.
- If `saleAmount` is \$1000 or less, `bonus` is unassigned because the first evaluated expression is `false` and there is no corresponding `else`.

TWO TRUTHS & A LIE

Making Decisions Using the `if-else` Statement

1. Dual-alternative decisions have two possible outcomes.
2. In an `if-else` statement, a semicolon is always the last character typed before the `else`.
3. When `if-else` statements are nested, the first `if` always is paired with the first `else`.

The false statement is #3. When `if-else` statements are nested, each `else` always is paired with the most recent unpaired `if`.

Using Compound Expressions in `if` Statements

In many programming situations you encounter, you need to make multiple decisions before taking action. For example, suppose a specific college scholarship is available:

- If your high school class rank is higher than 75 percent
- And if your grade-point average is higher than 3.0
- And if you are a state resident
- Or if you are a resident of a cooperating state
- Or if you have participated in at least five extracurricular activities and held a part-time job

No matter how many decisions must be made, you can decide on the scholarship eligibility for any student by using a series of `if` statements to test the appropriate variables. For convenience and clarity, however, you can combine multiple decisions into a single `if` statement using a combination of conditional AND and OR operators to create compound Boolean expressions.

Using the Conditional AND Operator

As an alternative to nested `if` statements, you can use the **conditional AND operator** (or simply the **AND operator**) to create a compound Boolean expression. The conditional AND operator is written as two ampersands (&&).

A tool that can help you understand the && operator is a truth table. **Truth tables** are diagrams used in mathematics and logic to help describe the truth of an entire expression based on the truth of its parts. Table 4-1 shows a truth table that lists all the possibilities with compound Boolean expressions. For any two expressions `x` and `y`, the expression `x && y` is true only if both `x` and `y` are individually true. If either `x` or `y` alone is false, or if both are false, then the expression `x && y` is false.

x	y	x && y
True	True	True
True	False	False
False	True	False
False	False	False

Table 4-1 Truth table for the conditional && operator

For example, the two code samples shown in Figure 4-13 work exactly the same way. The `age` variable is tested, and if it is greater than or equal to 0 and less than 120, a message is displayed to explain that the value is valid.

```
// using &&
  if(age >= 0 && age < 120)
     Console.WriteLine("Age is valid");
// using nested ifs
  if(age >= 0)
     if(age < 120)
       Console.WriteLine("Age is valid");
```

Figure 4-13 Comparison of the && operator and nested `if` statements

Using the && operator is never required, because nested `if` statements achieve the same result, but using the && operator often makes your code more concise, less error-prone, and easier to understand.

It is important to note that when you use the && operator, you must include a complete Boolean expression on each side of the operator. If you want to set a bonus to $400 when a `saleAmount` is both over $1000 and under $5000, the correct statement is as follows:

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

The following statement is incorrect and will not compile:

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

5000 is not a Boolean expression (it is a numeric constant), so this statement is invalid.

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

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

Use this format if it is clearer to you.

The expressions in each part of a compound Boolean expression are evaluated only as much as necessary to determine whether the entire expression is `true` or `false`. This feature is called **short-circuit evaluation.** With the && operator, both Boolean expressions must be `true` before the action in the statement can occur. If the first expression is `false`, the second expression is never evaluated, because its value does not matter. For example, if `a` is not greater than `LIMIT` in the following `if` statement, then the evaluation is complete because there is no need to evaluate whether `b` is greater than `LIMIT`.

```
if(a > LIMIT && b > LIMIT)
   Console.WriteLine("Both are greater than LIMIT");
```

Using the Conditional OR Operator

You can use the **conditional OR operator** (or simply the **OR operator**) when you want some action to occur even if only one of two conditions is `true`. The OR operator is written as `||`. When you use the `||` operator, only one of the listed conditions must be met for the resulting action to take place. Table 4-2 shows the truth table for the `||` operator. As you can see, the entire expression `x || y` is false only when `x` and `y` each are false individually.

x	y	x || y
True	True	True
True	False	True
False	True	True
False	False	False

Table 4-2 Truth table for the conditional `||` operator

You create the conditional OR operator by using two vertical pipes. On most keyboards, the pipe is found above the backslash key; typing it requires that you also hold down the Shift key.

For example, if you want to display a message indicating an invalid age when the variable is less than 0 or is 120 or greater, you can use either code sample in Figure 4-14.

```
// using ||
 if(age < 0 || age >= 120)
   Console.WriteLine("Age is not valid");
// using nested ifs
 if(age < 0)
   Console.WriteLine("Age is not valid");
 else
   if(age >= 120)
      Console.WriteLine("Age is not valid");
```

Figure 4-14 Comparison of the `||` operator and nested `if` statements

A common use of the `||` operator is to decide to take action whether a character variable is uppercase or lowercase. For example, in the following decision, any subsequent action occurs whether the selection variable holds an uppercase or lowercase 'A':

```
if(selection == 'A' || selection == 'a')...
```

Watch the video *Using the AND and OR Operators.*

When the `||` operator is used in an `if` statement, only one of the two Boolean expressions in the tested expression needs to be `true` for the resulting action to occur. As with the `&&` operator, this feature is called short-circuit evaluation. When you use the `||` operator and the first Boolean expression is `true`, the second expression is never evaluated, because it does not matter whether it is `true` or `false`.

Using the Logical AND and OR Operators

The **Boolean logical AND operator** (`&`) and **Boolean logical inclusive OR operator** (`|`) work just like their `&&` and `||` (*conditional* AND and OR) counterparts, except they do not support short-circuit evaluation. That is, they always evaluate both sides of the expression, no matter what the first evaluation is. This can lead to a **side effect**, or unintended consequence. For example, in the following statement that uses `&&`, if `salesAmountForYear` is not at least 10,000, the first half of the expression is `false`, so the second half of the Boolean expression is never evaluated and `yearsOfService` is not increased.

```
if(salesAmountForYear >= 10000 && ++yearsOfService > 10)
   bonus = 200;
```

On the other hand, when a single `&` is used and `salesAmountForYear` is not at least 10,000, then even though the first half of the expression is `false`, the second half is still evaluated, and `yearsOfService` is increased:

```
if(salesAmountForYear >= 10000 & ++yearsOfService > 10)
   bonus = 200;
```

Because the first half of the expression is `false`, the entire evaluation is `false`, and, as with `&&`, `bonus` is still not set to 200. However, a side effect has occurred: `yearsOfService` is incremented.

In general, you should avoid writing expressions that contain side effects. If you want `yearsOfService` to increase no matter what the `salesAmountForYear` is, then you should increase it in a stand-alone statement before the decision is made, and if you want it increased only when the sales amount exceeds 10,000, then you should increase it in a statement that depends on that decision.

The & and | operators are Boolean logical operators when they are placed between Boolean expressions. When the same operators are used between integer expressions, they are **bitwise operators** that are used to manipulate the individual bits of values.

Combining AND and OR Operators

You can combine as many AND and OR operators in an expression as you need. For example, when three conditions must be `true` before performing an action, you can use an expression such as `if(a && b && c)`. When you combine `&&` and `||` operators within the same Boolean expression, the `&&` operators take precedence, meaning their Boolean values are evaluated first.

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

```
if(age <= 12 || age >= 65 && rating == 'G')
   Console.WriteLine("Discount applies");
```

To keep the comparisons simple, this example assumes that movie ratings are always a single character.

For example, assume a movie patron is 10 years old and the movie rating is 'R'. The patron should not receive a discount (or be allowed to see the movie!). However, within the `if` statement above, the compound expression `age >= 65 && rating == 'G'` evaluates first. It is `false`, so the `if` becomes the equivalent of `if(age <= 12 || false)`. Because `age <= 12` is `true`, the `if` becomes the equivalent of `if(true || false)`, which evaluates as `true`, and the statement "Discount applies" incorrectly displays.

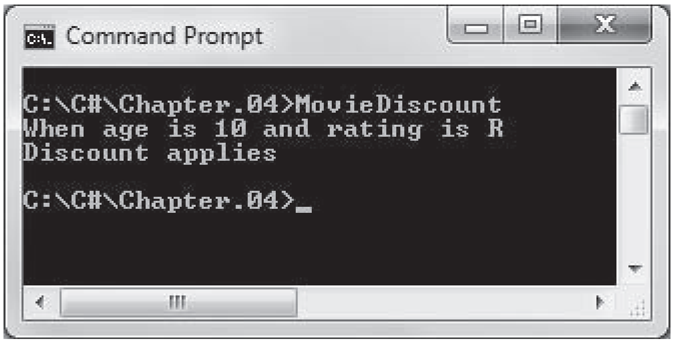
You can use parentheses to correct the logic and force the expression `age <= 12 || age >= 65` to evaluate first, as shown in the following code.

```
if((age <= 12 || age >= 65) && rating == 'G')
   Console.WriteLine("Discount applies");
```

With the added parentheses, if `age` is 12 or less OR 65 or greater, the expression is evaluated as `if(true && rating == 'G')`. When the `age` value qualifies a patron for a discount, then the `rating` value must also be acceptable. Figure 4-15 shows the `if` within a complete program; note that the discount age limits now are represented as named constants. Figure 4-16 shows the execution before the parentheses were added to the `if` statement, and Figure 4-17 shows the output after the inclusion of the parentheses.

```
using System;
public class MovieDiscount
{
   public static void Main()
   {
      int age = 10;
      char rating = 'R';
      const int CHILD_AGE = 12;
      const int SENIOR_AGE = 65;
      Console.WriteLine("When age is {0} and rating is {1}",
        age, rating);
      if((age <= CHILD_AGE || age >= SENIOR_AGE) &&
        rating == 'G')
        Console.WriteLine("Discount applies");
      else
        Console.WriteLine("Full price");
   }
}
```

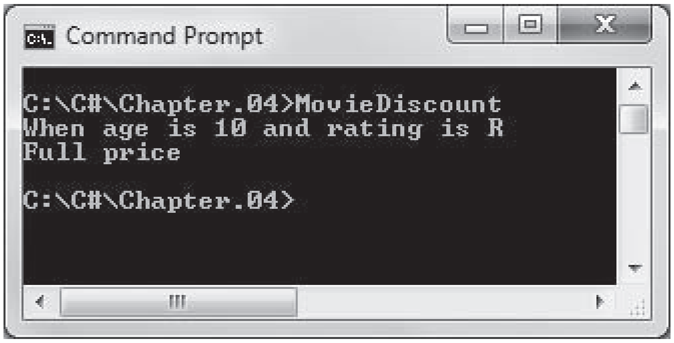
Figure 4-15 Movie ticket discount program using parentheses to alter precedence of Boolean evaluations



```
C:\C#\Chapter.04>MovieDiscount
When age is 10 and rating is R
Discount applies

C:\C#\Chapter.04>_
```

Figure 4-16 Incorrect results when `MovieDiscount` program is executed without added parentheses



```
C:\C#\Chapter.04>MovieDiscount
When age is 10 and rating is R
Full price

C:\C#\Chapter.04>
```

Figure 4-17 Correct results when parentheses are added to `MovieDiscount` program

You can use parentheses for clarity even when they are not required. For example, the following expressions both evaluate `a && b` first:

```
a && b || c
(a && b) || c
```

If the version with parentheses makes your intentions clearer, you should use it.

In Chapter 2, you controlled arithmetic operator precedence by using parentheses. Appendix A describes the precedence of every C# operator. For example, in Appendix A you can see that the comparison operators <= and >= have higher precedence than both && and ||.

Watch the video *Combining AND and OR Operations*.

TWO TRUTHS & A LIE

Using Compound Expressions in `if` Statements

1. If `a` is `true` and `b` and `c` are `false`, then the value of `b && c || a` is `true`.
2. If `d` is `true` and `e` and `f` are `false`, then the value of `e || d && f` is `true`.
3. If `g` is `true` and `h` and `i` are `false`, then the value of `g || h && i` is `true`.

The false statement is #2. If `d` is `true` and `e` and `f` are `false`, then the value of `e || d && f` is `false`. Because you evaluate && before ||, first `d && f` is evaluated and found to be `false`, then `e || false` is evaluated and found to be `false`.

Making Decisions 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 different strings based on a student's class year. Figure 4-18 shows the logic using nested `if` statements. The program segment tests the `year` variable four times and executes one of four statements, or displays an error message.

```
if(year == 1)
   Console.WriteLine("Freshman");
else
   if(year == 2)
      Console.WriteLine("Sophomore");
   else
      if(year == 3)
        Console.WriteLine("Junior");
      else
        if(year == 4)
          Console.WriteLine("Senior");
        else
          Console.WriteLine("Invalid year");
```

Figure 4-18 Executing multiple alternatives using a series of `if` statements

An alternative to the series of nested `if` statements in Figure 4-18 is to use the `switch` structure (see Figure 4-19). The **`switch` structure** tests a single variable against a series of exact matches. The `switch` structure in Figure 4-19 is easier to read and interpret than the series of nested `if` statements in Figure 4-18. The `if` statements would become harder to read if additional choices were required and if multiple statements had to execute in each case. These additional choices and statements might also make it easier to make mistakes.

The `switch` statement is not as flexible as the `if` because you can test only one variable, and it must be tested for equality.

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

Figure 4-19 Executing multiple alternatives using a `switch` statement

You are not required to list the `case` label values in ascending order, as shown in Figure 4-19. However, doing so can make the statement easier for a reader to follow. From the computer's perspective, it is most efficient to list the most common case first; often, the `default` case is the most common.

The switch structure uses four new keywords:

- The keyword **switch** starts the structure and is followed immediately by a test expression (called the **switch expression**) enclosed in parentheses.
- The keyword **case** is followed by one of the possible values that might equal the switch expression. A colon follows the value. The entire expression—for example, case 1:—is a case label. A **case label** identifies a course of action in a switch structure. Most switch structures contain several case labels.
- The keyword **break** usually terminates a switch structure at the end of each case. Although other statements can end a case, break is the most commonly used.
- The keyword **default** optionally is used prior to any action that should occur if the test expression does not match any case.

Instead of break, you can use a return statement or a throw statement to end a case. You learn about return statements in the chapter *Introduction to Methods* and throw statements in the chapter *Exception Handling*.

The switch structure shown in Figure 4-19 begins by evaluating the year variable. If year is equal to the first case label value, which is 1, then the statement that displays "Freshman" executes. The break statement causes a bypass of 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 label value of 1, then the next case label value is compared, and so on. If the year variable does not contain the same value as any of the case label expressions, then the default statement or statements execute.

The **governing type** of a switch statement is established by the switch expression. The governing type can be sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or an enum type. You learned about enum types in Chapter 2.

In C#, an error occurs if you reach the end point of the statement list of a switch section. For example, the following code is not allowed because when the year value is 1, "Freshman" is displayed, and the code reaches the end of the case.

```
switch(year)
{
   case 1:
      Console.WriteLine("Freshman");
   case 2:
      Console.WriteLine("Sophomore");
      break;
}
```

This code is invalid because the end of the case is reached after “Freshman” is displayed.

Not allowing code to reach the end of a case is known as the “no fall through rule.” In several other programming languages, such as Java and C++, when year equals 1, both "Freshman" and "Sophomore" would be displayed. However, falling through to the next case is not allowed in C#.

A switch structure does not need to contain a default case. If the test expression in a switch does not match any of the case label values, and there is no `default` value, then the program simply continues with the next executable statement. However, it is good programming practice to include a `default` label in a `switch` structure; that way, you provide for actions when your data does not match any case. The `default` label does not have to appear last, although usually it does.

You receive a compiler error if two or more `case` label values in a `switch` statement are the same.

You can use multiple labels to govern a list of statements. For example, in the code in Figure 4-20, `"Upperclass"` is displayed whether the `year` value is 3 or 4.

```
switch(year)
{
   case 1:
      Console.WriteLine("Freshman");
      break;
   case 2:
      Console.WriteLine("Sophomore");
      break;
   case 3:
   case 4:
      Console.WriteLine("Upperclass");
      break;
   default:
      Console.WriteLine("Invalid year");
      break;
}
```

Cases 3 and 4 are both "Upperclass".

Figure 4-20 Example `switch` structure using multiple labels to execute a single statement block

Using a `switch` structure is never required; you can always achieve the same results with nested `if` statements. The `switch` structure is simply a convenience you can use when there are several alternative courses of action depending on a match with a variable. Additionally, it makes sense to use a `switch` only when there are a reasonable number of specific matching values to be tested. For example, if every sale amount from $1 to $500 requires a 5% commission, it is not reasonable to test every possible dollar amount using the following code:

```
switch(saleAmount)
{
   case 1:
      commRate = .05;
      break;
   case 2:
      commRate = .05;
      break;
   case 3:
      commRate = .05;
      break;
//...and so on for several hundred more cases
```

With 500 different dollar values resulting in the same commission, one test—if(saleAmount <= 500)—is far more reasonable than listing 500 separate cases.

Using an Enumeration with a switch Statement

Using an enumeration with a switch structure can often be convenient. Recall from Chapter 2 that an enumeration allows you to apply values to a list of constants. For example, Figure 4-21 shows a program that uses an enumeration to represent major courses of study at a college. Suppose that students who are Accounting, CIS, or Marketing majors are in the Business Division of the college, and English or Math majors are in the Humanities Division. The program shows how the enumeration values can be used in a switch structure. In the shaded switch control statement, notice how the input integer is cast to an enumeration value. Figure 4-22 shows a typical execution of the program.

In the enumeration list in Figure 4-21, ACCOUNTING is assigned 1, so the other values in the list are 2, 3, 4, and 5 in order.

```
using System;
public class DivisionBasedOnMajor
{
   enum Major
   {
      ACCOUNTING = 1, CIS, ENGLISH, MATH, MARKETING
   }
   public static void Main()
   {
      int major;
      Console.Write("Enter major code >> ");
      major = Convert.ToInt32(Console.ReadLine());
      switch ((Major) major)
      {
        case Major.ACCOUNTING:
        case Major.CIS:
        case Major.MARKETING:
          Console.WriteLine("Major is in the Business Division");
          break;
        case Major.ENGLISH:
        case Major.MATH:
          Console.WriteLine("Major is in the Humanities Division");
          break;
        default:
          Console.WriteLine("Department number is invalid");
          break;
      }
   }
}
```

Figure 4-21 The DivisionBasedOnMajor class