CHAPTER 4

Making Decisions

In this chapter you will:

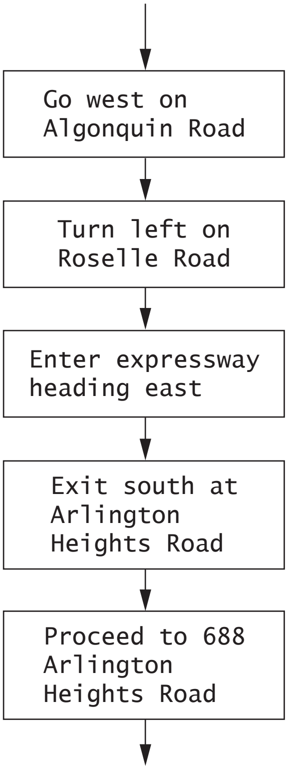
- Understand logic-planning tools and decision making
- Learn how to make decisions using `if` statements
- Learn how to make decisions using `if-else` statements
- Use compound expressions in `if` statements
- Make decisions using `switch` statements
- Use the conditional operator
- Use the NOT operator
- 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. 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 logic of a given task in everyday language and not the syntax used in a programming language. (You learned the difference between *logic* and *syntax* in Chapter 1.) 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 or *flowlines*.

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, diamonds, and connecting flowlines. You can use a rectangle to represent any process or 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. Figure 4-1 illustrates a logical structure called a **sequence structure**—one step follows another unconditionally. A sequence structure might contain any number of steps, but one task follows another with no chance to branch away or skip a step.




```
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 of a sequence structure

Sometimes, logical steps do not follow in an unconditional sequence. Instead, 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 corner of the diamond. Figure 4-2 shows a flowchart describing directions in which the execution of some steps depends on a decision.

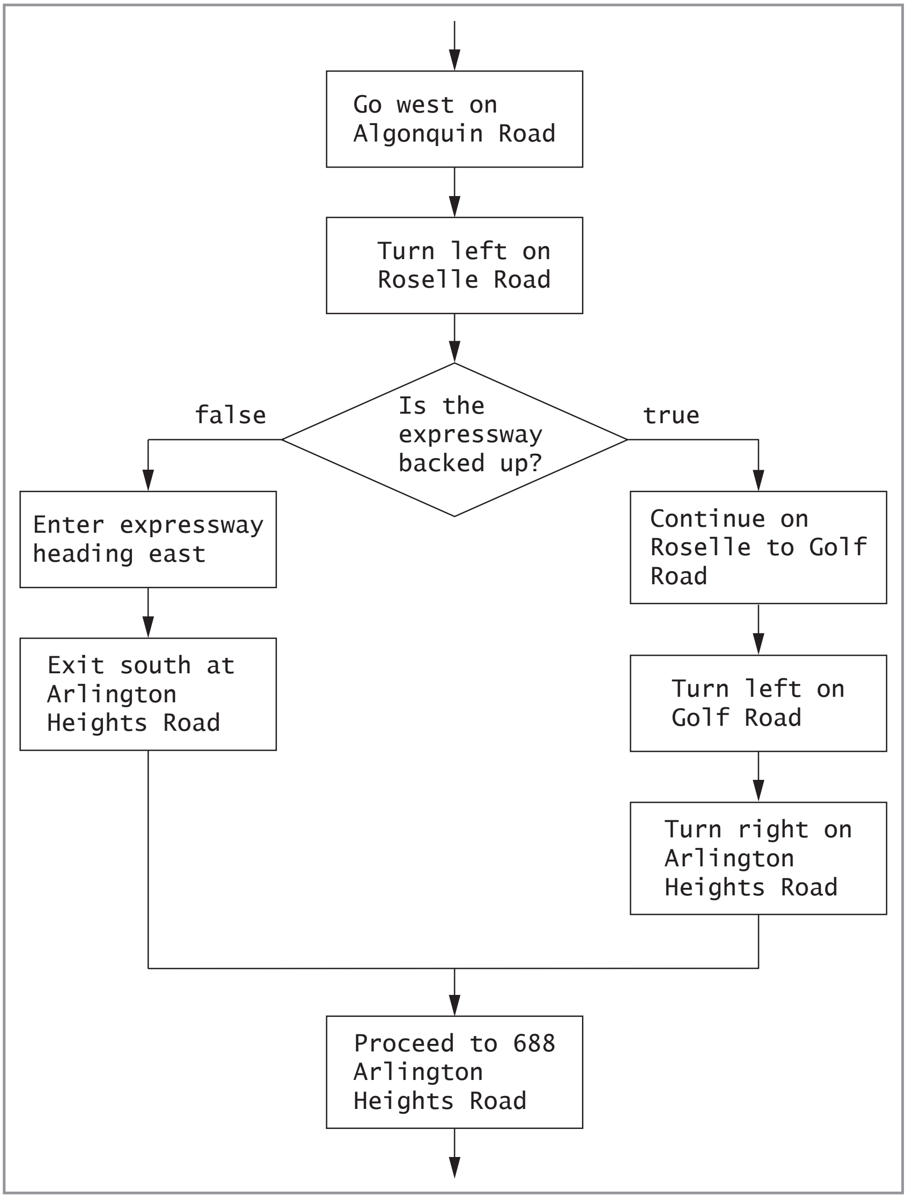


Figure 4-2 Flowchart including a decision structure

Figure 4-2 shows a **decision structure**—one that involves choosing between alternative courses of action based on a value. 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. For example, program code that you write never includes a question like "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

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;
```

In this statement, *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. You can leave a space between the keyword `if` and the opening parenthesis if you think that format is easier to read.

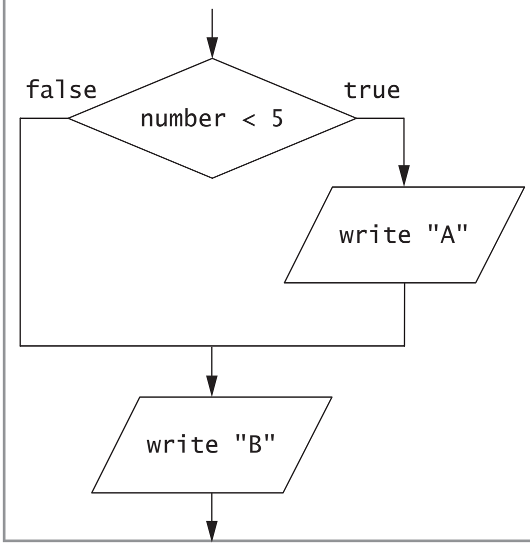
Usable expressions in an `if` statement include Boolean expressions such as `amount > 5` and `month == "May"` as well as the value of `bool` variables such as `isValidIDNumber`. If the expression evaluates as `true`, then the statement executes. Whether the expression evaluates as `true` or `false`,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

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

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.

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, but when `number` is 5 or greater, only *B* is displayed. When the tested Boolean expression `number < 5` is `false`, the statement that displays *A* never executes.




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

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

Although you can create a meaningful flowchart using only rectangles, diamonds, and flowlines, parallelograms have traditionally been used to represent input and output statements, so they are used in Figure 4-3 and in other figures in this chapter.

In the code 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 `WriteLine("A");`. If you incorrectly insert a semicolon at the end of `if(number < 5)`, then the code means, "If `number` is less than 5, do nothing; then, no matter what the value of `number` is, display *A*." Figure 4-4 shows the flowchart logic that matches the code when a semicolon is incorrectly placed at the end of the `if` expression.

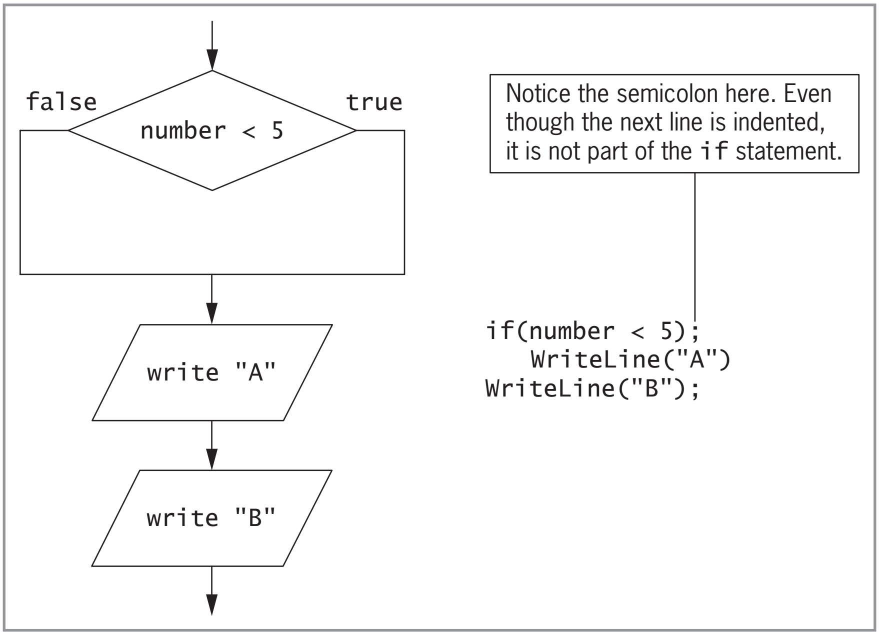


Figure 4-4 Flowchart and code including an `if` statement with a semicolon following the `if` expression

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` statement written on a single line; the second shows an `if` statement on two lines but with no indentation. The third uses conventional indentation. All three examples execute identically.

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

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

When you want to execute two or more statements conditionally, you 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. The `if` expression that precedes the block is the **control statement** for the decision structure.

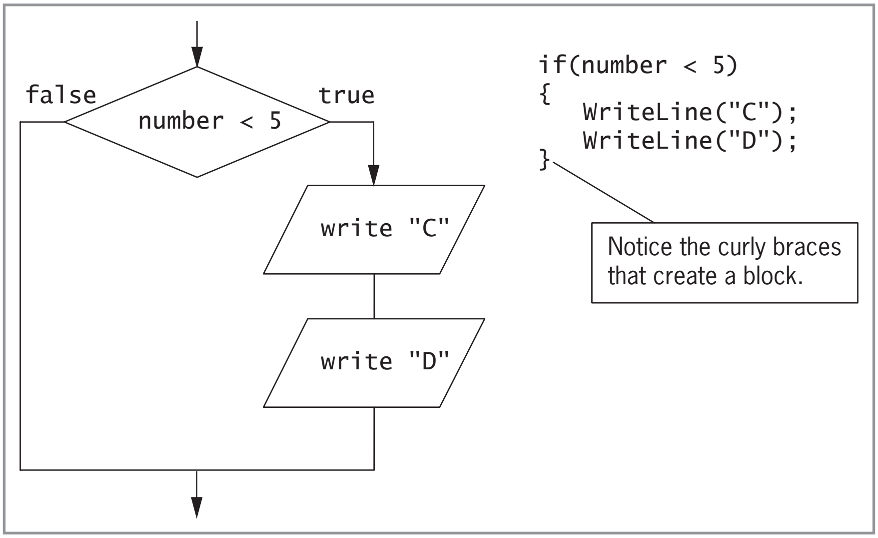


Figure 4-5 Flowchart and code including a typical `if` statement containing a block

Indenting alone does not cause multiple statements to depend on the evaluation of a Boolean expression following an `if` control statement. 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`; however, only the first one does, as shown in the flowchart, because the statements are not blocked.

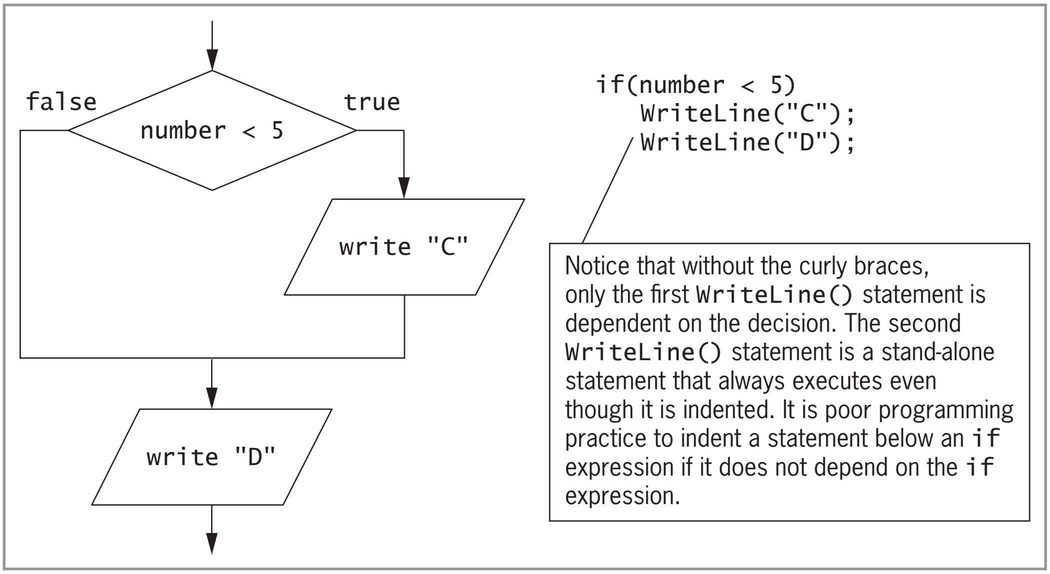
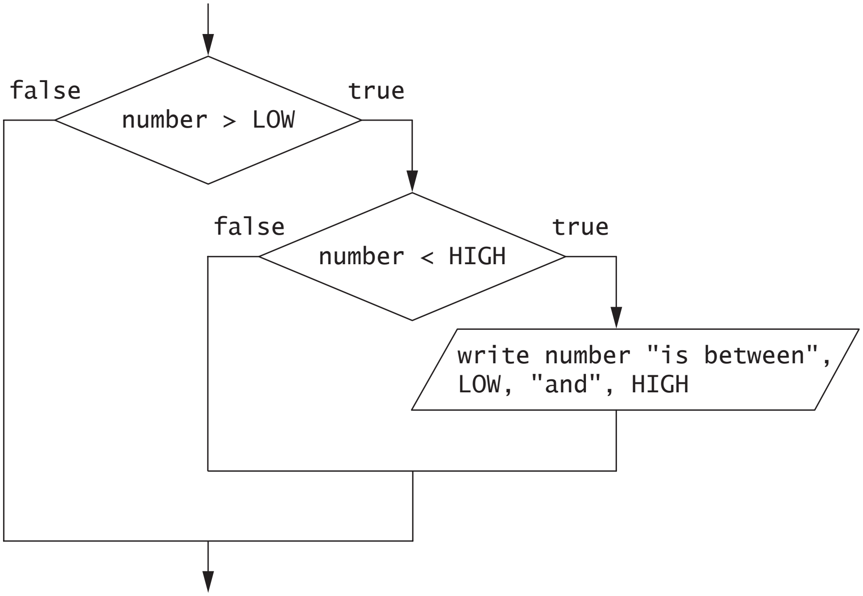


Figure 4-6 Flowchart and code including an `if` statement that is missing curly braces or that has inappropriate indenting

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 because when a program later is modified to include multiple statements that depend on the `if`, the braces serve as a reminder to use a block. Creating a block that contains no statements also is legal in C#. You usually do so only when starting to write a program, as a reminder to yourself to add statements later.

> In C#, it is customary to align a block's opening and closing braces. 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.

You can place any number of statements within a block, including other `if` statements. 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 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)
      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 in Figure 4-7. When a user enters a number greater than 5, the first tested expression is `true` and the `if` statement that tests whether the number is less than 10 executes. When the second tested expression also is `true`, the `WriteLine()` statement executes. If either the first or second tested expression is `false`, 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.

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

Figure 4-8 Program using nested `if`

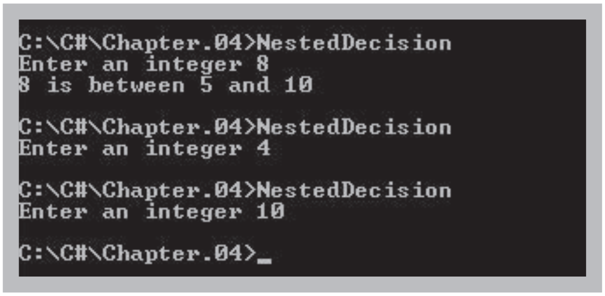


Figure 4-9 Output of three executions of the `NestedDecision` program

A Note on Equivalency Comparisons

Often, programmers mistakenly use a single equal sign rather than the double equal sign when testing for 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 an assignment is used between the parentheses in an `if` statement, as in `if(number = HIGH)`, the assignment is illegal, and the program will not compile.

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 that 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 assigned value. This is not a recommended way to make a comparison because it looks confusing. If your intention was to assign a value to `doesEmployeeHaveDependents` and to make a decision based on the value, then your intentions would be clearer with the following code:

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

Notice the difference in the following statement that uses two equal signs within the parentheses in the `if` statement:

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

This statement compares `doesEmployeeHaveDependents` to the result of comparing `numDependents` to 0 but does not change the value of `doesEmployeeHaveDependents`.

One of the many advantages of using the Visual Studio IDE to write programs is that if you use an assignment operator in place of an equivalency operator in a Boolean expression, your mistake will be flagged as an error immediately.

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 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:

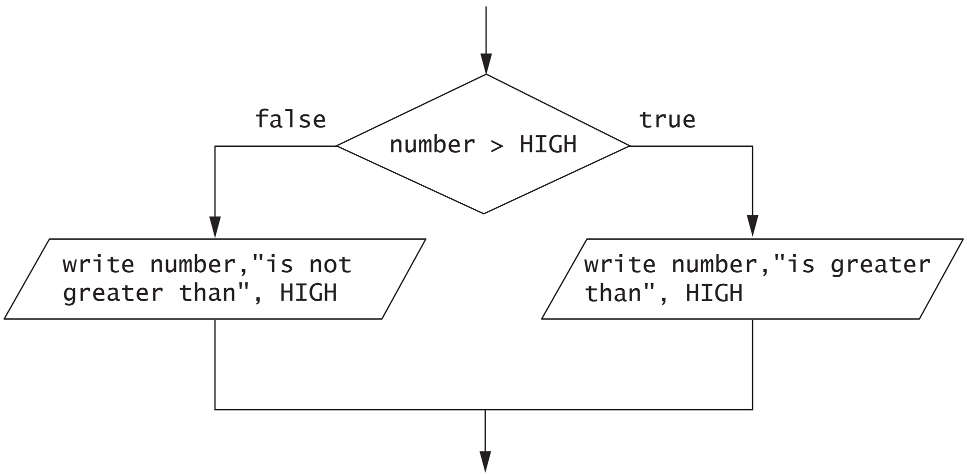
```
if(expression)
   statement1;
else
   statement2;
```

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, the following code assigns 0 to `bonus` and also produces a line of output when the Boolean variable `isProjectUnderBudget` is `false`:

```
if(isProjectUnderBudget)
   bonus = 200;
else
{
   bonus = 0;
   WriteLine("Notify contractor");
}
```

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. The indentation shown in the `if-else` example in Figure 4-11 is not required, but it is standard. You vertically align the keyword `if` with the keyword `else`, and then indent the action statements that depend on the evaluation. Figure 4-12 shows two executions of the program.




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

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

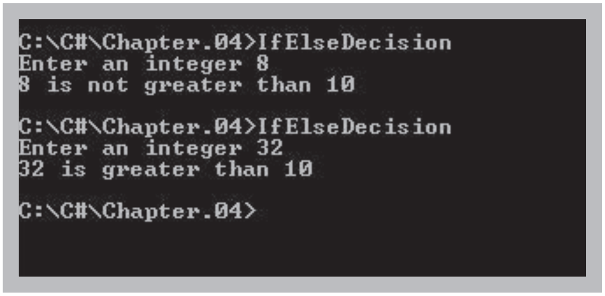


Figure 4-12 Output of two executions of the `IfElseDecision` program

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`. Correct indentation helps to make this clear.

```
if(saleAmount > 1000)
   if(saleAmount < 2000)
      bonus = 100;
   else
      bonus = 500;
```

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 $500 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 might be 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`.

You Do It

Using **`if-else`** *Statements*

In the next steps, you write a program that requires using multiple, nested `if-else` statements to accomplish its goal—determining whether any of the three integers entered by a user are equal.

1. Open a new file or project named **CompareThreeNumbers,** and write the first lines necessary for the class:

```
using System;
using static System.Console;
class CompareThreeNumbers
{
```

2. Begin a `Main()` method by declaring a string for input and three integers that will hold the input values.

```
static void Main()
{
   string numberString;
   int num1, num2, num3;
```

3. Add the statements that retrieve the three integers from the user and assign them to the appropriate variables.

```
Write("Enter an integer ");
numberString = ReadLine();
num1 = Convert.ToInt32(numberString);
Write("Enter an integer ");
numberString = ReadLine();
num2 = Convert.ToInt32(numberString);
Write("Enter an integer ");
numberString = ReadLine();
num3 = Convert.ToInt32(numberString);
```

In the chapter "Introduction to Methods," you will learn to write methods, which will allow you to avoid repetitive code like that shown here.

(continues)

(continued)

4. If the first number and the second number are equal, there are two possibilities: either the first is also equal to the third, in which case all three numbers are equal, or the first is not equal to the third, in which case only the first two numbers are equal. Insert the following code:

```
if(num1 == num2)
   if(num1 == num3)
      WriteLine("All three numbers are equal");
   else
      WriteLine("First two are equal");
```

5. If the first two numbers are not equal, but the first and third are equal, display an appropriate message. For clarity, the `else` should vertically align under the first `if` statement that compares `num1` and `num2`.

```
else
   if(num1 == num3)
      WriteLine("First and last are equal");
```

6. When `num1` and `num2` are not equal, and `num1` and `num3` are not equal, but `num2` and `num3` are equal, display an appropriate message. For clarity, the `else` should vertically align under the statement that compares `num1` and `num3`.

```
else
   if(num2 == num3)
      WriteLine("Last two are equal");
```

7. Finally, if none of the pairs (`num1` and `num2`, `num1` and `num3`, or `num2` and `num3`) is equal, display an appropriate message. For clarity, the `else` should vertically align under the statement that compares `num2` and `num3`.

```
else
   WriteLine("No two numbers are equal");
```

8. Add a closing curly brace for the `Main()` method and a closing curly brace for the class.

9. Save and compile the program, and then execute it several times, providing different combinations of equal and nonequal integers when prompted. Figure 4-13 shows several executions of the program.

(continues)

(continued)

```
C:\C#\Chapter.04>CompareThreeNumbers
Enter an integer 4
Enter an integer 16
Enter an integer 8
No two numbers are equal

C:\C#\Chapter.04>CompareThreeNumbers
Enter an integer 12
Enter an integer 45
Enter an integer 12
First and last are equal

C:\C#\Chapter.04>CompareThreeNumbers
Enter an integer 67
Enter an integer 5
Enter an integer 5
Last two are equal

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

Figure 4-13 Several executions of the `CompareThreeNumbers` program

Using Compound Expressions in `if` Statements

In many programming situations, you need to make multiple decisions before taking action. No matter how many decisions must be made, you can produce the correct results by using a series of `if` statements. For convenience and clarity, however, you can combine multiple decisions into a single, compound Boolean expression using a combination of conditional AND and OR operators.

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 expressions that use && and two operands. 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 4-14 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 the && operator
if(age >= 0 && age < 120)
   WriteLine("Age is valid");

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

Figure 4-14 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 `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;
```

Don't Do It
< 5000 is not a Boolean expression because the < operator requires two operands, 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;
```

In this example, the `if` clause has a set of parentheses (the first opening parenthesis and the last closing parenthesis), and each Boolean expression that is part of the compound condition has its own set of parentheses. Use this format if it is clearer to you.

The expressions in each part of a compound, conditional 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)
   WriteLine("Both are greater than the 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.

<table>
<tr><th>x</th><th>y</th><th>x||y</th></tr>
<tr><td>True</td><td>True</td><td>True</td></tr>
<tr><td>True</td><td>False</td><td>True</td></tr>
<tr><td>False</td><td>True</td><td>True</td></tr>
<tr><td>False</td><td>False</td><td>False</td></tr>
</table>

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

For example, if you want to display a message indicating an invalid age when the value of an `age` variable is less than 0 or more than 120, you can use either code sample in Figure 4-15.

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.

```
// using the || operator
if(age < 0 || age > 120)
   WriteLine("Age is not valid");

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

Figure 4-15 Comparison of the `||` operator and nested `if` statements

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. When the first Boolean expression is `true`, the second expression is never evaluated because it doesn't matter whether the second expression is `true` or `false`. As with the && operator, this feature is called *short-circuit evaluation*.

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

Using the Logical AND and OR Operators

C# provides two logical operators that you generally do *not* want to use when making comparisons. However, you should learn about them because they might be used in programs written by others, and because you might use one by mistake when you intend to use a conditional operator.

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`, and `bonus` is not set to 200, the second half of the expression is still evaluated, and `yearsOfService` is always increased whether it is more than 10 or not:

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

In general, you should avoid writing expressions that contain side effects. If you want `yearsOfService` to increase no matter what the value of `salesAmountForYear` is, then you should increase it in a stand-alone statement. 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 multiple AND or OR operators in the same expression. For example, in the following statement, all three Boolean variables must be true to produce the output:

```
if(isWeekendDay && isOver80Degrees && isSunny)
   WriteLine("Good day for the beach");
```

In the following statement, only one of the three Boolean variables needs to be true to produce the output:

```
if(isWeekendDay || isHoliday || amSick)
   WriteLine("No work today");
```

When you use a series of only && or only `||` operators in an expression, they are evaluated from left to right as far as is necessary to determine the value of the entire expression. However, when you combine && and `||`operators within the same Boolean expression, they are not necessarily evaluated from left to right. Instead, the && operators take precedence, meaning they are evaluated first. For example, consider a program that determines whether a movie theater patron can purchase a discounted ticket. Assume that discounts are allowed for children (age 12 and younger) and for senior citizens (age 65 and older) who attend G-rated movies. (To keep the comparisons simple, this example assumes that movie ratings are always a single character.) The following code looks reasonable, but it produces incorrect results because the && evaluates before the `||`.

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

For example, suppose that a movie patron is 10 years old and the movie rating is *R*. The patron should not receive a discount (or be allowed to see the movie!). However, within the `if` statement 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* is displayed, which is not the intention for a 10-year-old seeing an R-rated movie.

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')
   WriteLine("Discount applies");
```

With the added parentheses, both `age` comparisons are made first. If the `age` value qualifies a patron for a discount, the expression is evaluated as `if(true && rating == 'G')`. Then the `rating` value must also be acceptable for the message to be displayed. Figure 4-16 shows the `if` statement within a complete program; note that the discount age limits are represented as named constants in the complete program. Figure 4-17 shows the execution before the parentheses were added to the `if` statement, and Figure 4-18 shows the output after the inclusion of the parentheses.

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

Figure 4-16 Movie ticket discount program using parentheses to alter precedence of Boolean evaluations

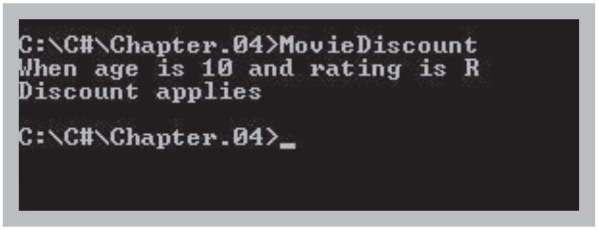


Figure 4-17 Incorrect results when `MovieDiscount` program is executed without added parentheses

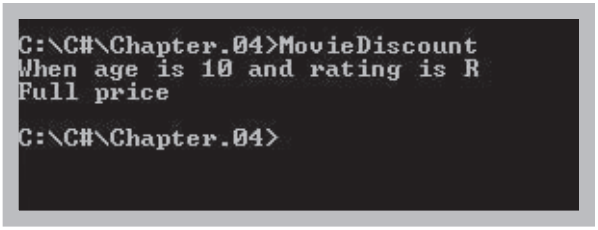


Figure 4-18 Correct results when parentheses are added to `MovieDiscount` program as shown in Figure 4-16

You can use parentheses for clarity in a Boolean expression, 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 learned that parentheses also control arithmetic operator precedence. Appendix A describes the precedence of every C# operator, which is important to understand in complicated expressions. For example, in Appendix A you can see that the arithmetic and relational operators have higher precedence than && 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`.

You Do It

Using AND and OR Logic

In the next steps, you create an interactive program that allows you to test AND and OR logic for yourself. The program decides whether a delivery charge applies to a shipment. If the customer lives in Zone 1 or Zone 2, then shipping is free, as long as the order contains fewer than 10 boxes. If the customer lives in another zone or if the order is too large, then a delivery charge applies.

1. Open a new file named **DemoORAndAND**, and then enter the first few lines of the program. Define constants for `ZONE1`, `ZONE2`, and the `LOW_QUANTITY` limit as well as variables to hold the customer's input string, which will be converted to the zone and number of boxes in the shipment.

```
using System;
using static System.Console;
class DemoORAndAND
{
   static void Main()
   {
      const int ZONE1 = 1, ZONE2 = 2;
      const int LOW_QUANTITY = 10;
      string inputString;
      int quantity;
      int deliveryZone;
```

2. Enter statements that describe the delivery charge criteria to the user and accept keyboard values for the customer's delivery zone and shipment size.

```
WriteLine("Delivery is free for zone {0} or {1}",
   ZONE1, ZONE2);
WriteLine("when the number of boxes is less than {0}",
   LOW_QUANTITY);
WriteLine("Enter delivery zone ");
inputString = ReadLine();
deliveryZone = Convert.ToInt32(inputString);
WriteLine("Enter the number of boxes in the shipment");
inputString = ReadLine();
quantity = Convert.ToInt32(inputString);
```

(continues)

(continued)

3. Write a compound `if` statement that tests whether the customer lives in Zone 1 or 2 and has a shipment consisting of fewer than 10 boxes. Notice that the first two comparisons joined with the `||` operator are contained in their own set of nested parentheses.

```
if((deliveryZone == ZONE1 || deliveryZone == ZONE2) &&
   quantity < LOW_QUANTITY)
        WriteLine("Delivery is free");
else
   WriteLine("A delivery charge applies");
```

4. Add closing curly braces for the `Main()` method and for the class, and save the program. Compile and execute the program. Enter values for the zone and shipment size. Figure 4-19 shows the output.

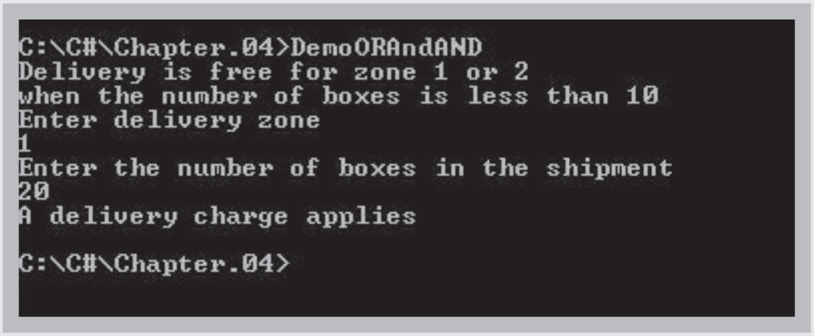


Figure 4-19 Sample execution of `DemoORAndAND` program

5. To demonstrate the importance of the nested parentheses in the `if` statement, remove the inner set of parentheses that surround the expression `deliveryZone == ZONE1 || deliveryZone == ZONE2` in the `Main()` method. Save the new version of the program, and compile it. When you execute this version of the program, the output indicates that any delivery to `ZONE1` is free, but it should not be. The way the `if` statement is currently constructed, as soon as `deliveryZone == zone1` is `true`, the rest of the Boolean expression is not even evaluated. Reinstate the parentheses, and then save and compile the program. Execute it, and confirm that the output is again correct.

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 that you want to display different strings based on a student's class year. Figure 4-20 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)
   WriteLine("Freshman");
else
   if(year == 2)
      WriteLine("Sophomore");
   else
      if(year == 3)
         WriteLine("Junior");
      else
         if(year == 4)
            WriteLine("Senior");
         else
            WriteLine("Invalid year");
```

Figure 4-20 Executing multiple alternatives using a series of `if` statements

An alternative to the series of nested `if` statements in Figure 4-20 is to use the `switch` structure (see Figure 4-21). The **`switch` structure** tests a single variable against a series of exact matches. The `switch` structure is sometimes called the *case structure* or the *switch-case structure*. The `switch` structure in Figure 4-21 is easier to read and interpret than the series of nested `if` statements in Figure 4-20. 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 increase the potential to make mistakes.

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

Figure 4-21 Executing multiple alternatives using a `switch` statement

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 value that follows `case` is the **governing type** of the `switch` statement; this value can be `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `string`, or an `enum` type. You learned about `enum` types in Chapter 2.
- 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 4-21 begins by evaluating the `year` variable. If `year` is equal to any `case` label value, then the output statement for that case executes. The `break` statements that follow each output statement cause a bypass of other cases. If `year` does not contain the same value as any of the `case` label expressions, then the `default` statement or statements execute.

You are not required to list the `case` label values in ascending order, as shown in Figure 4-21, but doing so can make the statement easier for a reader to follow. You can even list the `default` case first, although usually it is listed last. You receive a compiler error if two or more `case` label values are the same in a `switch` statement.

In C#, an error occurs if you reach the end point of the statement list of a `case` 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`. The problem could be fixed by inserting a `break` statement before `case 2`.

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

Don't Do It
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++, if you write a `case` without a `break` statement, the subsequent `case`s execute until a `break` is encountered. For example, in the code above, both *Freshman* and *Sophomore* would be displayed when `year` is 1. 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.

In C#, it is legal for a `case` to contain no list of statements. This feature allows you to use multiple labels to govern a list of actions. For example, in the code in Figure 4-22, *Upperclass* is displayed whether the `year` value is 3 or 4.

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

Cases 3 and 4 are both “Upperclass”.

Figure 4-22 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 `if` statements. The `switch` statement is not as flexible as the `if` statement because you can test only one variable, and it must be tested for equality. 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 percent commission, it is not reasonable to test every possible dollar amount using the following code:

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

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 4-23 shows a program that uses an enumeration to represent major courses of study at a college. In the enumeration list in Figure 4-23, `ACCOUNTING` is assigned 1, so the other values in the list are 2, 3, 4, and 5 in order. 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 program, the user enters an integer. Next, in the shaded `switch` control statement, the input integer is cast to an enumeration value. Then, enumeration values become the governing types for each case. For someone reading the code, the purposes of `enum` values such as `ACCOUNTING` and `CIS` are clearer than their integer equivalents would be. Figure 4-24 shows a typical execution of the program.

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

Figure 4-23 The `DivisionBasedOnMajor` class

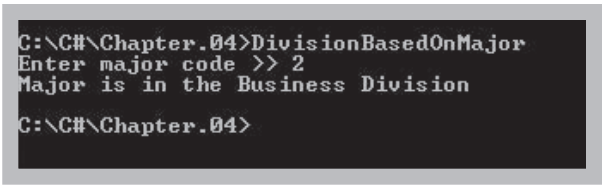


Figure 4-24 Typical execution of the `DivisionBasedOnMajor` program

TWO TRUTHS & A LIE

Making Decisions Using the `switch` Statement

1. In a `switch` statement, the keyword `case` is followed by one of the possible values that might equal the `switch` expression, and a colon follows the value.
2. The keyword `break` always terminates a `switch` structure at the end of each case.
3. A `switch` statement does not need to contain a `default` case.

The false statement is #2. The keyword `break` typically is used to terminate a `switch` structure at the end of each `case`, but other statements can end a `case`.

Using the Conditional Operator

The **conditional operator** is used as an abbreviated version of the `if-else` statement; it requires three expressions separated with a question mark and a colon. Like the `switch` structure, using the conditional operator is never required. Rather, it is simply a convenient shortcut, especially when you want to use the result immediately as an expression. The syntax of the conditional operator is:

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

Unary operators use one operand; binary operators use two. The conditional operator `?:` is **ternary** because it requires three arguments: a test expression and `true` and `false` result expressions. The conditional operator is the only ternary operator in C#.

The first expression, `testExpression`, is evaluated as `true` or `false`. If it is `true`, then the entire conditional expression takes on the value of the expression before the colon (`trueResult`). If the value of the `testExpression` is `false`, then the entire expression takes on the value of the expression following the colon (`falseResult`). For example, consider the following statement:

```
biggerNum = (a > b) ? a : b;
```

This statement evaluates `a > b`. If `a` is greater than `b`, then the entire conditional expression takes the value of `a`, which then is assigned to `biggerNum`. If `a` is not greater than `b`, then the expression assumes the value of `b`, and `b` is assigned to `biggerNum`.

The conditional operator is most often applied when you want to use the result as an expression without creating an intermediate variable. For example, a conditional operator can be used directly in an output statement using either of the following formats:

```
WriteLine((testScore >= 60) ? "Pass" : "Fail");
WriteLine("\{testScore >= 60 ? "Pass" : "Fail"}");
```

In these examples, no variable was created to hold *Pass* or *Fail*. Instead one of the strings was output directly based on the `testScore` comparison. The advantage to the second format (which is new in C# 6.0) is that other variables can easily be included in the same string. For example:

```
WriteLine("\{name}'s status is \{testScore >= 60 ? "Pass" : "Fail"}");
```

If `name` is *Sally* and `testScore` is *62*, the output would be *Sally's status is Pass*.

Conditional expressions can be more difficult to read than `if-else` statements, but they can be used in places where `if-else` statements cannot, such as in method calls.

TWO TRUTHS & A LIE

Using the Conditional Operator

1. If `j = 2` and `k = 3`, then the value of the following expression is 2:

   `int m = j < k ? j : k;`

2. If `j = 2` and `k = 3`, then the value of the following expression is 4:

   `int n = j < k ? j + j : k + k;`

3. If `j = 2` and `k = 3`, then the value of the following expression is 5:

   `int p = j > k ? j + k : j * k;`

The false statement is #3. If `j = 2` and `k = 3`, then the value of the expression `j > k` is false. Therefore 6 (`j * k`) is assigned to `p`.

Using the NOT Operator

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

In Chapter 2 you learned that an exclamation point and equal sign together form the "not equal to" operator. The `!=` operator is a binary operator; it compares two operands. The `!` operator is a unary operator; it reverses the meaning of a single Boolean expression.

For example, suppose that a monthly car insurance premium is $200 if the driver is younger than age 26 and $125 if the driver is age 26 or older. Each of the following `if` statements (which have been placed on single lines for convenience) correctly assigns the premium values:

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

The statements with the `!` operator are somewhat more difficult to read, particularly because they require the double set of parentheses, but the result 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 `bool oldEnough = (age >= 25);` can become part of the relatively easy-to-read expression `if(!oldEnough)...` .

The `!` operator has higher precedence than the `&&` and `||` operators. For example, suppose that you have declared two Boolean variables named `ageOverMinimum` and `ticketsUnderMinimum`. The following expressions are evaluated in the same way:

```
ageOverMinimum && !ticketsUnderMinimum
ageOverMinimum && (!ticketsUnderMinimum)
```

Augustus de Morgan was a 19th-century mathematician who originally observed the following:

- `!(a && b) is equivalent to !a || !b`
- `!(a || b) is equivalent to !(a && b)`

TWO TRUTHS & A LIE

Using the NOT Operator

1. Assume that `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 that `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 that `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`.

Avoiding Common Errors When Making Decisions

New programmers frequently make errors when they first learn to make decisions. As you have seen, the most frequent errors include the following:

- Using the assignment operator (=) instead of the comparison operator (==) when testing for equality
- Inserting a semicolon after the Boolean expression in an `if` statement instead of using it after the entire statement is completed
- Failing to block a set of statements with curly braces when several statements depend on the `if` or the `else` statement
- Failing to include a complete Boolean expression on each side of an && or || operator in an `if` statement

In this section, you will learn to avoid other types of errors with `if` statements. Programmers often make errors at the following times:

- When performing a range check incorrectly or inefficiently
- When using the wrong operator
- When using `!` incorrectly

Performing Accurate and Efficient Range Checks

When new programmers must make a range check, they often introduce incorrect or inefficient code into their programs. A **range check** is a series of `if` statements that determine whether a value falls within a specified range. Consider a situation in which salespeople can receive one of three possible commission rates based on an integer named `saleAmount`. For example, a sale totaling $1000 or more earns the salesperson an 8 percent commission, a sale totaling $500 through $999 earns 6 percent of the sale amount, and any sale totaling $499 or less earns 5 percent. Using three separate `if` statements to test single Boolean expressions might result in some incorrect commission assignments. For example, examine the following code:

```
if(saleAmount >= 1000)
   commissionRate = 0.08;
if(saleAmount >= 500)
   commissionRate = 0.06;
if(saleAmount <= 499)
   commissionRate = 0.05;
```

Don't Do It
Although it was not the programmer's intention, both of the first two if statements are true for any `saleAmount` greater than or equal to 1000.

Using this code, if `saleAmount` is $5000, the first `if` statement executes. The Boolean expression `(saleAmount >= 1000)` evaluates as `true`, and 0.08 is correctly assigned to `commissionRate`. However, the next `if` expression, `(saleAmount >= 500)`, also evaluates as `true`, so the `commissionRate`, which was 0.08, is incorrectly reset to 0.06.

A partial solution to this problem is to add an `else` clause to the statement:

```
if(saleAmount >= 1000)
   commissionRate = 0.08;
else if(saleAmount >= 500)
   commissionRate = 0.06;
else if(saleAmount <= 499)
   commissionRate = 0.05;
```

Don't Do It
If the logic reaches this point, the expression must be true, so it is a waste of time to test this condition.

The last two logical tests in this code are sometimes called `else-if` statements because each `else` and its subsequent `if` are placed on the same line. When the `else-if` format is used to test multiple cases, programmers frequently forego the traditional indentation and align each `else-if` with the others.

With this code, when `saleAmount` is $5000, the expression `(saleAmount >= 1000)` is `true` and `commissionRate` becomes 0.08; then the entire `if` structure ends. When `saleAmount` is not greater than or equal to $1000 (for example, $800), the first `if` expression is `false` and the `else` statement executes and correctly sets `commissionRate` to 0.06.

This version of the code works, but it is somewhat inefficient because it executes as follows:

- When `saleAmount` is at least $1000, the first Boolean test is `true`, so `commissionRate` is assigned .08 and the `if` structure ends.
- When `saleAmount` is under $1000 but at least $500, the first Boolean test is `false`, but the second one is `true`, so `commissionRate` is assigned .06 and the `if` structure ends.
- The only `saleAmount` values that reach the third Boolean test are under $500, so the next Boolean test, `if(saleAmount <= 499)`, is always `true`. When an expression is always `true`, there is no need to evaluate it. In other words, if `saleAmount` is not at least $1000 and is also not at least $500, it must by default be less than or equal to $499.

The improved code is as follows:

```
if(saleAmount >= 1000)
   commissionRate = 0.08;
else if(saleAmount >= 500)
   commissionRate = 0.06;
else
   commissionRate = 0.05;
```

In other words, because this example uses three commission rates, two boundaries should be checked. If there were four rates, there would be three boundaries to check, and so on.

Within a nested `if-else`, processing is most efficient when the first question asked is the one that is most likely to be `true`. In other words, if you know that a large number of `saleAmount` values are over $1000, compare `saleAmount` to that value first. That way, the logic bypasses the rest of the decisions. If, however, you know that most `saleAmount`s are small, processing is most efficient when the first decision is `if(saleAmount < 500)`.

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 $5.65 and when an employee's hourly pay rate is over $60." Because your boss used the word *and* in the request, you might be tempted to write a program statement like the following:

```
if(payRate < 5.65 && payRate > 60)
   WriteLine("Error in pay rate");
```

Don't Do It
This expression can never be `true`.

However, as a single variable, no `payRate` value can ever be both below 5.65 and over 60 at the same time, so the output statement can never execute, no matter what value `payRate` has. In this case, you must write the following statement to display the error message under the correct circumstances:

```
if(payRate < 5.65 || payRate > 60)
   WriteLine("Error in pay rate");
```

Similarly, your boss might request, "Output 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)
   WriteLine("Name is: {0}", name);
```

Don't Do It
This expression can never be `true`.

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 department the employee is in.

The correct statement is:

```
if(department == 1 || department == 2)
   WriteLine("Name is: {0}", name);
```

Using the ! Operator Correctly

Whenever you use negatives, it is easy to make logical mistakes. For example, suppose that your boss says, "Make sure if the sales code is not *A* or *B*, the customer gets a 10 percent discount." You might be tempted to code the following:

```
if(salesCode != 'A' || salesCode != 'B')
   discount = 0.10;
```

Don't Do It
This expression can never be `true`.

However, this logic will result in every customer receiving the 10 percent discount because every `salesCode` is either not *A* or not *B*. For example, if `salesCode` is *A*, then it is not *B*. The expression `salesCode != 'A' || salesCode != 'B'` is always `true`. The correct statement is either one of the following:

```
if(salesCode != 'A' && salesCode != 'B')
   discount = 0.10;
```

```
if(!(salesCode == 'A' || salesCode == 'B'))
   discount = 0.10;
```

In the first example, if `salesCode` is not `'A'` and it also is not `'B'`, then the discount is applied correctly. In the second example, if `salesCode` is `'A'` or `'B'`, the inner Boolean expression is `true`, and the NOT operator (`!`) changes the evaluation to `false`, not applying the discount for *A* or *B* sales. You also could avoid the confusing negative situation by asking questions in a positive way, as in the following:


```
if(salesCode == 'A' || salesCode == 'B')
   discount = 0;
else
   discount = 0.10;
```

Watch the video *Avoiding Common Decision Errors*.

TWO TRUTHS & A LIE

Avoiding Common Errors When Making Decisions

1. If you want to display *OK* when `userEntry` is 12 and when it is 13, then the following is a usable C# statement:

```
if(userEntry == 12 && userEntry == 13)
   WriteLine("OK");
```

2. If you want to display *OK* when `userEntry` is 20 or when `highestScore` is at least 70, then the following is a usable C# statement:

```
if(userEntry ==20 || highestScore >= 70)
  WriteLine("OK");
```

3. If you want to display *OK* when `userEntry` is anything other than 99 or 100, then the following is a usable C# statement:

```
if(userEntry != 99 && userEntry != 100)
   WriteLine("OK");
```

The false statement is #1. If you want to display *OK* when `userEntry` is 12 and when it is 13, then you want to display it when it is either 12 or 13 because it cannot be both simultaneously. The expression `userEntry == 12 && userEntry == 13` can never be true. The correct Boolean expression is `userEntry == 12 || userEntry == 13`.

Decision-Making Issues in GUI Programs

Making a decision within a method in a GUI application is no different from making one in a console application; you can use `if`, `if...else`, and `switch` statements in the same ways. For example, Figure 4-25 shows a GUI `Form` that determines a movie patron discount as described in a program earlier in this chapter. Patrons who are under 12 or over 65 and are seeing a G-rated movie receive a discount, and any other combination pays full price. Figure 4-26 contains the `Click()` method that makes the discount determination based on age and rating after a user clicks the *Discount?* button. The Boolean expression tested in the `if` statement in this method is identical to the one in the console version of the program in Figure 4-16.

Figure 4-25 The Movie Discount `Form`

```
private void discountButton_Click(object sender,
   EventArgs e)
{
   int age;
   char rating;
   const int CHILD_AGE = 12;
   const int SENIOR_AGE = 65;
   age = Convert.ToInt32(textBox1.Text);
   rating = Convert.ToChar(textBox2.Text);
   outputLabel.Text = String.Format
      ("When age is {0} and rating is {1}", age, rating);
   if ((age <= CHILD_AGE || age >= SENIOR_AGE) && rating == 'G')
      outputLabel.Text += "\nDiscount applies";
   else
      outputLabel.Text += "\nFull price";
}
```

Figure 4-26 The `discountButton_Click()` method for the `Form` in Figure 4-25

Event-driven programs often require fewer coded decisions than console applications. That's because in event-driven programs, some events are determined by the user's actions when the program is running (also called **at runtime**), rather than by the programmer's coding beforehand. You might say that in many situations, a console-based application must act, but an event-driven application has to *react*.

Suppose that you want to write a program in which the user must select whether to receive instructions in English or Spanish. In a console application, you would issue a prompt such as the following:

Which language do you prefer? Enter 1 for English or 2 for Spanish >>

The program would accept the user's entry, make a decision about it, and take appropriate action. However, in a GUI application, you are more likely to place controls on a `Form` to get a user's response. For example, you might use two `Button`s—one for English and one for Spanish. The user clicks a `Button`, and an appropriate method executes. No decision is written in the program because a different event is fired from each `Button`, causing execution of a different `Click()` method. The interactive environment decides which method is called, so the programmer does not have to code a decision. (Of course, you might alternately place a `TextBox` on a `Form` and ask a user to enter a 1 or a 2. In that case, the decision-making process would be identical to that in the console-based program.)

An additional benefit to having the user click a button to select an option is that the user cannot enter an invalid value. For example, if the user enters a letter in response to a prompt for an integer, the program will fail unless you write additional code to handle the mistake. However, if the user has a limited selection of buttons to click, no invalid entry can be made.

TWO TRUTHS & A LIE

Decision-Making Issues in GUI Programs

1. Event-driven programs can contain `if, if...else,` and `switch` statements.
2. Event-driven programs often require fewer coded decisions than console applications.
3. Event-driven programs usually contain more coded decisions than corresponding console-based applications.

The false statement is #3. Event-driven programs often require fewer coded decisions because user actions, such as clicking a button, are often used to trigger different methods.

You Do It

Creating a GUI Application That Uses an Enumeration and a `switch` Structure

In these steps, you create a GUI application for the Chatterbox Diner that allows a user to enter a day and see the special meal offered that day. Creating the program provides experience using an enumeration in a `switch` structure.

1. Open a new project in Visual Studio, and name it **DailySpecial**.
2. Design a `Form` like the one in Figure 4-27 that prompts the user for a day number and allows the user to enter it in a `TextBox`. Name the `TextBox` **`dayBox`** and the `Button` **`specialButton`**.

Figure 4-27 The Daily Special form

3. Below the `Button`, add a `Label` named **`outputLabel`** and delete its text.

(continues)

(continued)

4. Double-click **`specialButton`** to create a `specialButton_Click()` method shell. Above the method, add an enumeration for the days of the week as follows:

```
enum Day
{
   SUNDAY = 1, MONDAY, TUESDAY, WEDNESDAY,
      THURSDAY, FRIDAY, SATURDAY
}
```

5. Within the method, declare an integer and accept a value from the `TextBox`, and then declare a `string` to hold the daily special.

```
int day = Convert.ToInt32(dayBox.Text);
string special;
```

6. Add a `switch` structure that lists the daily specials as follows:

```
switch ((Day)day)
{
   case Day.SUNDAY:
      special = "fried chicken";
      break;
   case Day.MONDAY:
      special = "Sorry - closed";
      break;
   case Day.TUESDAY:
   case Day.WEDNESDAY:
   case Day.THURSDAY:
      special = "meat loaf";
      break;
   case Day.FRIDAY:
      special = "fish fry";
      break;
   case Day.SATURDAY:
      special = "liver and onions";
      break;
   default:
      special = "Invalid day";
      break;
}
```

7. Following the completed `case` structure, assign the result to the `Text` property of `outputLabel`:

```
outputLabel.Text = "Today's special is " + special;
```

8. Save, compile, and execute the application. The appropriate special is displayed for each day of the week.

Chapter Summary

- A flowchart is a pictorial tool that helps you understand a program's logic. A decision structure is one that involves choosing between alternative courses of action based on some value within a program.
- The `if` statement makes a single-alternative decision using the keyword `if`, followed by parentheses that contain a Boolean expression. When the expression is true, the statement body executes. The body can be a single statement or a block of statements.
- When you make a dual-alternative decision, you can use an `if-else` statement. You can block multiple statements after an `else` so they all execute when the evaluated expression is `false`.
- The conditional AND operator (&&) takes action when two operand Boolean expressions are both `true`. The conditional OR operator (||) takes action when at least one of two operand Boolean expressions is `true`. When && and || operators are combined within the same Boolean expression without parentheses, the && operators take precedence, meaning their Boolean values are evaluated first.
- The `switch` statement tests a single variable against a series of exact matches.
- The conditional operator is used as an abbreviated version of the `if-else` statement. It requires three expressions separated with a question mark and a colon.
- The NOT operator, which is written as an exclamation point (!), negates the result of any Boolean expression.
- Common errors when making decisions include using the assignment operator instead of the comparison operator, inserting a semicolon after the Boolean expression in an `if` statement, failing to block a set of statements when they should be blocked, and performing a range check incorrectly or inefficiently.
- Making a decision within a method in a GUI application is no different from making one in a console application; you can use `if`, `if...else`, and `switch` statements in the same ways. However, event-driven programs often require fewer coded decisions than console applications because some events are determined by the user's actions when the program is running, rather than by the programmer's coding beforehand.

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 program steps in diagram form, as a series of shapes connected by arrows.

A **sequence structure** is a unit of program logic in which one step follows another unconditionally.

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

An **if statement** is used to make a single-alternative decision.

A **block** is a collection of one or more statements contained within a pair of curly braces.

A **control statement** is the part of a structure that determines whether the subsequent block of statements executes.

A **nested** if statement is one in which one decision structure is contained within another.

Dual-alternative decisions have two possible outcomes.

An **if-else statement** performs a dual-alternative decision.

The **conditional AND operator** (or simply the **AND operator**) determines whether two expressions are both true; it is written using two ampersands (&&).

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

Short-circuit evaluation is the C# feature in which parts of an AND or OR expression are evaluated only as far as necessary to determine whether the entire expression is true or false.

The **conditional OR operator** (or simply the **OR operator**) determines whether at least one of two conditions is **true**; it is written using two pipes (||).

The **Boolean logical AND operator** determines whether two expressions are both true; it is written using a single ampersand (&). Unlike the conditional AND operator, it does not use short-circuit evaluation.

The **Boolean logical inclusive OR operator** determines whether at least one of two conditions is true; it is written using a single pipe (|). Unlike the conditional OR operator, it does not use short-circuit evaluation.

A **side effect** is an unintended consequence.

Bitwise operators are used to manipulate the individual bits of values.

The **switch structure** tests a single variable against a series of exact matches.

A **case label** identifies a course of action in a switch structure.

The **governing type** of a switch statement is established by the switch expression and can be sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or enum.

The **conditional operator** is used as an abbreviated version of the if-else statement; it requires three expressions separated by a question mark and a colon.

A **ternary** operator requires three arguments.

The **NOT operator** (!) negates the result of any Boolean expression.

A **range check** is a series of `if` statements that determine whether a value falls within a specified range.

At runtime is a phrase that means *during the time a program is running.*

Review Questions

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

```
int a = 3, b = 4;
if(a == b)
   Write("Black");
   WriteLine("White");
```

a. Black
b. White
c. BlackWhite
d. nothing

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

```
int a = 3, b = 4;
if(a < b)
{
   Write("Black");
   WriteLine("White");
}
```

a. Black
b. White
c. BlackWhite
d. nothing

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

```
int a = 3, b = 4;
if(a > b)
   Write("Black");
else
   WriteLine("White");
```

a. Black
b. White
c. BlackWhite
d. nothing

4. If the following code segment compiles correctly, what do you know about the variable `x`?

```
if(x) WriteLine("OK");
```

a. `x` is an integer variable.
b. `x` is a Boolean variable.
c. `x` is greater than 0.
d. none of these

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

```
int c = 6, d = 12;
if(c > d);
   Write("Green");
   WriteLine("Yellow");
```

a. Green
b. Yellow
c. GreenYellow
d. nothing

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

```
int c = 6, d = 12;
if(c < d)
   if(c > 8)
      Write("Green");
   else
      Write("Yellow");
else
   Write("Blue");
```

a. Green
b. Yellow
c. Blue
d. nothing

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

```
int e = 5, f = 10;
if(e < f && f < 0)
   Write("Red");
else
   Write("Orange");
```

a. Red
b. Orange
c. RedOrange
d. nothing

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

```
int e = 5, f = 10;
if(e < f || f < 0)
   Write("Red");
else
   Write("Orange");
```

a. Red
b. Orange
c. RedOrange
d. nothing

9. Which of the following expressions is equivalent to the following code segment?

```
if(g > h)
   if(g < k)
      Write("Brown");
```

a. `if(g > h && g < k) Write("Brown");`

b. `if(g > h && < k) Write("Brown");`

c. `if(g > h || g < k) Write("Brown");`

d. two of these

10. Which of the following expressions assigns `true` to a Boolean variable named `isIDValid` when `idNumber` is both greater than 1000 and less than or equal to 9999, or else is equal to 123456?

a.
```
isIDValid = (idNumber > 1000 && idNumber <= 9999 &&
   idNumber == 123456)
```

b.
```
isIDValid = (idNumber > 1000 && idNumber <= 9999 ||
   idNumber == 123456)
```

c.
```
isIDValid = ((idNumber > 1000 && idNumber <= 9999) ||
   idNumber == 123456)
```

d. two of these

11. Which of the following expressions is equivalent to `a || b && c || d`?

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

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

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

d. two of these

12. How many `case` labels would a `switch` statement require to be equivalent to the following `if` statement?

```
if(v == 1)
   WriteLine("one");
else
   WriteLine("two");
```

a. zero

b. one

c. two

d. impossible to tell

13. Falling through a `switch case` is most often prevented by using the ______________________ statement.

a. `break`

b. `default`

c. `case`

d. `end`

14. If the test expression in a `switch` does not match any of the `case` values, and there is no `default` value, then ___________________.

    a. a compiler error occurs

    b. a runtime error occurs

    c. the program continues with the next executable statement

    d. the expression is incremented and the `case` values are tested again

15. Which of the following is equivalent to the following statement:

```
if(m == 0)
   d = 0;
else
   d = 1;
```

    a. `d = (m == 0) : d = 0, d = 1;`

    b. `m ? (d = 0); (d = 1);`

    c. `m == 0; d = 0; d = 1?`

    d. `d = (m == 0) ? 0 : 1;`

16. Which of the following C# expressions is equivalent to `a < b && b < c`?

    a. `c > b > a`

    b. `a < b && c >= b`

    c. `!(b <= a) && b < c`

    d. two of these

17. Which of the following C# expressions means, "If `itemNumber` is not 8 or 9, add `TAX` to `price`"?

    a.
```
if(itemNumber != 8 || itemNumber != 9)
    price = price + TAX;
```

    b.
```
if(itemNumber != 8 && itemNumber != 9)
    price = price + TAX;
```

    c.
```
if(itemNumber != 8 && !=          9)
    price = price + TAX;
```

    d. two of these

18. Which of the following C# expressions means, "If `itemNumber` is 1 or 2 and `quantity` is 12 or more, add `TAX` to `price`"?

    a.
```
if(itemNumber = 1 || itemNumber = 2 && quantity >=12)
    price = price + TAX;
```

    b.
```
if(itemNumber == 1 || itemNumber == 2 || quantity >=12)
    price = price + TAX;
```

    c.
```
if(itemNumber == 1 && itemNumber == 2 && quantity >=12)
    price = price + TAX;
```

    d. none of these

19. Which of the following C# expressions means, "If `itemNumber` is 5 and `zone` is 1 or 3, add `TAX` to `price`"?

    a.
    ```
    if(itemNumber == 5 && zone == 1 || zone == 3)
        price = price + TAX;
    ```
    b.
    ```
    if(itemNumber == 5 && (zone == 1 || zone == 3))
        price = price + TAX;
    ```
    c.
    ```
    if(itemNumber == 5 && (zone ==1 || 3))
        price = price + TAX;
    ```
    d. two of these

20. Which of the following C# expressions results in `TAX` being added to `price` if the integer `itemNumber` is not 100?

    a.
    ```
    if(itemNumber != 100)
        price = price + TAX;
    ```
    b.
    ```
    if(!(itemNumber == 100))
        price = price + TAX;
    ```
    c.
    ```
    if(itemNumber <100 || itemNumber > 100)
        price = price + TAX;
    ```
    d. all of these

Exercises

Programming Exercises

1. Write a program named **CheckCredit** that prompts users to enter a purchase price for an item. If the value entered is greater than a credit limit of $5,000, display an error message; otherwise, display *Approved*.

2. Write a program named **Twitter** that accepts a user's message and determines whether it is short enough for a social networking service that does not accept messages of more than 140 characters.

3. Write a program named **Admission** for a college's admissions office. The user enters a numeric high school grade point average (for example, 3.2) and an admission test score. Display the message *Accept* if the student meets either of the following requirements:

    - A grade point average of 3.0 or higher, and an admission test score of at least 60
    - A grade point average of less than 3.0, and an admission test score of at least 80
    - If the student does not meet either of the qualification criteria, display *Reject*.

4. The Saffir-Simpson Hurricane Scale classifies hurricanes into five categories numbered 1 through 5. Write an application named **Hurricane** that outputs a hurricane's category based on the user's input of the wind speed. Category 5 hurricanes have sustained winds of at least 157 miles per hour. The minimum sustained wind speeds for categories 4 through 1 are 130, 111, 96, and 74 miles per hour, respectively. Any storm with winds of less than 74 miles per hour is not a hurricane.

5. Write a program named **CheckMonth** that prompts a user to enter a birth month. If the value entered is greater than 12 or less than 1, display an error message; otherwise, display the valid month with a message such as *3 is a valid month.*

6. Write a program named **CheckMonth2** that prompts a user to enter a birth month and day. Display an error message if the month is invalid (not 1 through 12) or the day is invalid for the month (for example, not between 1 and 31 for January or between 1 and 29 for February). If the month and day are valid, display them with a message.

7. You can create a random number that is at least `min` but less than `max` using the following statements:

```
Random ranNumberGenerator = new Random();
int randomNumber;
randomNumber = ranNumberGenerator.Next(min, max);
```

   Write a program named **GuessingGame** that generates a random number between 1 and 10. (In other words, `min` is 1 and `max` is 11.) Ask a user to guess the random number, then display the random number and a message indicating whether the user's guess was too high, too low, or correct.

8. 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 character, *r*, *p*, or *s,* each representing one of the three choices. Then, determine the winner. Save the application as **RockPaperScissors.cs**.

9. Create a lottery game application named **Lottery**. Generate three random numbers, each between 1 and 4. 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	1000
Three matching in exact order	10,000
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.

Debugging Exercises

1. Each of the following files in the Chapter.04 folder of your downloadable student files has syntax and/or logical 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 *Fixed*. For example, save DebugFour1.cs as **FixedDebugFour1.cs**.
   a. DebugFour1.cs
   b. DebugFour2.cs
   c. DebugFour3.cs
   d. DebugFour4.cs

Case Problems

1. In Chapter 2, you created an interactive application named **GreenvilleRevenue**, and in Chapter 3 you created a GUI version of the application named **GreenvilleRevenueGUI**. The programs prompt a user for the number of contestants entered in this year's and last year's Greenville Idol competition, and then they display the revenue expected for this year's competition if each

contestant pays a $25 entrance fee. The programs also display a statement that compares the number of contestants each year. Now, replace that statement with one of the following messages:

- If the competition has more than twice as many contestants as last year, display *The competition is more than twice as big this year!*
- If the competition is bigger than last year's but not more than twice as big, display *The competition is bigger than ever!*
- If the competition is smaller than last year's, display, *A tighter race this year! Come out and cast your vote!*

2. In Chapter 2, you created an interactive application named **MarshallsRevenue**, and in Chapter 3 you created a GUI version of the application named **MarshallsRevenueGUI**. The programs prompt a user for the number of interior and exterior murals scheduled to be painted during the next month by Marshall's Murals. Next, the programs compute the expected revenue for each type of mural when interior murals cost $500 each and exterior murals cost $750 each. The applications also display the total expected revenue and a statement that indicates whether more interior murals are scheduled than exterior ones. Now, modify one or both of the applications to accept a numeric value for the month being scheduled and to modify the pricing as follows:

   - Because of uncertain weather conditions, exterior murals cannot be painted in December through February, so change the number of exterior murals to 0 for those months.
   - Marshall prefers to paint exterior murals in April, May, September, and October. To encourage business, he charges only $699 for an exterior mural during those months. Murals in other months continue to cost $750.
   - Marshall prefers to paint interior murals in July and August, so he charges only $450 for an interior mural during those months. Murals in other months continue to cost $500.