CHAPTER 5
Branching

All the statements in your program execute in order. Unfortunately, that's not very useful, unless you want your program to do exactly the same thing every time you run it. In fact, often you won't want to execute all the code, but rather you'll want the program to do one thing if a variable has a certain value, and something different if the variable has another value. That means you need to be able to cause your program to pick and choose which statements to execute based on conditions that change as the program runs. This process is called *branching*, and there are two ways to accomplish it: unconditionally or conditionally.

As the name implies, unconditional branching happens every time the branch point is reached. An unconditional branch happens, for example, whenever the compiler encounters a new method call. The compiler stops execution in the current method and branches to the newly called method. When the newly called method *returns* (completes its execution), execution picks up in the original method on the line just below the branch point (the line where the new method was called).

Conditional branching is more complicated. Methods can branch based on the evaluation of certain conditions that occur at runtime. For instance, you might create a branch that will calculate an employee's federal withholding tax only when their earnings are greater than the minimum taxable by law. C# provides a number of statements that support conditional branching, such as `if`, `else`, and `switch`. The use of these statements is discussed later in this chapter.

A second way that methods break out of their mindless step-by-step processing of instructions is by looping. A loop causes the method to repeat a set of steps until some condition is met ("Keep asking for input until the user tells you to stop or until you receive ten values"). C# provides many statements for looping, including `for`, `while`, and `do...while`, which are also discussed in this chapter.

Unconditional Branching Statements

The most simple example of an unconditional branch is a method call. When a method call is reached, there is no test made to evaluate the state of the object; the program execution branches immediately (and unconditionally) to the start of the new method.

You call a method by writing its name; for example:

```
UpdateSalary(); // invokes the method UpdateSalary
```

As I explained earlier in the chapter, when the compiler encounters a method call, it stops execution of the current method and branches to the new method. When that new method completes its execution, the compiler picks up where it left off in the original method. This process is illustrated schematically in Figure 5-1.

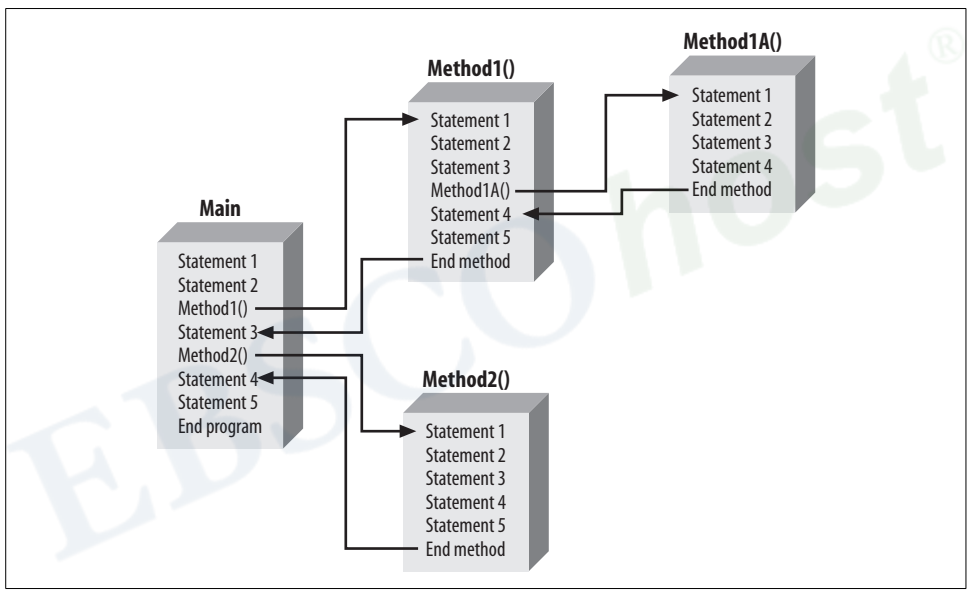

Figure 5-1. How branching works

As Figure 5-1 suggests, it is actually quite common for there to be unconditional branching several methods deep. In Figure 5-1, execution begins in a method called `Main()`. Statement1 and Statement2 execute; then the compiler sees a call to `Method1()`. Program execution branches unconditionally to the first line of `Method1()`, where its first three statements are executed. At the call to `Method1A()`, execution again branches, this time to the start of `Method1A()`.

The four statements in `Method1A()` are executed, and `Method1A()` returns. Execution resumes on the first statement after the method call in `Method1()` (Statement4). Execution continues until `Method1()` ends, at which time execution resumes back in

Main( ) at Statement3. At the call to Method2( ), execution again branches; all the statements in Method2( ) execute, and then Main( ) resumes at Statement4. When Main( ) ends, the program itself ends.

You can see the effect of method calls in Example 5-1. Execution begins in Main( ), but branches to a method named SomeMethod( ). The WriteLine( ) statements in each method assist you in seeing where you are in the code as the program executes.

Example 5-1. Branching to a method

```
using System;
class Functions
{
   static void Main( )
   {
      Console.WriteLine( "In Main! Calling SomeMethod( )..." );
      SomeMethod( );
      Console.WriteLine( "Back in Main( )." );

   }
   static void SomeMethod( )
   {
      Console.WriteLine( "Greetings from SomeMethod!" );
   }
}
```

The output looks like this:

```
In Main! Calling SomeMethod( )...
Greetings from SomeMethod!
Back in Main( ).
```

Program flow begins in Main( ) and proceeds until SomeMethod( ) is invoked. (Invoking a method is sometimes referred to as *calling* the method.) At that point, program flow branches to the method. When the method completes, program flow resumes at the next line after the call to that method.

You can instead create an unconditional branch by using one of the unconditional branch keywords: goto, break, continue, return, or throw. The first four of these are discussed later in this chapter, while the final statement, throw, is discussed in Chapter 16.

Methods and their parameters and return values are discussed in detail in Chapter 8.

Conditional Branching Statements

Although methods branch unconditionally, often you will want to branch within a method depending on a condition that you evaluate while the program is running.

This is known as *conditional branching*. Conditional branching statements allow you to write logic such as, "If you are over 25 years old, then you may rent a car."

C# provides a number of constructs that allow you to write conditional branches into your programs; these constructs are described in the following sections.

if Statements

The simplest branching statement is `if`. An `if` statement says, "if a particular condition is true, then execute the statement; otherwise, skip it." The condition is a Boolean expression. An expression is a statement that evaluates to a value, and a Boolean expression evaluates to either true or false.

The formal description of an `if` statement is:

```
if (expression)
 Statement1
```

This is the kind of description of the `if` statement you are likely to find in your compiler documentation. It shows you that the `if` statement takes an expression (a statement that returns a value) in parentheses, and executes Statement1 if the expression evaluates true. Note that Statement1 can actually be a block of statements within braces, as illustrated in Example 5-2.

Example 5-2. The if statement

```
using System;
namespace Branching
{
   class Test
   {
      static void Main( )
      {
         int valueOne = 10;
         int valueTwo = 20;
         int valueThree = 30;

         Console.WriteLine( "Testing valueOne against valueTwo..." );
         if ( valueOne > valueTwo )
         {
            Console.WriteLine(
            "ValueOne: {0} larger than ValueTwo: {1}",
            valueOne, valueTwo );
         }

         Console.WriteLine( "Testing valueThree against valueTwo..." );
         if ( valueThree > valueTwo )
         {
            Console.WriteLine(
            "ValueThree: {0} larger than ValueTwo: {1}",
            valueThree, valueTwo );
         }    // end if
```

Example 5-2. The if statement (continued)

```
      }       // end Main
   }          // end class
}             // end namespace
```

Just about anywhere in C# that you are expected to provide a statement, you can instead provide a block of statements within braces. (See the sidebar, "Brace Styles," later in this chapter.)

In this simple program, you declare three variables, `valueOne`, `valueTwo`, and `valueThree`, with the values 10, 20, and 30, respectively. In the first `if` statement, you test whether `valueOne` is greater than `valueTwo`:

```
if ( valueOne > valueTwo )
{
   Console.WriteLine(
   "ValueOne: {0} larger than ValueTwo: {1}",
   valueOne, valueTwo );
}
```

Because `valueOne` (10) is less than `valueTwo` (20), this `if` statement fails (the condition returns false), and thus the body of the `if` statement (the statements within the braces) doesn't execute.

You then test whether `valueThree` is greater than `valueTwo`:

```
if ( valueThree > valueTwo )
{
   Console.WriteLine(
   "ValueThree: {0} larger than ValueTwo: {1}",
   valueThree, valueTwo );
} // end if
```

Because `valueThree` (30) *is* greater than `valueTwo` (20), the test returns true, and thus the statement executes. The statement in this case is the block in which you call the `WriteLine( )` method, shown in bold. The output reflects that the first `if` fails but the second succeeds:

```
Testing valueOne against valueTwo...
Testing valueThree against valueTwo...
ValueThree: 30 larger than ValueTwo: 20
```

Single-Statement if Blocks

Notice that the `if` statement blocks shown in Example 5-2 each contain only a single statement, one call to `WriteLine( )`. In such cases, you can leave out the braces enclosing the `if` block. Thus, you might rewrite Example 5-2, as shown in Example 5-3.

Example 5-3. Single statements with if

```
using System;

namespace Branching
{
   class Test
   {
      static void Main( )
      {
         int valueOne = 10;
         int valueTwo = 20;
         int valueThree = 30;

         Console.WriteLine( "Testing valueOne against valueTwo..." );
         if ( valueOne > valueTwo )
            Console.WriteLine(
            "ValueOne: {0} larger than ValueTwo: {1}",
            valueOne, valueTwo );

         Console.WriteLine( "Testing valueThree against valueTwo..." );
         if ( valueThree > valueTwo )
            Console.WriteLine(
            "ValueThree: {0} larger than ValueTwo: {1}",
            valueThree, valueTwo );

      }  // end Main
   }     // end class
}        // end namespace
```

It is generally a good idea, however, to use the braces even when your `if` block has only a single statement. There are two reasons for this advice. First, the code is somewhat easier to read and understand with the braces. Code that is easier to read is easier to maintain.

When programmers talk about *maintaining* code, they mean either adding to the code as requirements change or fixing the code as bugs arise.

The second reason for using braces is to avoid a common error: adding a second statement to the `if` and forgetting to add the braces. Consider the code shown in Example 5-4. The programmer has changed the value of `valueThree` to 10 and added a second statement to the second `if` block, as shown in bold.

Example 5-4. Adding a second statement to if

```
using System;

namespace Branching
{
   class Test
```

Example 5-4. Adding a second statement to if (continued)

```
   {
      static void Main( )
      {
         int valueOne = 10;
         int valueTwo = 20;
         int valueThree = 10;

         Console.WriteLine( "Testing valueOne against valueTwo..." );
         if ( valueOne > valueTwo )
            Console.WriteLine(
            "ValueOne: {0} larger than ValueTwo: {1}",
            valueOne, valueTwo );

         Console.WriteLine( "Testing valueThree against valueTwo..." );
         if ( valueThree > valueTwo )
            Console.WriteLine(
            "ValueThree: {0} larger than ValueTwo: {1}",
            valueThree, valueTwo );
         Console.WriteLine( "Good thing you tested again!" );

      } // end Main
   }    // end class
}       // end namespace
```

Now, before reading any further, review the code and decide for yourself what the output should be. Don't cheat by looking past this paragraph. Then, when you think you know what the output will be, take a look at this:

```
Testing valueOne against valueTwo...
Testing valueThree against valueTwo...
Good thing you tested again!
```

Were you surprised?

The programmer was fooled by the lack of braces and the indentation. Remember that indentation is whitespace and is ignored by the compiler. From the perspective of the programmer, the second statement ("Good thing…") is part of the `if` block:

```
if ( valueThree > valueTwo )
     Console.WriteLine(
     "ValueThree: {0} larger than ValueTwo: {1}",
     valueThree, valueTwo);
     Console.WriteLine("Good thing you tested again!");
```

The compiler, however, considers only the first statement after the `if` test to be part of the `if` statement. The second statement is not part of the `if` statement. To the compiler, the `if` statement looks like this:

```
if ( valueThree > valueTwo )
     Console.WriteLine(
     "ValueThree: {0} larger than ValueTwo: {1}",
     valueThree, valueTwo);

Console.WriteLine("Good thing you tested again!");
```

If you want the second statement to be part of the `if` statement, you must use braces, as in the following:

```
if ( valueThree > valueTwo )
{
     Console.WriteLine(
     "ValueThree: {0} larger than ValueTwo: {1}",
     valueThree, valueTwo);
     Console.WriteLine("Good thing you tested again!");
}
```

Because of this potential for confusion, many C# programmers use braces with every `if` statement, even if the statement is only one line.

Brace Styles

There are many ways you can form braces around an `if` statement (and around other blocks of code), but most C# programmers will use one of three styles:

```
if (condition)
{
 // statement
}

if (condition)
 {
 // statement
 }

if (condition){
 // statement
}
```

The first style, used throughout this book, is to put the braces under the keyword `if` and to indent the contents of the `if` block. The second style, which is not very popular anymore, is to indent the braces with the contents of the `if` block. The third style is to put the opening brace on the same line as the `if` statement and the closing brace under the `if` statement.

The third style is called K&R style, after Kernighan and Ritchie, the authors of the seminal book *The C Programming Language* (Prentice Hall, 1988). Their book was so influential that many programmers feel a strong commitment to this style of braces. Although it does save room in a book, the K&R style is a bit less clear, and so this book will use the first style.

Short-Circuit Evaluation

Consider the following code snippet:

```
int x = 8;
int y = 15;
if ((x == 8) || (y == 12))
```

The `if` statement here is a bit complicated. The entire `if` statement is in parentheses, as are all `if` statements in C#. Thus, everything within the outer set of parentheses must evaluate true for the `if` statement to be true.

Within the outer parentheses are two expressions, `(x == 8)` and `(y == 12)`, which are separated by an `or` operator (`||`). Because `x` is 8, the first term `(x == 8)` evaluates true. There is no need to evaluate the second term `(y == 12)`. It doesn't matter whether `y` is 12; the entire expression will be true. Similarly, consider this snippet:

```
int x = 8;
int y = 12;
if ((x == 5) && (y == 12))
```

Again, there is no need to evaluate the second term. Because the first term is false, the `and` must fail. (Remember, for an `and` statement to evaluate true, both tested expressions must evaluate true.)

In cases such as these, the C# compiler will short-circuit the evaluation; the second test will never be performed. This allows you to create if statements in which you first check a value before you take action on it, avoiding the possibility of an exception. Here's a short example:

```
public bool QuotientOverTwenty(float dividend, float divisor)
{
   if ( divisor != 0 && dividend / divisor > 20 )
   {
      return true;
   }
   return false;
}
```

In this code, we only want to decide if the quotient is greater than 20, but we must first make sure we are not dividing by zero (division by zero causes the system to throw an exception). With short circuiting, the second part of the `if` statement (the division) will never occur if the first part is false (that is, if the divisor is zero), and this code is terser and perhaps easier to understand than writing.

```
public bool QuotientOverTwenty(float dividend, float divisor)
{
   bool retVal = false;
   if ( divisor != 0 )
   {
      if ( dividend / divisor > 20 )
```

```
            retVal = true;
        }
        return retVal;
    }
```

if . . . else Statements

Often, you will find that you want to take one set of actions when the condition tests true and a different set of actions when the condition tests false. This allows you to write logic such as, "If you are over 25 years old, then you may rent a car; *otherwise*, you must take the train."

The *otherwise* portion of the logic is executed in the `else` statement. For example, you can modify Example 5-2 to print an appropriate message whether or not `valueOne` is greater than `valueTwo`, as shown in Example 5-5.

Example 5-5. The else statement

```
using System;

namespace Branching
{
   class Test
   {
      static void Main( )
      {
         int valueOne = 10;
         int valueTwo = 20;

         Console.WriteLine( "Testing valueOne against valueTwo..." );
         if ( valueOne > valueTwo )
         {
            Console.WriteLine(
            "ValueOne: {0} larger than ValueTwo: {1}",
            valueOne, valueTwo );
         } // end if
         else
         {
            Console.WriteLine(
            "Nope, ValueOne: {0} is NOT larger than ValueTwo: {1}",
            valueOne, valueTwo );
         } // end else

      } // end Main
   }    // end class
}       // end namespace
```

The output looks like this:

```
Testing valueOne against valueTwo...
Nope, ValueOne: 10 is NOT larger than ValueTwo: 20
```

Because the test in the `if` statement fails (`valueOne` is *not* larger than `valueTwo`), the body of the `if` statement is skipped and the body of the `else` statement is executed. Had the test succeeded, the `if` statement body would execute and the `else` statement would be skipped.

Nested if Statements

It is possible, and not uncommon, to nest `if` statements to handle complex conditions. For example, suppose you need to write a program to evaluate the temperature and specifically to return the following types of information:

- If the temperature is 32 degrees or lower, the program should warn you about ice on the road.
- If the temperature is exactly 32 degrees, the program should tell you that there may be ice patches.
- If the temperature is higher than 32 degrees, the program should assure you that there is no ice.

There are many good ways to write this program. Example 5-6 illustrates one approach using nested `if` statements.

Example 5-6. Nested if statements

```
using System;
class Values
{
   static void Main( )
   {
      int temp = 32;

      if ( temp <= 32 )
      {
         Console.WriteLine( "Warning! Ice on road!" );
         if ( temp == 32 )
         {
            Console.WriteLine("Temp exactly freezing, beware of water." );
         }
         else
         {
            Console.WriteLine( "Watch for black ice! Temp: {0}", temp );
         }
      }

   }
}
```

The logic of Example 5-6 is that it tests whether the temperature is less than or equal to 32. If so, it prints a warning:

```
if (temp <= 32)
{
    Console.WriteLine("Warning! Ice on road!");
```

The program then checks whether the temp is equal to 32 degrees. If so, it prints one message; if not, the temp must be less than 32 and the program prints the next message. Notice that this second `if` statement is nested within the first `if`, so the logic of the `else` statement is: "because it has been established that the temp is less than or equal to 32, and it isn't equal to 32, it must be less than 32."

Another way of chaining together more than one possibility with `if` statements is the `else if` idiom that some C# programmers use. The program tests the condition in the first `if` statement. If that first statement is false, control passes to the `else` statement, which is immediately followed by another `if` that tests a different condition. For example, you could rewrite Example 5-6 to test if the temperature is greater than, less than, or exactly equal to freezing with three tests, as shown in Example 5-7.

Example 5-7. Using else if

```
using System;
class Values
{
   static void Main( )
   {
      int temp = 32;

      if ( temp < 32 )
      {
         Console.WriteLine( "Warning! Ice on road!" );
      }
      else if ( temp == 32 )
      {
         Console.WriteLine("Temp exactly freezing, beware of water." );
      }
      else
      {
         Console.WriteLine( "Watch for black ice! Temp: {0}", temp );
      }
   }
}
```

In this case, the condition in the first `if` statement tests whether `temp` is less than 32, not less than or equal. Because `temp` is hard-wired to exactly 32, the first expression is false, and control passes to the `else if` statement. The second statement is true, so the third case, the `else` statement, never executes. Please note, however, that this code is *identical* (as far as the compiler is concerned) to the following:

```
using System;
class Values
```

```
{
   static void Main( )
   {
      int temp = 32;

      if ( temp < 32 )
      {
         Console.WriteLine( "Warning! Ice on road!" );
      }
      else
      {
          if ( temp == 32 )
          {
            Console.WriteLine("Temp exactly freezing, beware of water." );
          }
          else
         {
            Console.WriteLine( "Watch for black ice! Temp: {0}", temp );
         }
      }
   }
}
```

In any case, if you do use the `else if` idiom, be sure to use an `else`, (not an `else if`), as your final test, making it the default case that will execute even if nothing else does.

switch Statements

Nested `if` statements are hard to read, hard to get right, and hard to debug. When you have a complex set of choices to make, the `switch` statement is a more powerful alternative. The logic of a `switch` statement is this: "Pick a matching value and act accordingly."

```
switch (expression)
{
 case constant-expression:
 statement
 jump-statement
 [default: statement]
}
```

The expression you are "switching on" is put in parentheses in the head of the `switch` statement. Each `case` statement compares a constant value with the expression. The constant expression can be a literal, symbolic, or enumerated constant.

The compiler starts with the first `case` statement and works its way down the list, looking for a value that matches the expression. If a case is matched, the statement (or block of statements) associated with that case is executed.

The `case` block must end with a jump statement. Typically, the jump statement is `break`, which abruptly ends the entire `switch` statement. When you execute a `break` in

a switch statement, execution continues after the closing brace of the switch statement. (We'll consider the use of the optional default keyword later in this section.)

In the next, somewhat whimsical listing (Example 5-8), the user is asked to choose his political affiliation among Democrat, Republican, or Progressive. To keep the code simple, I'll hardwire the choice to be Democrat.

Example 5-8. Using a switch statement

```
using System;


   class Values
   {
      enum Party
      {
         Democrat,
         Republican,
         Progressive
      }
      static void Main()
      {

         // hard wire to Democratic
         Party myChoice = Party.Democrat;

         // switch on the value of myChoice
         switch ( myChoice )
         {
            case Party.Democrat:
               Console.WriteLine( "You voted Democratic." );
               break;
            case Party.Republican:
               Console.WriteLine( "You voted Republican." );
               break;
            case Party.Progressive:
               Console.WriteLine( "You voted Progressive." );
               break;

         }
         Console.WriteLine( "Thank you for voting." );
      }
   }
```

The output looks like this:

```
You voted Democratic.
Thank you for voting.
```

Rather than using a complicated if statement, Example 5-8 uses a switch statement. The user's choice is evaluated in the head of the switch statement, and the block of statements that gets executed depends on whatever case matches (in this instance, Democrat).

The statements between the case statement and the break are executed in series. You can have more than one statement here without braces; in effect, the case statement and the closing break statement act as the braces.

It is possible that the user will not make a choice among Democrat, Republican, and Progressive. You may want to provide a default case that will be executed whenever no valid choice has been made. You can do that with the default keyword, as shown in Example 5-9.

Example 5-9. A default statement

```
using System;

   class Values
   {
      enum Party
      {
         Democrat,
         Republican,
         Progressive
      }
      static void Main()
      {

         // hard wire to Democratic
         Party myChoice = Party.Democrat;

         // switch on the value of myChoice
         switch ( myChoice )
         {
            case Party.Democrat:
               Console.WriteLine( "You voted Democratic." );
               break;
            case Party.Republican:
               Console.WriteLine( "You voted Republican." );
               break;
            case Party.Progressive:
               Console.WriteLine( "You voted Progressive." );
               break;
            default:
               Console.WriteLine( "You did not make a valid choice." );
               break;

         }
         Console.WriteLine( "Thank you for voting." );
      }
   }
```

The output looks like this:

```
You did not make a valid choice.
Thank you for voting.
```

If the user does not choose one of the values that correspond to a case statement, the default statements will execute. In this case, a message is simply printed telling the user he did not make a valid choice; in production code, you would put all this in a while loop, re-prompting the user until a valid choice is made (or the user elects to quit).

Falling-Through and Jumping-to Cases

If two cases will execute the same code, you can create what's known as a "fall through" case, grouping the case statements together with the same code, as shown here:

```
case CompassionateRepublican:
case Republican:
     Console.WriteLine("You voted Republican.\n");
     Console.WriteLine("Don't you feel compassionate?");
     break;
```

In this example, if the user chooses either CompassionateRepublican or Republican, the same set of statements will be executed.

Note that you can only fall through if the first case executes no code. In this example, the first case, CompassionateRepublican, meets that criteria. Thus, you can fall through to the second case.

If, however, you want to execute a statement with one case and then fall through to the next, you must use the goto keyword to jump to the next case you want to execute.

The goto keyword is an unconditional branch. When the compiler sees this word, it immediately transfers the flow (jumps) to wherever the goto points to. Thus, even within this conditional branching statement, you've inserted an unconditional branch.

For example, if you create a NewLeft party, you might want the NewLeft voting choice to print a message and then fall through to Democrat (that is, continue on with the statements in the Democrat case). You might (incorrectly) try writing the following:

```
case NewLeft:
     Console.WriteLine("The NewLeft members are voting Democratic.");
case Democrat:
     Console.WriteLine("You voted Democratic.\n");
     break;
```

This code will not compile; it will fail with the error:

```
Control cannot fall through from one case label (case '4:') to another
```

This is a potentially misleading error message. Control *can* fall through from one case label to another, but only if there is no code in the first case label.

Notice that the error displays the name of the case with its numeric value (4) rather than its symbolic value (`NewLeft`). Remember that `NewLeft` is just the name of the constant:

```
const int Democrat = 0;
const int CompassionateRepublican = 1;
const int Republican = 2;
const int Progressive = 3;
const int NewLeft = 4;
```

Because the `NewLeft` case has a statement, the `WriteLine( )` method, you must use a `goto` statement to fall through:

```
case NewLeft:
     Console.WriteLine("The NewLeft members are voting Democratic.");
     goto case Democrat;
case Democrat:
     Console.WriteLine("You voted Democratic.\n");
     break;
```

This code will compile and execute as you expect.

The `goto` can jump over labels; you do not need to put `NewLeft` just above `Democrat`. In fact, you can put `NewLeft` last in the list (just before `default`), and it will continue to work properly.

Switch on string Statements

In the previous example, the `switch` value was an integral constant. C# also offers the ability to switch on a `string`. Thus, you can rewrite Example 5-9 to switch on the string "NewLeft," as in Example 5-10.

Example 5-10. Switching on a string

```
using System;

class Values
{
   static void Main( )
   {
      String myChoice = "NewLeft";

      // switch on the string value of myChoice
      switch ( myChoice )
      {
         case "NewLeft":
            Console.WriteLine(
            "The NewLeft members are voting Democratic." );
            goto case "Democrat";
```

Example 5-10. Switching on a string (continued)

```
            case "Democrat":
               Console.WriteLine( "You voted Democratic.\n" );
               break;
            case "CompassionateRepublican": // fall through
            case "Republican":
               Console.WriteLine( "You voted Republican.\n" );
               Console.WriteLine( "Don't you feel compassionate?" );
               break;
            case "Progressive":
               Console.WriteLine( "You voted Progressive.\n" );
               break;
            default:
               Console.WriteLine( "You did not make a valid choice." );
               break;
         }
         Console.WriteLine( "Thank you for voting." );
      }
}
```

Iteration (Looping) Statements

There are many situations in which you will want to do the same thing again and again, perhaps slightly changing a value each time you repeat the action. This is called *iteration*, or looping. Typically, you'll iterate (or loop) over a set of items, taking the same action on each item in the collection. This is the programming equivalent of an assembly line. On an assembly line, you might take a hundred car bodies and put a windshield on each one as it comes by. In an iterative program, you might work your way through a collection of text boxes on a form, retrieving the value from each in turn and using those values to update a database.

C# provides an extensive suite of iteration statements, including `for` and `while`, and also `do...while` and `foreach` loops. You can also create a loop by using the `goto` statement. The remainder of this chapter considers the use of `goto`, `for`, `while`, and `do...while`. However, we'll postpone coverage of `foreach` until Chapter 10.

Creating Loops with goto

The `goto` statement was used earlier in this chapter as an unconditional branch in a `switch` statement. The more common use of `goto`, however, is to create a loop. In fact, the `goto` statement is the seed from which all other looping statements have been germinated. Unfortunately, it is a semolina seed, producer of "spaghetti code" (see the following sidebar) and endless confusion.

Spaghetti Code

Goto can cause your method to loop back and forth in ways that are difficult to follow. If you were to try to draw the flow of control in a program that makes extensive use of goto statements, the resulting morass of intersecting and overlapping lines might look like a plate of spaghetti—hence the term "spaghetti code."

Spaghetti code is a contemptuous epithet; no one *wants* to write spaghetti code, and so most experienced programmers avoid using goto to create loops.

Because of the problems created by the goto statement, it is rarely used in C# outside of switch statements, but in the interest of completeness, here's how you create goto loops:

1. Create a label.
2. goto that label.

The *label* is an identifier followed by a colon. You place the label in your code, and then you use the goto keyword to jump to that label. The goto command is typically tied to an if statement, as illustrated in Example 5-11.

Example 5-11. Using goto

```
using System;
public class Tester
{

   public static void Main( )
   {
      int counterVariable = 0;

   repeat: // the label

      Console.WriteLine(
      "counterVariable: {0}", counterVariable );

      // increment the counter
      counterVariable++;

      if ( counterVariable < 10 )
         goto repeat; // the dastardly deed
   }
}
```

The output looks like this:

```
    counterVariable: 0
    counterVariable: 1
    counterVariable: 2
```

```
counterVariable: 3
counterVariable: 4
counterVariable: 5
counterVariable: 6
counterVariable: 7
counterVariable: 8
counterVariable: 9
```

This code is not terribly complex; you've used only a single `goto` statement. However, with multiple such statements and labels scattered through your code, tracing the flow of execution becomes very difficult.

It was the phenomenon of spaghetti code that led to the creation of alternatives, such as the `while` loop.

The while Loop

The semantics of the `while` loop are "while this condition is true, do this work." The syntax is:

```
while (Boolean expression) statement
```

As usual, a Boolean expression is any statement that evaluates to true or false. The statement executed within a `while` statement can of course be a block of statements within braces. Example 5-12 illustrates the use of the `while` loop.

Example 5-12. The while loop

```
using System;
public class Tester
{
   public static void Main( )
   {
      int counterVariable = 0;

      // while the counter variable is less than 10
      // print out its value
      while ( counterVariable < 10 )
      {
         Console.WriteLine( "counterVariable: {0}", counterVariable );
         counterVariable++;
      }
   }
}
```

The output looks like this:

```
counterVariable: 0
counterVariable: 1
counterVariable: 2
counterVariable: 3
counterVariable: 4
counterVariable: 5
```

```
counterVariable: 6
counterVariable: 7
counterVariable: 8
counterVariable: 9
```

The code in Example 5-12 produces results identical to the code in Example 5-11, but the logic is a bit more clear. The `while` statement is nicely self-contained, and it reads like an English sentence: "while `counterVariable` is less than 10, print this message and increment `counterVariable`."

Notice that the `while` loop tests the value of `counterVariable` before entering the loop. This ensures that the loop will not run if the condition tested is false. Thus, if `counterVariable` is initialized to 11, the loop will never run.

The do . . . while Loop

There are times when a `while` loop might not serve your purpose. In certain situations, you might want to reverse the semantics from "run while this is true" to the subtly different "do this, while this condition remains true." In other words, take the action, and then, after the action is completed, check the condition. Such a loop will *always* run at least once.

To ensure that the action is taken before the condition is tested, use a `do...while` loop:

```
do statement while (boolean-expression);
```

The syntax is to write the keyword `do`, followed by your statement (or block), the `while` keyword, and the condition to test in parentheses. End the statement with a semicolon.

Example 5-13 rewrites Example 5-12 to use a `do...while` loop.

Example 5-13. The do...while loop

```
using System;
public class Tester
{
   public static void Main( )
   {
      int counterVariable = 11;

      // display the message and then test that the value is
      // less than 10
      do
      {
         Console.WriteLine( "counterVariable: {0}", counterVariable );
         counterVariable++;
      } while ( counterVariable < 10 );
   }
}
```

The output looks like this:

```
counterVariable: 11
```

In Example 5-13, `counterVariable` is initialized to 11 and the `while` test fails, but only after the body of the loop has run once.

The for Loop

A careful examination of the `while` loop in Example 5-12 reveals a pattern often seen in iterative statements: initialize a variable (`counterVariable=0`), test the variable (`counterVariable<10`), execute a series of statements, and increment the variable (`counterVariable++`). The `for` loop allows you to combine all these steps in a single statement. You write a `for` loop with the keyword `for`, followed by the `for` header, using the syntax:

```
for ([initializers]; [expression]; [iterators]) statement
```

The first part of the header is the *initializer*, in which you initialize a variable. The second part is the Boolean expression to test. The third part is the *iterator*, in which you update the value of the counter variable. All of this is enclosed in parentheses.

A simple `for` loop is shown in Example 5-14.

Example 5-14. A for loop

```
using System;
public class Tester
{

   public static void Main( )
   {
      for ( int counter = 0; counter < 10; counter++ )
      {
         Console.WriteLine(
         "counter: {0} ", counter );
      }
   }
}
```

The output looks like this:

```
counter: 0
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
counter: 6
counter: 7
counter: 8
counter: 9
```

The counter variable is initialized to zero in the initializer:

```
for (int counter=0; counter<10; counter++)
```

The value of counter is tested in the expression part of the header:

```
for (int counter=0; counter<10; counter++)
```

Finally, the value of counter is incremented in the iterator part of the header:

```
for (int counter=0; counter<10; counter++)
```

The initialization part runs only once, when the for loop begins. The integer value counter is created and initialized to zero, and the test is then executed. Because counter is less than 10, the body of the for loop runs and the value is displayed.

After the loop completes, the iterator part of the header runs and counter is incremented. The value of the counter is tested, and, if the test evaluates true, the body of the for statement is executed again.

Your iterator doesn't just have to be ++. You can use --, or any other expression that changes the value of the counter variable, as the needs of your program dictate. Also, for the purposes of a for loop, counter++ and ++counter will have the same result.

The logic of the for loop is as if you said, "For every value of counter that I initialize to zero, take this action if the test returns true, and after the action, update the value of counter."

Controlling a for loop with the modulus operator

The modulus operator really comes into its own in controlling for loops. When you perform modulus *n* on a number that is a multiple of *n*, the result is zero. Thus, 80%10=0 because 80 is an even multiple of 10. This fact allows you to set up loops in which you take an action every *n*th time through the loop by testing a counter to see whether %n is equal to zero, as illustrated in Example 5-15.

Example 5-15. Using modulus to find the tenth iteration

```
using System;
public class Tester
{

   public static void Main( )
   {
      for ( int counter = 1; counter <= 100; counter++ )
      {
         Console.Write( "{0} ", counter );

         if ( counter % 10 == 0 )
         {
            Console.WriteLine( "\t{0}", counter );
```

Example 5-15. Using modulus to find the tenth iteration (continued)

```
        }     // end if
      }       // end for
   }          // end Main
}             // end namespace
```

The output looks like this:

```
1 2 3 4 5 6 7 8 9 10    10
11 12 13 14 15 16 17 18 19 20    20
21 22 23 24 25 26 27 28 29 30    30
31 32 33 34 35 36 37 38 39 40    40
41 42 43 44 45 46 47 48 49 50    50
51 52 53 54 55 56 57 58 59 60    60
61 62 63 64 65 66 67 68 69 70    70
71 72 73 74 75 76 77 78 79 80    80
81 82 83 84 85 86 87 88 89 90    90
91 92 93 94 95 96 97 98 99 100   100
```

In Example 5-15, the value of the counter variable is incremented each time through the loop. Within the loop, the value of counter is compared with the result of modulus 10 (`counter % 10`). When this evaluates to zero, the value of counter is evenly divisible by 10, and the value is printed in the righthand column.

Breaking out of a for loop

It is possible to exit from a `for` loop even before the test condition has been fulfilled. To end a `for` loop prematurely, use the unconditional branching statement `break`.

The `break` statement halts the `for` loop, and execution resumes after the `for` loop statement (or closing brace), as in Example 5-16.

Example 5-16. Using break to exit a for loop

```
using System;
public class Tester
{
   public static void Main( )
   {
      for ( int counter = 0; counter < 10; counter++ )
      {
         Console.WriteLine(
         "counter: {0} ", counter );

         // if condition is met, break out.
         if ( counter == 5 )
         {
            {
               Console.WriteLine( "Breaking out of the loop" );
               break;
            }
         }
```

Example 5-16. Using break to exit a for loop (continued)

```
            Console.WriteLine( "For loop ended" );
        }
    }
}
```

The output looks like this:

```
counter: 0
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
Breaking out of the loop
For loop ended
```

In this for loop, you test whether the value counter is equal to 5. If that value is found (and in this case, it always will be), you break out of the loop.

The continue statement

Rather than breaking out of a loop, you may at times want the semantics of saying, "Don't execute any more statements in this loop, but start the loop again from the top of the next iteration." To accomplish this, use the unconditional branching statement continue.

Break and continue create multiple exit points and make for hard-to-understand, and thus hard-to-maintain, code. Use them with care.

Example 5-17 illustrates the mechanics of both continue and break. This code, suggested to me by one of my technical reviewers, Donald Xie, is intended to create a traffic signal processing system.

Example 5-17. Break and continue

```
using System;
public class Tester
{
   public static int Main( )
   {
      string signal = "0"; // initialize to neutral
      while ( signal != "X" ) // X indicates stop
      {
         Console.Write( "Enter a signal. X = stop. A = Abort: " );
         signal = Console.ReadLine( );

         // do some work here, no matter what signal you
         // receive
         Console.WriteLine( "Received: {0}", signal );
```

Example 5-17. Break and continue (continued)

```
        if ( signal == "A" )
        {
           // faulty - abort signal processing
           // Log the problem and abort.
           Console.WriteLine( "Fault! Abort\n" );
           break;
        }

        if ( signal == "0" )
        {
           // normal traffic condition
           // log and continue on
           Console.WriteLine( "All is well.\n" );
           continue;
        }

        // Problem. Take action and then log the problem
        // and then continue on
        Console.WriteLine( "{0} -- raise alarm!\n", signal );
     }
     return 0;
  }
}
```

The signals are simulated by entering numerals and uppercase characters from the keyboard, using the `Console.ReadLine( )` method, which reads a line of text from the keyboard. `ReadLine( )` reads a line of text into a string variable. The string ends when you press A.

The algorithm is simple: receipt of a "0" (zero) means normal conditions, and no further action is required except to log the event. (In this case, the program simply writes a message to the console; a real application might enter a time-stamped record in a database.)

On receipt of an Abort signal (simulated with an uppercase "A"), the problem is logged and the process is ended. Finally, for any other event, an alarm is raised, perhaps notifying the police. (Note that this sample does not actually notify the police, though it does print out a harrowing message to the console.) If the signal is "X," the alarm is raised but the `while` loop is also terminated.

Here's one sample output:

```
Enter a signal. X = stop. A = Abort: 0
Received: 0
All is well.
Enter a signal. X = stop. A = Abort: 1
Received: 1
1 -- raise alarm!
Enter a signal. X = stop. A = Abort: X
Received: X
X -- raise alarm!
```

Here's a second sample output:

```
Enter a signal. X = stop. A = Abort: A
Received: A
Fault! Abort
```

The point of this exercise is that when the A signal is received, the action in the `if` statement is taken and then the program `breaks` out of the loop, without raising the alarm. When the signal is 0, it is also undesirable to raise the alarm, so the program `continues` from the top of the loop.

Be sure to use uppercase letters for X and A. To keep the code simple, there is no code to check for lowercase letters or other inappropriate input.

Optional for loop header elements

You will remember that the `for` loop header has three parts—initialization, expression, and iteration—and the syntax is as follows:

```
for ([initializers]; [expression]; [iterators]) statement
```

Each part of the `for` loop header is optional. You can, for example, initialize the value outside the `for` loop, as shown in Example 5-18.

Example 5-18. No initialization with for loop

```
using System;
public class Tester
{

   public static void Main( )
   {
      int counter = 0;
      // some work here
      counter = 3;
      // more work here

      for ( ; counter < 10; counter++ )
      {
         Console.WriteLine(
         "counter: {0} ", counter );
      }
   }
}
```

The output looks like this:

```
counter: 3
counter: 4
counter: 5
counter: 6
counter: 7
```

```
counter: 8
counter: 9
```

In this example, the counter variable was initialized and modified before the `for` loop began. Notice that a semicolon is used to hold the place of the missing initialization statement.

You can also leave out the iteration step if you have reason to increment the counter variable inside the loop, as shown in Example 5-19.

Example 5-19. Leaving out the iterator step

```
using System;
public class Tester
{

   public static void Main( )
   {

      for ( int counter = 0; counter < 10; ) // no increment
      {
         Console.WriteLine(
         "counter: {0} ", counter );

         // do more work here

         counter++; // increment counter
      }
   }
}
```

You can mix and match which statements you leave out of a `for` loop.

If you create a `for` loop with no initializer or incrementer, like this:

```
for  ( ; counter < 10 ; )
```

you have a `while` loop in `for` loop's clothing; and of course that construct is silly, and thus not used very often.

It is even possible to leave *all* the statements out, creating what is known as a *forever* loop:

```
for ( ;; )
```

You can create the exact same effect with a `while(true)` loop

```
while ( true )
```

You break out of a forever (or `while(true)`) loop with a `break` statement. A forever loop is shown in Example 5-20.

Example 5-20. A forever loop

```
using System;
public class Tester
{
   public static void Main( )
   {
      int counterVariable = 0; // initialization

      for ( ; ; )  // forever
      {
         Console.WriteLine(
         "counter: {0} ", counterVariable++ ); // increment

         if ( counterVariable > 10 ) // test
            break;
      }
   }
}
```

The output looks like this:

```
counter: 0
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
counter: 6
counter: 7
counter: 8
counter: 9
counter: 10
```

Use a forever loop to indicate that the "normal" case is to continue the loop indefinitely; for example, if your program is waiting for an event to occur somewhere in the system. The conditions for breaking out of the loop would then be exceptional and managed inside the body of the loop.

Although it is possible to use a forever loop to good effect, Example 5-20 is a degenerate case. The initialization, increment, and test would be done more cleanly in the header of the `for` loop, and the program would then be easier to understand. It is shown here to illustrate that a forever loop is possible.

The while (true) construct

You can accomplish exactly the same semantics of a forever loop using the `while (true)` construct, as shown in Example 5-21.

Example 5-21. The while (true) construct

```
using System;
public class Tester
{
   public static void Main( )
   {
      int counterVariable = 0; // initialization

      while ( true )
      {
         Console.WriteLine(
         "counter: {0} ", counterVariable++ ); // increment

         if ( counterVariable > 10 ) // test
            break;
      }
   }
}
```

The output looks like this:

```
counter: 0
counter: 1
counter: 2
counter: 3
counter: 4
counter: 5
counter: 6
counter: 7
counter: 8
counter: 9
counter: 10
```

Example 5-21 is identical to Example 5-20, except that the forever construct:

```
for ( ;; )
```

is replaced with a:

```
while (true)
```

statement. Of course, the keyword `true` always returns the Boolean value true; so like the forever loop, this `while` loop runs until the `break` statement is executed.

Summary

- Branching causes your program to depart from a top-down statement-by-statement execution.
- A method call is the most common form of *unconditional* branching. When the method completes, execution returns to the point where it left off.

- Conditional branching enables your program to branch based on runtime conditions, typically based on the value or relative value of one or more objects or variables.
- The `if` construct executes a statement if a condition is true and skips it otherwise.
- When the condition in an `if` statement is actually two conditions joined by an `or` operator, if the first condition evaluates to `true`, the second condition will not be evaluated at all. This is called short-circuiting.
- The `if/else` construct lets you take one set of actions if the condition tested evaluates true, and a different set of actions if the condition tested evaluates false.
- `if` statements can be nested to evaluate more complex conditions.
- The `switch` statement lets you compare the value of an expression with several constant values (either integers, enumerated constants, or strings), and take action depending on which value matches.
- It is good programming practice for `switch` statements to include a default statement that executes if no other matches are found.
- Iteration, or looping, allows you to take the same action several times consecutively. Iterations are typically controlled by a conditional expression.
- The `goto` statement is used to redirect execution to another point in the program, and its use is typically discouraged.
- The `while` loop executes a block of code while the tested condition evaluates true. The condition is tested before each iteration.
- The `do...while` loop is similar to the `while` loop, but the condition is evaluated at the end of the iteration, so that the iterated statement is guaranteed to be executed at least once.
- The `for` loop is used to execute a statement a specific number of times. The header of the `for` loop can be used to initialize one or more variables, test a logical condition, and modify the variables. The typical use of a `for` loop is to initialize a counter once, test that a condition is using that counter before each iteration, and modify the counter after each iteration.

Quiz

Question 5-1. What statements are generally used for conditional branching?

Question 5-2. True or false: an `if` statement's condition must evaluate to an expression.

Question 5-3. Why should you use braces when there is only one statement following the `if`?

Question 5-4. What kind of expression can be placed in a `switch` statement?

Question 5-5. True or false: you can never fall through in a `switch` statement.

Question 5-6. Name two uses of `goto`.

Question 5-7. What is the difference between `while` and `do...while`?

Question 5-8. What does the keyword `continue` do?

Question 5-9. What are two ways to create a loop that never ends until you hit a `break` statement?

Exercises

Exercise 5-1. Create a method that counts from 1–10 using each of the `while`, `do...while`, and `for` statements.

Exercise 5-2. Create a program that evaluates whether a given input is odd or even; a multiple of 10, or too large (over 100) by using four levels of `if` statement. Then expand the program to do the same work with a `switch` statement.

Exercise 5-3. Create a program that initializes a variable `i` at 0 and counts up, and initializes a second variable `j` at 25 and counts down. Use a `for` loop to increment `i` and decrement `j` simultaneously. When `i` is greater than `j`, end the loop and print out the message "Crossed over!"