WEEK 1

DAY 5

Control Statements

You've learned a lot in the previous four days. This includes knowing how to store information, knowing how to do operations, and even knowing how to avoid executing certain commands by using the `if` statement. You have learned a little about controlling the flow of a program using the `if` statement; however, there are often times where you need to be able to control the flow of a program even more. Today you

- See the other commands to use for program flow control
- Explore how to do even more with the `if` command
- Learn to switch between multiple options
- Investigate how to repeat a block of statements multiple times
- Discover how to abruptly stop the repeating of code

Controlling Program Flow

By controlling the flow of a program, you are able to create functionality that results in something useful. As you continue to program, you will want to

change the flow in a number of additional ways. You will want to repeat a piece of code a number of times, skip a piece of code altogether, or switch between a number of different pieces of code. Regardless of how you want to change the flow of a program, C# has an option for doing it. Most of the changes of flow can be categorized into two types:

- Selection statements
- Iterative statements

Using Selection Statements

Selection statements enable you to execute specific blocks of code based on the results of a condition. The `if` statement that you learned about previously is a selection statement, as is the `switch` statement.

Revisiting `if`

You've learned about the `if` statement; however, it is worth revisiting. Consider the following example:

```
if( gender == 'm' || gender == 'f' )
{
   System.Console.WriteLine("The gender is valid");
}
if( gender != 'm' && gender != 'f' )
{
   System.Console.WriteLine("The gender value, {0} is not valid", gender);
}
```

This example uses a character variable called `gender`. The first `if` statement checks to see whether `gender` is equal to an `'m'` or an `'f'`. This uses the OR operator you learned about in yesterday's lesson. A second `if` statement prints an error message in the case where the gender is not equal to `'m'` or `'f'`. This second `if` statement is an example of making sure that the variable has a valid value. If there is a value other than `'m'` and `'f'`, an error message is displayed.

If you look at these two statements and think that something is just not quite optimal, you are correct. C#, like many other languages, offers another keyword that can be used with the `if` statement: the `else` statement. The `else` statement is used specifically with the `if` statement. The format of the `if...else` statement is

```
if ( condition )
{
   // If condition is true, do these lines
}
else
{
```

```
   // If condition is false, do these lines
}
// code after if... statement
```

The `else` statement gives you the ability to have code that will be executed when the `if` statement's `condition` fails. You should also note that either the block of code after the `if` or the block of code after the `else` will execute—but not both. After either of these blocks of code is done executing, the program jumps to the first line after the `if...else condition`.

Listing 5.1 presents the gender code from earlier. This time, the code has been modified to use the `if...else` command. As you can see in the listing, this version is much more efficient and easier to follow than the one presented earlier.

LISTING 5.1 ifelse.cs—Using the `if...else` Command

```
 1:  //  ifelse.cs - Using the if...else statement
 2:  //---------------------------------------------------------------------
 3:
 4:  class ifelse
 5:  {
 6:     static void Main()
 7:     {
 8:        char gender = 'x';
 9:
10:        if( gender == 'm' || gender == 'f' )
11:        {
12:           System.Console.WriteLine("The gender is valid");
13:        }
14:        else
15:        {
16:           System.Console.WriteLine("The gender value, {0}, is not valid",
➥gender);
17:        }
18:        System.Console.WriteLine("The if statement is now over!");
18:     }
19:  }
```

OUTPUT

```
The gender value, x, is not valid
The if statement is now over!
```

ANALYSIS This listing declares a simple variable called `gender` of type `char` in line 8. This variable is set to a value of `'x'` when it is declared. The `if` statement starts in line 10. Line 10 checks to see whether `gender` is either `'m'` or `'f'`. If it is, a message is printed in line 12 saying that `gender` is valid. If `gender` is not `'m'` or `'f'`, the `if condition` fails and control is passed to the `else` statement in line 14. In this case, `gender` is

equal to `'x'`, so the `else` command is executed. A message is printed stating that the `gender` value is invalid. Control is then passed to the first line after the `if...else` statement—line 18.

Modify line 8 to set the value of `gender` to either `'m'` or `'f'`. Recompile and rerun the program. This time the output will be

OUTPUT

```
The gender is valid
The if statement is now over!
```

What would you expect to happen if you set the value of gender to a capital `M` or `F`? Remember, C# is case sensitive. A capital letter is not the same thing as a lowercase letter.

Nesting and Stacking `if` Statements

NEW TERM *Nesting* is simply the inclusion of one statement within another. Almost all C# flow commands can be nested within each other.

To nest an `if` statement, you place a second `if` statement within the first. You can nest within the `if` section or the `else` section. Using the `gender` example, you could do the following to make the statement a little more effective (the nested statement appears in bold):

```
if( gender == 'm' )
{
   // it is a male
}
else
{
   if ( gender == 'f' )
   {
      // it is a female
   }
   else
   {
      //neither a male or a female
   }
}
```

A complete `if...else` statement is nested within the `else` section of the original `if` statement. This code operates just as you expect. If `gender` is not equal to `'m'`, the flow goes to the first `else` statement. Within this `else` statement is another `if` statement that starts from its beginning. This second `if` statement checks to see whether the gender is

equal to 'f'. If not, the flow goes to the else statement of the nested if. At that point, you know that gender is neither 'm' nor 'f', and you can add appropriate coding logic.

While nesting makes some functionality easier, you can also stack if statements. In the example of checking gender, stacking is actually a much better solution.

Stacking if Statements

Stacking if statements combines the else with another if. The easiest way to understand stacking is to see the gender example one more time, stacked (see Listing 5.2).

LISTING 5.2 stacking.cs—Stacking an if Statement

```
 1:  //  stacked.cs - Using the if...else statement
 2:  //---------------------------------------------------------------------
 3:
 4:  class stacked
 5:  {
 6:     static void Main()
 7:     {
 8:        char gender = 'x';
 9:
10:        if( gender == 'm' )
11:        {
12:           System.Console.WriteLine("The gender is male");
13:        }
14:        else if ( gender == 'f' )
15:        {
16:           System.Console.WriteLine("The gender is female");
17:        }
18:        else
19:        {
20:           System.Console.WriteLine("The gender value, {0}, is not valid",
➥gender);
21:        }
22:        System.Console.WriteLine("The if statement is now over!");
23:     }
24:  }
```

OUTPUT

```
The gender value, x, is not valid
The if statement is now over!
```

ANALYSIS The code presented in this example is very close to the code presented in the previous example. The primary difference is in line 14. The else statement is immediately followed by an if. There are no braces or a block. The format for stacking is

```
if ( condition 1 )
{
```

```
   // do something about condition 1
}
else if ( condition 2 )
{
   // do something about condition 2
}
else if ( condition 3 )
{
   // do something about condition 3
}
else if ( condition x )
{
    // do something about condition x
else
{
   // All previous conditions failed
}
```

This is relatively easy to follow. With the `gender` example, you had only two conditions. There are times when you might have more than two. For example, you could create a computer program that checks the roll of a die. You could then do something different depending on what the roll is. Each stacked condition could check for a different number (from 1 to 6) with the final `else` statement presenting an error because there can be only six numbers. The code for this would be

```
if (roll == 1)
   // roll is 1
else if (roll == 2)
         // roll is 2
else if (roll == 3)
         // roll is 3
else if (roll == 4)
         // roll is 4
else if (roll == 5)
         // roll is 5
else if (roll == 6)
         // roll is 6
else
   // it isn't a number from 1 to 6
```

This code is relatively easy to follow because it's easy to see that each of the 6 possible numbers is checked against the roll. If the roll is not one of the 6, the final `else` statement can take care of any error logic or reset logic.

As you can see in the die code, there are no braces used around the `if` statements. If you are using only a single statement within the `if` or the `else`, you don't need the braces. You include them only when you have more than one statement.

The `switch` Statement

C# provides a much easier way to modify program flow based on multiple values stored in a variable: the `switch` statement. The format of the `switch` statement is

```
switch ( value )
{
    case result_1 :
            // do stuff for result_1
            break;
    case result_2 :
            // do stuff for result_2
            break;
    ...
    case result_n :
            // do stuff for result_x
            break;
    default:
            // do stuff for default case
            break;
}
```

You can see by the format of the `switch` statement that there is no `condition`. Rather a *`value`* is used. This value can be the result of an expression, or it can be a variable. This value is then compared to each of the values in each of the `case` statements until a match is found. If a match is not found, the flow goes to the `default` case. If there is not a `default` case, the flow goes to the first statement following the `switch` statement.

When a match is found, the code within the matching `case` statement is executed. When the flow reaches another `case` statement, the `switch` statement is ended. Only one `case` statement will be executed at most. Flow then continues with the first command following the `switch` statement. Listing 5.3 shows the `switch` statement in action, using the earlier example of a roll of a six-sided die.

LISTING 5.3 roll.cs—Using the `switch` Statement with the Roll of a Die

```
 1:  //  roll.cs- Using the switch statement.
 2:  //---------------------------------------------------------------------
 3:
 4:  class roll
 5:  {
 6:     public static void Main()
 7:     {
 8:        int roll = 0;
 9:
10:        // The next two lines set the roll to a random number from 1 to 6
11:        System.Random rnd = new System.Random();
12:        roll = (int) rnd.Next(1,7);
```

LISTING 5.3 continued

```
13:
14:         System.Console.WriteLine("Starting the switch... ");
15:
16:         switch (roll)
17:         {
18:             case 1:
19:                         System.Console.WriteLine("Roll is 1");
20:                         break;
21:             case 2:
22:                         System.Console.WriteLine("Roll is 2");
23:                         break;
24:             case 3:
25:                         System.Console.WriteLine("Roll is 3");
26:                         break;
27:             case 4:
28:                         System.Console.WriteLine("Roll is 4");
29:                         break;
30:             case 5:
31:                         System.Console.WriteLine("Roll is 5");
32:                         break;
33:             case 6:
34:                         System.Console.WriteLine("Roll is 6");
35:                         break;
36:             default:
37:                         System.Console.WriteLine("Roll is not 1 through 6");
38:                         break;
39:         }
40:         System.Console.WriteLine("The switch statement is now over!");
41:     }
42: }
```

OUTPUT

```
Starting the switch...
Roll is 1
The switch statement is now over!
```

Your answer for the roll in the output might be a number other than 1.

ANALYSIS This listing is a little longer than a lot of the previous listings; however, it is also more functional. The first thing to focus on is lines 16 to 39. These lines contain the `switch` statement that is the center of this discussion. The `switch` statement uses the value stored in the roll. Depending on the value, one of the cases will be selected. If the number is something other than 1 though 6, the `default` statement starting in line 39 is

executed. If any of the numbers are rolled (1 through 6), the appropriate `case` statement is executed.

You should note that at the end of each section of code for each `case` statement there is a `break` command, which is required at the end of each set of code. This signals the end of the statements within a `case`. If you don't include the `break` command, you will get a compiler error.

To make this listing a more interesting, lines 11 and 12 were added. Line 11 might look unfamiliar. This line creates a variable called `rnd`, which is an object that will hold a random number. In tomorrow's lesson, you revisit this line of code and learn the details of what it is doing. For now, simply know that it is setting up a variable for a random number.

Line 12 is also a line that will become more familiar over the next few days. The command, `(int) rnd.Next(1,7)` provides a random number from 1 to 6.

You can use lines 11 and 12 to generate random numbers for any range by simply changing the values from 1 and 7 to the range you want numbers between. The first number is the lowest number that will be returned. The second number is one higher than the highest number that will be returned. For example, if you wanted a random number from 90 to 100, you could change line 12 to:

```
Roll = (int) rnd.Next(90, 101);
```

Multiple Cases for Single Solutions

Sometimes you might want to execute the same piece of code for multiple values. For example, if you want to switch based on the roll of a six-sided die, but you want to do something based only on odd or even numbers, you could group multiple `case` statements. The `switch` statement is

```
switch (roll)
{
     case 1:
     case 3:
     case 5:
               System.Console.WriteLine("Roll is odd");
               break;
     case 2:
     case 4:
     case 6:
               System.Console.WriteLine("Roll is even");
               break;
```

```
     default:
              System.Console.WriteLine("Roll is not 1 through 6");
              break;
}
```

The same code is executed if the roll is 1, 3, or 5. Additionally, the same code is executed if the roll is 2, 4, or 6.

If you are a C++ programmer, you learned that you could have code execute from multiple `case` statements by leaving out the `break` command. This causes the code to drop through to the next `case` statement. In C#, this is not valid. Code cannot drop through from one case to another. This means that if you are going to group `case` statements, you cannot place any code between them. You can place it only after the last `case` statement in each group.

Executing More than One `case` Statement

There are times when you might want to execute more than one `case` statement within a `switch` statement. To do this in C#, you can use the `goto` command. The `goto` command can be used within the `switch` statement to go to either a `case` statement or to the default command. The following code snippet shows the `switch` statement from the previous section executed with `goto` statements instead of simply dropping through:

```
switch (roll)
{
     case 1:
               goto case 5;
               break;
     case 2:
               goto case 6;
               break;
     case 3:
               goto case 5;
               break;
     case 4:
               goto case 6;
               break;
     case 5:
               System.Console.WriteLine("Roll is odd");
               break;
     case 6:
               System.Console.WriteLine("Roll is even");
               break;
     default:
```

```
                System.Console.WriteLine("Roll is not 1 through 6");
                break;
}
```

Although this example illustrates using the `goto`, it is much easier to use the previous example of grouping multiple `case` statements. You will find there are times, however, when the `goto` provides the solution you need.

Governing Types for `switch` Statements

A `switch` statement has only certain types that can be used. The data type—or the "governing type" for a `switch` statement—is the type that the `switch` statement's expression resolves to. If this governing type is `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, or a text string, this type is the governing type. There is another type called an `enum` that is also valid as a governing type. You will learn about `enum` types on Day 8, "Advanced Data Storage: Structures, Enumerators, and Arrays."

If the data type of the expression is something other than these types, the type must have a single implicit conversion that converts it to a type of `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, or a string. If there isn't a conversion available, or if there is more than one, you get an error when you compile your program.

If you don't remember what implicit conversions are, you should review Day 3, "Storing Information with Variables."

Do	Don't
DO use a `switch` statement when you are checking for multiple different values in the same variable.	**DON'T** accidentally put a semicolon after the condition of a `switch` or `if` statement: `if ( condition );`

Using Iteration Statements

In addition to changing the flow through selection statements, there are times when you might want to repeat a piece of code multiple times. For times when you want to repeat code, C# provides a number of iteration statements. Iteration statements can execute a block of code zero or more times. Each execution of the code is a single iteration.

The iteration statements in C# are

- `while`
- `do`
- `for`
- `foreach`

The `while` Statement

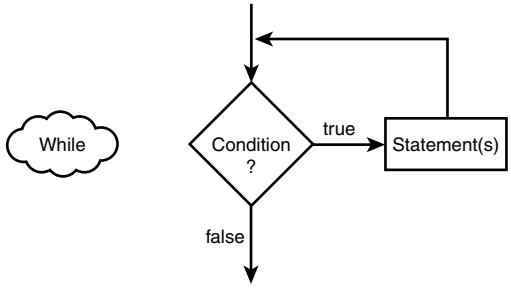
The `while` command is used to repeat a block of code as long as a condition is true. The format of the `while` statement is

```
while ( condition )
    Statement(s)
```

This format is also presented in Figure 5.1.

FIGURE 5.1

The `while` *command.*



As you can see from the figure, a `while` statement uses a conditional statement. If this conditional statement evaluates to true, the statement(s) are executed. If the condition evaluates to false, the statements are not executed and program flow goes to the next command following the `while` statement. Listing 5.4 presents a `while` statement that enables you to print the average of 10 random numbers from 1 to 10.

LISTING 5.4 average.cs—Using the `while` Command

```
1:  //  average.cs Using the while statement.
2:  //  print the average of 10 random numbers that are from 1 to 10.
3:  //---------------------------------------------------------------------
4:
5:  class average
6:  {
7:     public static void Main()
8:     {
9:        int ttl = 0;  // variable to store the running total
```

LISTING 5.4 continued

```
10:        int nbr = 0;  // variable for individual numbers
11:        int ctr = 0;  // counter
12:
13:        System.Random rnd = new System.Random();  // random number
14:
15:        while ( ctr < 10 )
16:        {
17:            //Get random number
18:            nbr = (int) rnd.Next(1,11);
19:
20:            System.Console.WriteLine("Number {0} is {1}", (ctr + 1), nbr);
21:
22:            ttl += nbr;        //add nbr to total
23:            ctr++;             //increment counter
24:        }
25:
26:        System.Console.WriteLine("\nThe total of the {0} numbers is {1}",
➥ctr, ttl);
27:        System.Console.WriteLine("\nThe average of the numbers is {0}",
➥ttl/ctr );
28:     }
29:  }
```

The numbers in your output will differ from those shown here. Because random numbers are assigned, each time you run the program, the numbers will be different.

OUTPUT

```
Number 1 is 2
Number 2 is 5
Number 3 is 4
Number 4 is 1
Number 5 is 1
Number 6 is 5
Number 7 is 2
Number 8 is 5
Number 9 is 10
Number 10 is 2

The total of the 10 numbers is 37

The average of the numbers is 3
```

ANALYSIS This listing uses the code for random numbers that you saw earlier in today's lesson. Instead of a random number from 1 to 6, this code picks numbers from

1 to 10. You see this in line 18, where the value of 10 is multiplied against the next random number. Line 13 initialized the random variable before it was used in this manner.

The `while` statement starts in line 15. The condition for this `while` statement is a simple check to see whether a counter is less than 10. Because the counter was initialized to `0` in line 11, this condition evaluates to true, so the statements within the `while` will be executed. This `while` statement simply gets a random number from 1 to 10 in line 18 and adds it to the total counter, `ttl` in line 22. Line 23 then increments the counter variable, `ctr`. After this increment, the end of the `while` is reached in line 24. The flow of the program is automatically put back to the `while` condition in line 15. This condition is reevaluated to see whether it is still true. If true, the statements are executed again. This continues to happen until the `while` condition fails. For this program, the failure occurs when `ctr` becomes `10`. At that point, the flow goes to line 25 which immediately follows the `while` statement.

The code after the `while` statement prints the total and the average of the 10 random numbers that were found. The program then ends.

For a `while` statement to eventually end, you must make sure that you change something in the statement(s) that will impact the condition. If your condition can never be false, your `while` statement could end up in an infinite loop. There is one alternative to creating a false condition: the `break` statement. This is covered in the next section.

Breaking Out Of or Continuing a `while` Statement

It is possible to end a `while` statement before the condition is set to false. It is also possible to end an iteration of a `while` statement before getting to the end of the statements.

To break out of a `while` and thus end it early, you use the `break` command. A `break` immediately takes control to the first command after the `while`.

You can also cause a `while` statement to jump immediately to the next iteration. This is done by using the `continue` statement. The `continue` statement causes the program's flow to go to the condition statement of the `while`. Listing 5.5 illustrates both the `continue` and `break` statements within a `while`.

LISTING 5.5 even.cs—Using `break` and `continue`

```
1:  //  even.cs- Using the while with the break and continue commands.
2:  //-------------------------------------------------------------------
3:
```

LISTING 5.5 continued

```
 4:  class even
 5:  {
 6:     public static void Main()
 7:     {
 8:        int ctr = 0;
 9:
10:        while (true)
11:        {
12:           ctr++;
13:
14:           if (ctr > 10 )
15:           {
16:              break;
17:           }
18:           else if ( (ctr % 2) == 1 )
19:           {
20:              continue;
21:           }
22:           else
23:           {
24:              System.Console.WriteLine("...{0}...", ctr);
25:           }
26:        }
27:        System.Console.WriteLine("Done!");
28:     }
29:  }
```

OUTPUT

```
...2...
...4...
...6...
...8...
...10...
Done!
```

ANALYSIS This listing prints even numbers and skips odd numbers. When the value of the counter is greater than 10, the `while` statement is ended with a `break` statement.

This listing declares and sets a counter variable, `ctr`, to `0` in line 8. A `while` statement is then started in line 10. Because a `break` is used to end the loop, the condition in line 10 is simply set to `true`. This, in effect, creates an infinite loop. Because this is an infinite loop, a `break` statement is needed to end the `while` statement's iterations. The first thing done in the `while` statement is that `ctr` is incremented in line 12. Line 14 then checks to see whether the `ctr` is greater than 10. If `ctr` is greater than 10, line 16 executes a `break` statement, which ends the `while` and sends the program flow to line 27.

If `ctr` is less than 10, the `else` statement in line 18 is executed. This `else` statement is stacked with an `if` statement that checks to see whether the current number is odd. This is done using the modulus operator. If the counter is even, by using the modulus operator with `2`, you get a result of `0`. If it is odd, you get a result of `1`. When an odd number is found, the `continue` statement is called in line 20. This sends control back to the top of the `while` statement where the condition is checked again. Because the condition is always true (literally), the `while` statement's statements are executed again. This starts with the increment of the counter in line 12 again, followed by the checks.

If the number is not odd, the `else` statement in line 22 will execute. This final `else` statement contains a single call to `WriteLine`, which prints the counter's value.

The `do` Statement

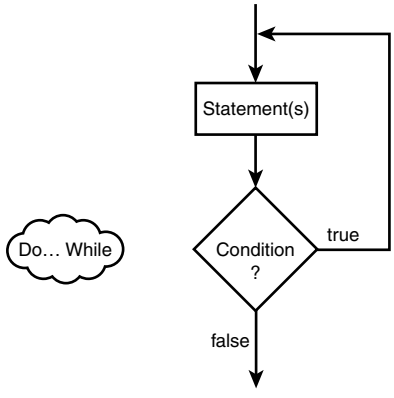
If a `while` statement's condition is false on the initial check, the `while` statements will never execute. There are times, however, when you want statements to execute at least once. For these times, the `do` statement might be a better solution.

The format of the `do` statement is

```
do
Statement(s)
while ( condition );
```

This format is also presented in Figure 5.2.

FIGURE 5.2
The `do` *command.*



As you can see from the figure, a `do` statement first executes its statements. After executing the statements, a `while` statement is presented with a condition. This `while` statement and condition operate the same as the `while` you explored earlier in listing 5.4. If the condition evaluates to true, program flow returns to the statements. If the condition

evaluates to false, the flow goes to the next line after the `do...while`. Listing 5.6 presents a `do` command in action.

Because of the use of the `while` with the `do` statement, a `do` statement is often referred to as a `do...while` statement.

LISTING 5.6 do_it.cs—The `do` Command in Action

```
 1:  //  do_it.cs Using the do statement.
 2:  //  Get random numbers (from 1 to 10) until a 5 is reached.
 3:  //---------------------------------------------------------------------
 4:
 5:  class do_it
 6:  {
 7:     public static void Main()
 8:     {
 9:        int ttl = 0;  // variable to store the running total
10:        int nbr = 0;  // variable for individual numbers
11:        int ctr = 0;  // counter
12:
13:        System.Random rnd = new System.Random();  // random number
14:
15:        do
16:        {
17:            //Get random number
18:            nbr = (int) rnd.Next(1,11);
19:
20:            ctr++;             //number of numbers counted
21:            ttl += nbr;        //add nbr to total of numbers
22:
23:            System.Console.WriteLine("Number {0} is {1}", ctr, nbr);
24:
25:        } while ( nbr != 5 );
26:
27:        System.Console.WriteLine("\n{0} numbers were read", ctr);
28:        System.Console.WriteLine("The total of the numbers is {0}", ttl);
29:        System.Console.WriteLine("The average of the numbers is {0}", ttl/ctr
➥);
30:     }
31:  }
```

OUTPUT

```
Number 1 is 1
Number 2 is 6
Number 3 is 5

3 numbers were read
```

```
The total of the numbers is 12
The average of the numbers is 4
```

ANALYSIS As with the previous listings that used random numbers, your output will most likely be different from what is displayed. You will have a list of numbers, ending with `5`.

For this program, you want to do something at least once—get a random number. You want to then keep doing this until you have a condition met—you get a `5`. This is a great scenario for the `do` statement. This listing is very similar to an earlier listing. In lines 9 to 11, you set up a number of variables to keep track of totals and counts. In line 13, you again set up a variable to get random numbers.

Line 15 is the start of your `do` statement. The body of the `do` (lines 16 to 24) is executed. First, the next random number is obtained. Again, this is a number from 1 to 10 that is assigned to the variable `nbr`. Line 20 keeps track of how many numbers have been obtained by adding `1` to `ctr` each time a number is read. Line 21 then adds the value of the number read to the total. Remember, the following code:

```
ttl += nbr
```

is the same as

```
ttl = ttl + nbr
```

Line 23 prints the obtained number to the screen with the count of which number it is.

Line 25 is the conditional portion of the `do` statement. In this case, the condition is that `nbr` is not equal to `5`. As long as the number obtained, `nbr`, is not equal to `5`, the body of the `do` statement will continue to execute. When a `5` is received, the loop ends. If you look in the output of your program, you will find that there is always only one `5`, and that it is always the last number.

Lines 27, 28, and 29 prints statistical information regarding the numbers you found.

The `for` Statement

Although the `do...while` and the `while` statement give you all the functionality you really need to control iterations of code, they are not the only commands available. Before looking at the `for` statement, check out the code in the following snippet:

```
ctr = 1;
while ( ctr < 10 )
{
   //do some stuff
   ctr++;
}
```

In looking at this snippet of code, you can see that a counter is used to loop through a `while` statement. The flow if this code is this:

Step 1: Set a counter to the value of `1`.

Step 2: Check to see whether the counter is less than `10`.

If the counter is not less than `10` (the condition fails), go to the end.

Step 3: Do some stuff.

Step 4: Add `1` to the counter.

Step 5: Go to Step 2.

These steps are a very common use of iteration. Because this is a common use, you are provided with the `for` statement, which consolidates the steps into a much simpler format:

```
for ( initializer; condition; incrementor )
Statement(s);
```

You should review the format presented here for the `for` statement, which contains three parts within parentheses: the `initializer`, the `condition`, and the `incrementor`. Each of these three parts is separated by a semicolon. If one of these expressions is to be left out, you still need to include the semicolon separators.

The `initializer` is executed when the `for` statement begins. It is executed only once at the beginning and then never again.

After executing the `initializer`, the `condition` statement is evaluated. Just as the condition in the `while` statement, this must evaluate to either true or false. If this evaluates to true, the statement(s) are executed.

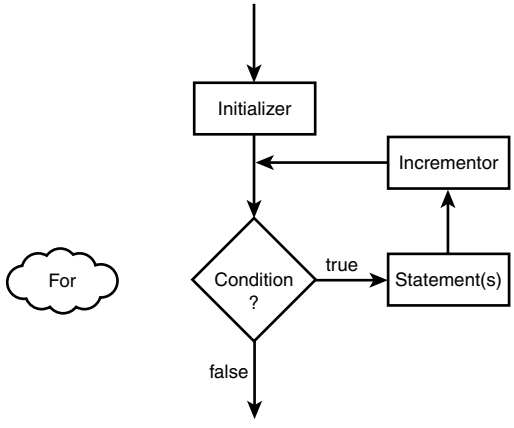
After the statement or statement block executes, program flow is returned to the `for` statement where the `incrementor` is evaluated. This `incrementor` can actually be any valid C# expression; however, it is generally used to increment a counter.

After the `incrementor` is executed, the `condition` is again evaluated. As long as the `condition` remains true, the statements will be executed, followed by the `incrementor`. This continues until the `condition` evaluates to false. Figure 5.3 illustrates the flow of the `for` statement.

Before jumping into a listing, let's revisit the `while` statement that was presented at the beginning of this section:

```
for ( ctr = 1; ctr < 10; ctr++ )
{
   //do some stuff
}
```

FIGURE 5.3
The `for` *statement.*



This `for` statement is much simpler than the code used earlier with the `while` statement. The steps that this `for` statement executes are

Step 1: Set a counter to the value of `1`.

Step 2: Check to see whether the counter is less than `10`.

If the counter is not less than `10` (condition fails), go to the end of the `for` statement.

Step 3: Do some stuff.

Step 4: Add `1` to the counter.

Step 5: Go to Step 2.

These are the same steps that were followed with the `while` statement snippet earlier. The difference is that the `for` statement is much more concise and easier to follow. Listing 5.7 presents a more robust use of the `for` statement. In fact, this is the same program you saw in sample code earlier, only now it is much more concise.

LISTING 5.7 foravg.cs—Using the `for` Statement

```
 1:  //  foravg.cs Using the for statement.
 2:  //  print the average of 10 random numbers that are from 1 to 10.
 3:  //---------------------------------------------------------------------
 4:
 5:  class average
 6:  {
 7:     public static void Main()
 8:     {
 9:        int ttl = 0;  // variable to store the running total
10:        int nbr = 0;  // variable for individual numbers
```

LISTING 5.7 continued

```
11:        int ctr = 0;  // counter
12:
13:        System.Random rnd = new System.Random();  // random number
14:
15:        for ( ctr = 1; ctr <= 10; ctr++ )
16:        {
17:            //Get random number
18:            nbr = (int) rnd.Next(1,11);
19:
20:            System.Console.WriteLine("Number {0} is {1}", (ctr), nbr);
21:
22:            ttl += nbr;        //add nbr to total
23:        }
24:
25:        System.Console.WriteLine("\nThe total of the 10 numbers is {0}",
➥ttl);
26:        System.Console.WriteLine("\nThe average of the numbers is {0}",
➥ttl/10 );
27:     }
28:  }
```

OUTPUT

```
Number 1 is 10
Number 2 is 3
Number 3 is 6
Number 4 is 5
Number 5 is 7
Number 6 is 8
Number 7 is 7
Number 8 is 1
Number 9 is 4
Number 10 is 3

The total of the 10 numbers is 54

The average of the numbers is 5
```

ANALYSIS Much of this listing is identical to what you saw earlier in today's lessons. You should note, however, the difference. In line 15, you see the use of the `for` statement. The counter is initialized to `1`, which makes it easier to display the value in the `WriteLine` routine in line 20. The condition statement in the `for` statement is adjusted appropriately as well.

What happens when the program flow reaches the `for` statement? Simply put, the counter is set to `1`. It is then verified against the condition. In this case, the counter is less than or equal to 10, so the body of the `for` statement is executed. When the body in lines 16 to 23 is done executing, control goes back to the `incrementor` of the `for` statement in line 15. In this `for` statement's `incrementor`, the counter is incremented by `1`. The condition is then checked again and if true, the body of the `for` statement executes again. This

continues until the condition fails. For this program, this happens when the counter is set to 11.

The `for` Statement Expressions

You can do a lot with the `initializer`, `condition`, and `incrementor`. You can actually put any expressions within these areas. You can even put in more than one expression.

If you use more than one expression within one of the segments of the `for` statement, you need to separate them. The separator control is used to do this. The separator control is the comma. As an example, the following `for` statement initializes two variables and increments both:

```
for ( x = 1, y = 1; x + y < 100; x++, y++ )
   // Do something...
```

In addition to being able to do multiple expressions, you also are not restricted to using each of the parts of a `for` statement as described. The following example actually does all of the work in the `for` statement's control structure. The body of the `for` statement is an empty statement—a semicolon:

```
for ( x = 0; ++x <= 10; System.Console.WriteLine("{0}", x) )
   ;
```

This simple line of code actually does quite a lot. If you enter this into a program, it prints the numbers `1` to `10`. You're asked to turn this into a complete listing in one of today's exercises at the end of the lesson.

You should be careful about how much you do within the `for` statement's control structures. You want to make sure you don't make your code too complicated to follow.

The `foreach` Statement

The `foreach` statement iterates in a way similar to the `for` statement. The `foreach` statement, however, has a special purpose. It can loop through collections such as arrays. The `foreach` statement, collections, and arrays are covered on Day 8.

Revisiting `break` and `continue`

The use of `break` and `continue` were presented earlier with the `while` statement. Additionally, you saw the use of the `break` command with the `switch` statement. These two commands can also be used with the other program flow statements.

In the `do...while` statement, `break` and `continue` operate exactly like the `while` statement. The `continue` command loops to the conditional statement. The `break` command sends the program flow to the statement following the `do...while`.

With the `for` statement, the `continue` statement sends control to the `incrementor` statement. The condition is then checked, and if true, the `for` statement continues to loop. The `break` statement sends the program flow to the statement following the `for` statement.

The `break` command exits the current routine. The `continue` command starts the next iteration.

Using `goto`

The `goto` statement is fraught with controversy, regardless of the programming language you use. Because the `goto` statement can unconditionally change program flow, it is very powerful. With power comes responsibility. Many developers avoid the `goto` statement because it is easy to create code that is hard to follow.

There are three ways the `goto` statement can be used. As you saw earlier, the `switch` statement is home to two of the uses of `goto`: `goto case` and `goto default`. You saw these in action earlier in the discussion on the `switch` statement.

The third `goto` statement takes the following format:

```
goto label;
```

With this form of the `goto` statement, you are sending the control of the program to a label statement.

Labeled Statements

A label statement is simply a command that marks a location. The format of a label is

```
label_name:
```

Notice that this is followed by a colon, not a semicolon. Listing 5.8 presents the `goto` statement being used with labels.

LISTING 5.8 Score.cs—Using the `goto` Statement with a Label

```
1:  //  score.cs    Using the goto and label statements.
2:  //  Disclaimer: This program shows the use of goto and label
3:  //              This is not a good use; however, it illustrates
4:  //              the functionality of these keywords.
```

LISTING 5.8 continued

```
 5:  //-------------------------------------------------------------------
 6:
 7:  class score
 8:  {
 9:     public static void Main()
10:     {
11:        int score = 0;
12:        int ctr = 0;
13:
14:        System.Random rnd = new System.Random();
15:
16:        Start:
17:
18:        ctr++;
19:
20:        if (ctr > 10)
21:            goto EndThis;
22:        else
23:            score = (int) rnd.Next(60, 101);
24:
25:        System.Console.WriteLine("{0} - You received a score of {1}",
26:                     ctr, score);
27:
28:        goto Start;
29:
30:       EndThis:
31:
32:        System.Console.WriteLine("Done with scores!");
33:     }
34:  }
```

OUTPUT

```
1 - You received a score of 83
2 - You received a score of 99
3 - You received a score of 72
4 - You received a score of 67
5 - You received a score of 80
6 - You received a score of 98
7 - You received a score of 64
8 - You received a score of 91
9 - You received a score of 79
10 - You received a score of 76
Done with scores!
```

ANALYSIS The purpose of this listing is relatively simple; it prints 10 scores that are obtained by getting 10 random numbers from 60 to 100. This use of random numbers is similar to what you've seen before except for one small change. In line 23, instead of starting at `1` for the number to be obtained, you start at `60`. Additionally,

because the numbers you want are from 60 to 100, the upper limit is set to 101. By using 101 as the second number, you get a number less than 101.

The focus of this listing is lines 16, 21, 28, and 30. In line 16 you see a label called `Start`. Because this is a label, the program flow skips past this line and goes to line 18 where a counter is incremented. In line 20, the condition within an `if` statement is checked. If the counter is greater than 10, a `goto` statement in line 21 is executed, which sends program flow to the `EndThis` label in line 30. Because the counter is not greater than 10, program flow goes to the `else` statement in line 22. The `else` statement gets the random score in line 23 that was already covered. Line 25 prints the score obtained. Program flow then hits line 28, which sends the flow unconditionally to the `Start` label. Because the `Start` label is in line 16, program flow goes back to line 16.

This listing does a similar iteration to what can be done with the `while`, `do`, or `for` statements. In many cases, you will find there are programming alternatives to using `goto`. If there is a different option, use it first.

Avoid using the `goto` whenever possible. It can lead to what is referred to as *spaghetti code*. Spaghetti code is code that winds all over the place and is therefore hard to follow from one end to the next.

Nesting Flow

All of the program flow commands from today can be nested. When nesting program flow commands, make sure that the commands are ended appropriately. You can create a logic error and sometimes a syntax error if you don't nest properly.

Do	DON'T
DO comment your code to make it clearer what the program and program flow is doing.	**DON'T** use a `goto` statement unless it is absolutely necessary.

Summary

You learned a lot in today's lesson, and you'll use this knowledge in virtually every C# application you create.

In today's lesson, you once again covered some of the constructs that are part of the basic C# language. You first expanded on your knowledge of the `if` statement by learning about

the `else` statement. You then learned about another selection statement, the `switch`. Selection statements were followed by a discussion of iterative program flow control statements. This included use of the `while`, `do`, and `for` statements. You learned that there is another command called the `foreach` that you will learn about on Day 8. In addition to learning how to use these commands, you also learned that they can be nested within each other. Finally, you learned about the `goto` statement and how it can be used with `case`, `default`, or labels.

Q&A

Q **Are there other types of control statements?**

A Yes—`throw`, `try`, `catch`, and `finally`. You will learn about these in future lessons.

Q **Can you use a text string with a `switch` statement?**

A Yes. A string is a "governing type" for `switch` statements. This means that you can use a variable that holds a string in the `switch` and then use string values in the `case` statements. Remember, a string is simply text in quotation marks. In one of the exercises, you create a `switch` statement that works with strings.

Q **Why is the `goto` considered so bad?**

A The `goto` statement has gotten a bad rap. If used cautiously and in a structured, organized manner, the `goto` statement can help solve a number of programming problems. The use of `goto case` and `goto default` are prime examples of good uses of `goto`. `goto` has a bad rap because the `goto` statement is often not used cleanly; programmers use it to get from one piece of code to another quickly and in an unstructured manner. In an object-oriented programming language, the more structure you can keep in your programs, the better—and more maintainable—they will be.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to understand the quiz and exercise answers before continuing to the next day's lesson. Answers are provided in Appendix A, "Answers."

Quiz

1. What commands are provided by C# for repeating lines of code multiple times?
2. What is the fewest times the statements in a `while` will execute?

3. What is the fewest times the statements in a `do` will execute?
4. Consider the following `for` statement:

   ```
   for ( x = 1; x == 1; x++ )
   ```

   What is the conditional statement?
5. In the `for` statement in question 4, what is the incrementor statement?
6. What statement is used to end a `case` expression in a `select` statement?
7. What punctuation character is used with a label?
8. What punctuation is used to separate multiple expressions in a `for` statement?
9. What is nesting?
10. What command is used to jump to the next iteration of a loop?

Exercises

1. Write an `if` statement that checks to see whether a variable called `file-type` is `'s'`, `'m'`, or `'j'`. Print the following message based on the `file-type`:

   ```
   s    The filer is single
   m    The filer is married filing at the single rate
   j    The filer is married filing at the joint rate
   ```

2. Is the following `if` statement valid? If so, what is the value of `x` after this code executes?

   ```
   int x = 2;
   int y = 3;
   if (x==2) if (y>3) x=5; else x=9;
   ```

3. Write a `while` loop that counts from 99 to 1.
4. Rewrite the `while` loop in exercise 3 as a `for` loop.
5. **BUG BUSTER:** Is the following listing correct? If so, what does it do? If not, what is wrong with the listing (Ex5-5.cs)?

   ```
   //  Ex5-5.cs. Exercise 5 for Day 5
   //-------------------------------------------------------------------

   class score
   {
      public static void Main()
      {
         int score = 99;

         if ( score == 100 );
         {
             System.Console.WriteLine("You got a perfect score!");
   ```

```
        }
        else
            System.Console.WriteLine("Bummer, you were not perfect!");
    }
}
```

6. Create a `for` loop that prints the numbers `1` to `10` all within the initializer, condition, and incrementor sections of the `for`. The body of the `for` should be an empty statement.
7. Write the code for a `switch` statement that switches on the variable `name`. If the name is `"Robert"`, print a message that says `"Hi Bob"`. If the name is `"Richard"`, print a message that says `"Hi Rich"`. If the name is `"Barbara"`, print a message that says `"Hi Barb"`. If the name is `"Kalee"`, print a message that says `"You Go Girl!"`. On any other name, print a message that says `"Hi x"` where *x* is the person's name.
8. Write a program to roll a six-sided die 100 times. Print the number of times each of the sides of the die was rolled.