Chapter 7

Contax T / Kodak Tri-X

Amsterdam

Controlling The Flow Of Program Execution

Learning Objectives

- *State the difference between selection and iteration statements*
- *Describe the purpose and use of the "if" statement*
- *Describe the purpose and use of the "if/else" statement*
- *Describe the purpose and use of the "for" statement*
- *Describe the purpose and use of the "while" statement*
- *Describe the purpose and use of the "do/while" statement*
- *Describe the purpose and use of chained "if/else" statements*
- *Describe the purpose and use of nested "for" statements*
- *Describe the purpose and use of the "switch" statement*
- *Describe the purpose and use of the "break" and "continue" statements*
- *Describe the purpose and use of the "goto" statement*
- *Describe the purpose and use of the "try/catch" statement*
- *Demonstrate your ability to use control-flow statements in simple C# programs*

Introduction

Program control-flow statements are an important part of the C# programming language because they allow you to alter the course of program execution while the program is running. The program control-flow statements presented in this chapter fall into two categories: 1) *selection statements* and 2) *iteration statements*.

Selection statements allow you to alter the course of program execution flow based on the evaluation of a conditional expression. There are three types of selection statements: `if`, `if/else`, and `switch`.

Iteration statements provide a mechanism for repeating one or more program statements based on the result of a conditional expression evaluation. There are three types of iteration statements: `for`, `while`, and `do`. There is also a `foreach` statement that is used with arrays and collections. I will therefore postpone coverage of the `foreach` statement until Chapter 8.

As you will soon learn, each type of control-flow statement has a unique personality. After reading this chapter you will be able to select and apply the appropriate control-flow statement for the particular type of processing you require. This will enable you to write increasingly powerful programs.

In addition to selection and iteration statements, I will show you how to use the keywords `break`, `continue`, and `goto`. The proper use of these keywords combined with selection and iteration statements provides a greater level of processing control.

The material you learn in this chapter will fill your C# programming tool bag with lots of powerful tools. You will be pleasantly surprised at what you can program with the aid of program control-flow statements.

Selection Statements

There are three types of C# selection statements: `if`, `if/else`, and `switch`. The use of each of these statements is covered in detail in this section.

If Statement

The `if` statement conditionally executes a block of code based on the evaluation of a conditional expression. Figure 7-1 graphically shows what happens during the processing of an `if` statement.

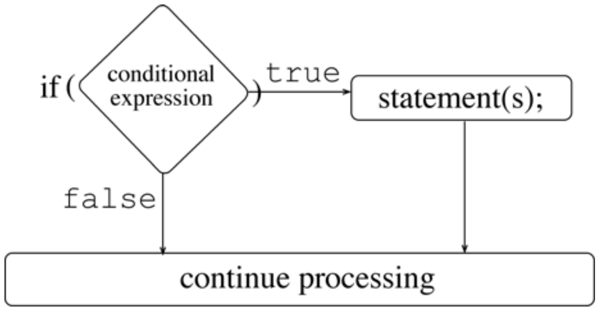


Figure 7-1: if Statement Execution Diagram

A *conditional expression* evaluates to either *true* or *false*. If the conditional expression contained within the parentheses evaluates to true, then the body of the `if` statement executes. If the expression evaluates to false, then the program bypasses the statements contained within the `if` statement body and processing continues with the next statement following the `if` statement.

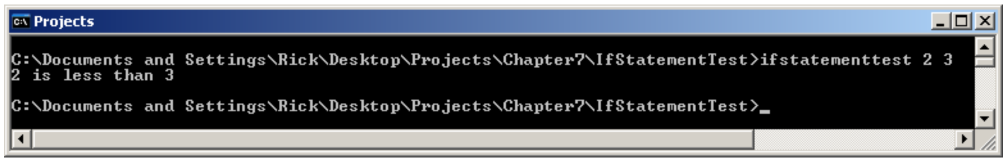
Example 7.1 shows an `if` statement being used in a simple program that reads two integer values from the command line and compares their values.

7.1 IfStatementTest.cs

```
using System;

public class IfStatementTest {
  static void Main(String[] args){
```

```
5         int int_i = Convert.ToInt32(args[0]);
6         int int_j = Convert.ToInt32(args[1]);
7
8         if(int_i < int_j)
9           Console.WriteLine(int_i + " is less than " + int_j);
10      }
11    }
```

Referring to Example 7.1 — the program converts command-line input from strings to integers with the help of the System.Convert.ToInt32() method and assigns the resultant values to the variables int_i and int_j on lines 5 and 6. (Study the Convert class for more conversion methods.) The `if` statement begins on line 8. The less-than operator ‘<’ compares the values of int_i and int_j. If the value of int_i is less than the value of int_j, then the statement on line 9 executes. If not, line 9 is skipped and the program exits. Figure 7-2 shows the results of running this program.

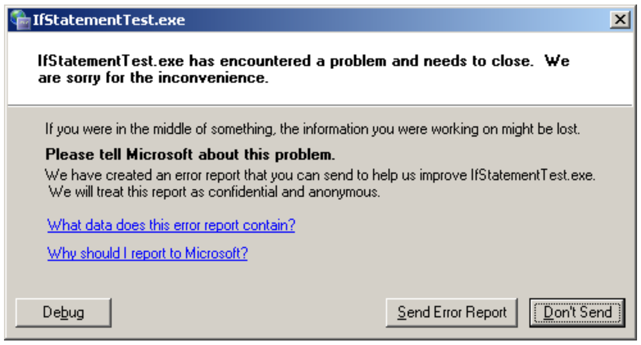


```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementTest>ifstatementtest 2 3
2 is less than 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementTest>_
```

Figure 7-2: Results of Running Example 7.1

Handling Program Error Conditions

If you were unlucky enough to forget to supply two properly-formed integer values on the command-line when you ran the previous program you would have received an error message like the one shown in Figure 7-3.



IfStatementTest.exe

IfStatementTest.exe has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.

We have created an error report that you can send to help us improve IfStatementTest.exe. We will treat this report as confidential and anonymous.

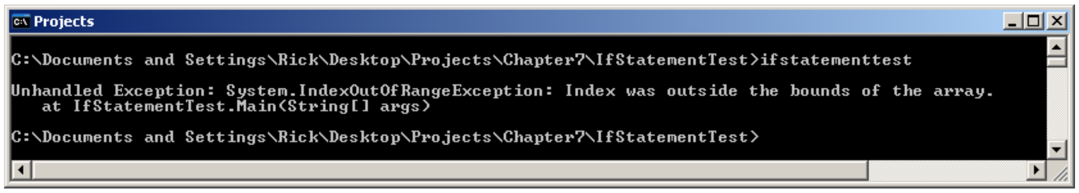
What data does this error report contain?

Why should I report to Microsoft?

Figure 7-3: Typical .NET Error Message Dialog Window

This dialog window signals to you that an error condition or exception of some kind has occurred in your program, and it has stopped running. (It has stopped running rather abruptly I might add!) Since you’re running a program you wrote, click the **Don’t Send** button.

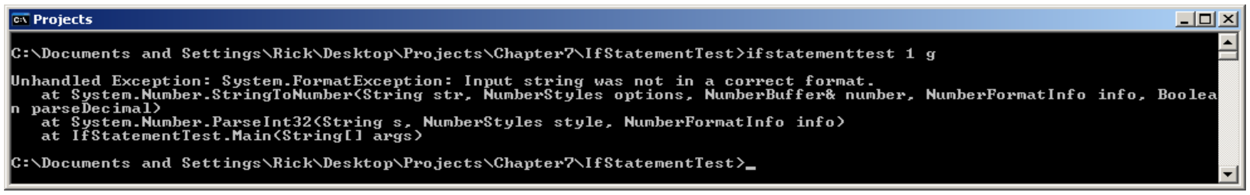
As it turns out, there are several things that can go wrong with this program. Failure to supply the proper number of arguments on the command line will cause an IndexOutOfRangeException. This type of exception occurs if you attempt to access an array element that does not exist. Figure 7-4 shows the exception message output to the console for this type of exception.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementTest>ifstatementtest

Unhandled Exception: System.IndexOutOfRangeException: Index was outside the bounds of the array.
   at IfStatementTest.Main(String[] args)

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementTest>
```

Figure 7-4: Unhandled IndexOutOfRangeException Message

You will receive a different error message if you supply the right number of arguments, but one of the arguments fails to convert properly to an integer value. Figure 7-5 shows the results of running Example 7.1 with one good and one bad input argument.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementTest>ifstatementtest 1 g

Unhandled Exception: System.FormatException: Input string was not in a correct format.
   at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number, NumberFormatInfo info, Boolea
n parseDecimal)
   at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   at IfStatementTest.Main(String[] args)

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementTest>_
```

Figure 7-5: FormatException Error Message

This time the program threw a FormatException. To produce this output, the arguments '1' and 'g' were used as input to the program. The character 'g' failed to convert to an integer and caused the exception condition.

Catching Exceptions With Try/Catch Blocks

Any code that could potentially throw an exception when executed should appear within a `try/catch` statement, also referred to as a *try/catch block*. Doing this places you in control of your program. Your code may still throw an exception, but if it's in a `try/catch` block, you can properly handle the exception and gracefully recover. The user may never know the exception occurred. Look at the code in Example 7.2.

7.2 IfStatementTest.cs (Mod 1)

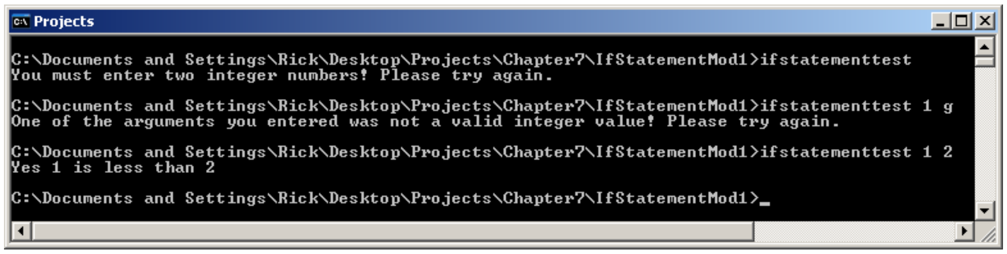
```
1    using System;
2
3    public class IfStatementTest {
4      static void Main(String[] args){
5
6        try{
7          int int_i = Convert.ToInt32(args[0]);
8          int int_j = Convert.ToInt32(args[1]);
9
10         if(int_i < int_j)
11            Console.WriteLine(int_i + " is less than " + int_j);
12
13       }catch(IndexOutOfRangeException){
14         Console.WriteLine("You must enter two integer numbers! Please try again.");
15       }catch(FormatException){
16         Console.Write("One of the arguments you entered was not a valid integer value! ");
17         Console.WriteLine("Please try again.");
18       }
19     }
20   }
```

Referring to Example 7.2 — the body of the original program has been placed inside the body of a `try` block. You can think of the `try` block as a *guarded region*. If everything placed within the `try` block executes, great! If not, the `try` block will exit at the point where the exception is thrown, and the rest of the code within the `try` block will be skipped.

In Example 7.2, all of the code from Example 7.1 inside the body of the Main() method, including the `if` statement, was placed inside the body of the `try` block. Doing this avoids executing the `if` statement if either error condition is encountered since the values of int_i and int_j will not have been properly initialized.

A `try` block can be followed by one or more `catch` blocks. If the program throws an exception, the `catch` block assumes execution based on the type of exception thrown. This is referred to as *handling* the exception. The first `catch` block begins on line 13. It's catching the IndexOutOfRangeException. If this exception occurs, then a message is printed to the screen reminding the user to enter two valid numbers. The second `catch` block handles the FormatException, which will occur if either of the command-line arguments cannot be converted to integers.

Figure 7-6 shows the results of running Example 7.2 with various types of bad input.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod1>ifstatementtest
You must enter two integer numbers! Please try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod1>ifstatementtest 1 g
One of the arguments you entered was not a valid integer value! Please try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod1>ifstatementtest 1 2
Yes 1 is less than 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod1>_
```

Figure 7-6: Results of Running Example 7.2

Referring to Figure 7-6 — notice that now when you forget to supply arguments to the program, you get the gentle reminder. Also, if one or more of the supplied arguments fail to convert to an integer value, you get an appropriate message.

Discovering What Methods Throw What Exceptions

How do you know what type of an exception a .NET Framework API method might throw? You guessed it. You'll have to look it up in the API documentation.

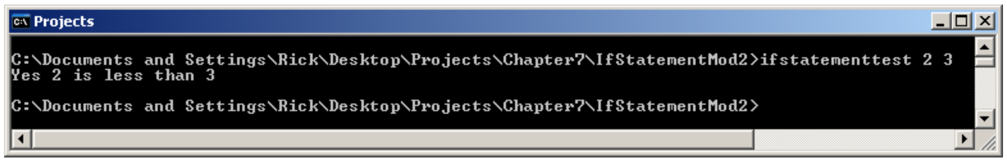
Executing Code Blocks In If Statements

More likely than not, you will want to execute multiple statements in the body of an `if` statement. To do this simply enclose the statements in a set of braces to create a code block. Example 7.3 gives an example of such a code block.

7.3 IfStatementTest.cs (Mod 2)

```
1    using System;
2
3    public class IfStatementTest {
4      static void Main(String[] args){
5
6        try{
7          int int_i = Convert.ToInt32(args[0]);
8          int int_j = Convert.ToInt32(args[1]);
9
10         if(int_i < int_j) {
11            Console.Write("Yes ");
12            Console.WriteLine(int_i + " is less than " + int_j);
13         }
14
15       }catch(IndexOutOfRangeException){
16         Console.WriteLine("You must enter two integer numbers! Please try again.");
17       }catch(FormatException){
18         Console.Write("One of the arguments you entered was not a valid integer value! ");
19         Console.WriteLine("Please try again.");
20       }
21     }
22   }
```

Referring to Example 7.3 — notice that now the statements executed by the `if` statement are contained within a set of braces. The code block begins with the opening brace at the end of line 10 and ends with the closing brace on line 13. Figure 7-7 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod2>ifstatementtest 2 3
Yes 2 is less than 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod2>
```

Figure 7-7: Results of Running Example 7.3

Executing Consecutive If Statements

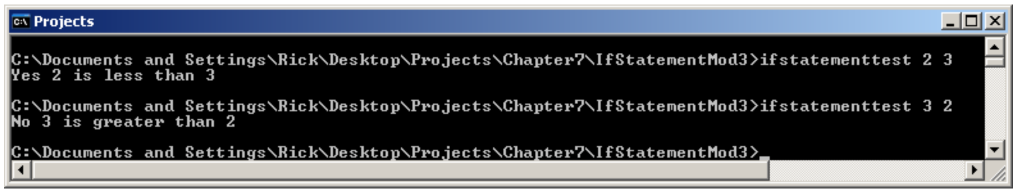
You can follow one `if` statement with another as is shown in Example 7.4.

7.4 IfStatementTest.cs (Mod 3)

```
using System;

public class IfStatementTest {
  static void Main(String[] args){

    try{
   int int_i = Convert.ToInt32(args[0]);
   int int_j = Convert.ToInt32(args[1]);

   if(int_i < int_j) {
      Console.Write("Yes ");
      Console.WriteLine(int_i + " is less than " + int_j);
      }

15       if(int_i > int_j) {
16          Console.Write("No ");
17          Console.WriteLine(int_i + " is greater than " + int_j);
18       }
19
20
21      }catch(IndexOutOfRangeException){
22        Console.WriteLine("You must enter two integer numbers! Please try again.");
23      }catch(FormatException){
24        Console.Write("One of the arguments you entered was not a valid integer value!");
25        Console.WriteLine("Please try again.");
26      }
27     }
28   }
```

Referrring to Example 7.4 — when this program executes, the program evaluates the expressions of both `if` statements. When the program is run with the inputs 2 and 3, the expression of the first `if` statement on line 10 evaluates to true and its body statements execute. The second `if` statement's expression evaluates to false and its body statements are skipped.

The opposite happens when the program is run a second time using input values 3 and 2. This time around, the first `if` statement's expression evaluates to false, its body statements are skipped, and the second `if` statement's expression evaluates to true and its body statements execute. Figure 7-8 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod3>ifstatementtest 2 3
Yes 2 is less than 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod3>ifstatementtest 3 2
No 3 is greater than 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfStatementMod3>_
```

Figure 7-8: Results of Running Example 7.4

If/Else Statement

When you want to provide two possible execution paths for an `if` statement, add the `else` keyword to form an `if/else` statement. Figure 7-9 provides an execution diagram of the `if/else` statement.

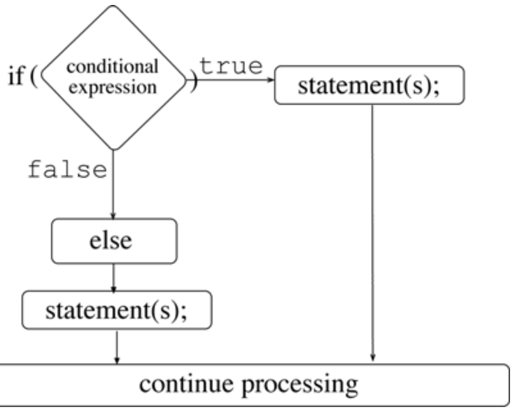


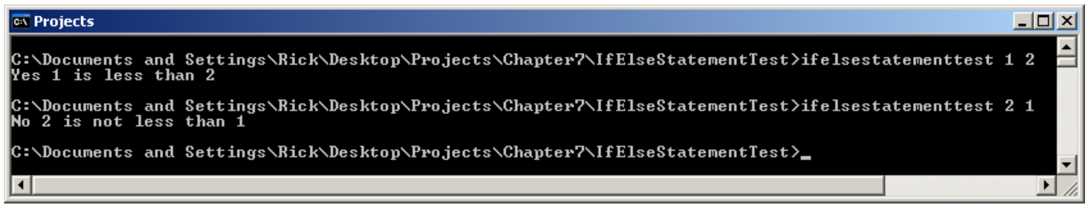
Figure 7-9: if/else Statement Execution Diagram

The `if/else` statement behaves like the `if` statement, except that now when the expression evaluates to false, the statements contained within the body of the `else` clause execute. Example 7.5 provides the same functionality as Example 7.4 using one `if/else` statement.

7.5 IfElseStatementTest.cs

```
1    using System;
2
3    public class IfElseStatementTest {
4      public static void Main(String[] args){
5
6      try{
7       int int_i = Convert.ToInt32(args[0]);
8       int int_j = Convert.ToInt32(args[1]);
9
10      if(int_i < int_j) {
11        Console.Write("Yes ");
12        Console.WriteLine(int_i + " is less than " + int_j);
13        } else {
14           Console.Write("No ");
15           Console.WriteLine(int_i + " is not less than " + int_j);
16           }
17       }catch(IndexOutOfRangeException){
18        Console.WriteLine("You must enter two integer numbers! Please try again.");
19      }catch(FormatException){
20        Console.Write("One of the arguments you entered was not a valid integer value!");
21        Console.WriteLine("Please try again.");
22      }
23     }
24   }
```

Referring to Example 7.5 — the `if` statement begins on line 10. Should the expression evaluate to true, the code block that forms the body of the `if` statement executes. Should the expression evaluate to false, the code block following the `else` keyword executes. Figure 7-10 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfElseStatementTest>ifelsestatementtest 1 2
Yes 1 is less than 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfElseStatementTest>ifelsestatementtest 2 1
No 2 is not less than 1

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\IfElseStatementTest>_
```

Figure 7-10: Results of Running Example 7.5

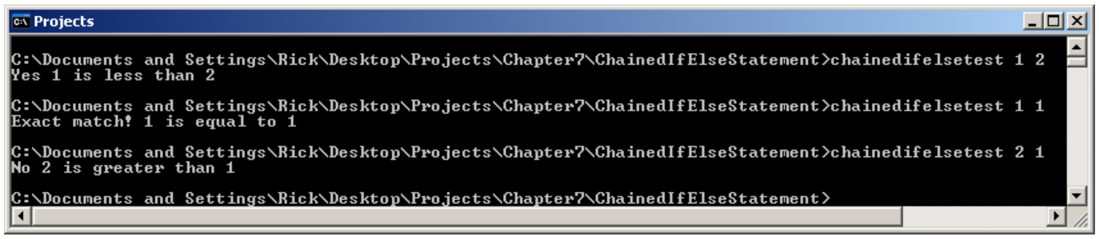
Chained If/Else Statements

You can chain `if/else` statements together to form complex programming logic. To chain one `if/else` statement to another, simply follow the `else` keyword with an `if/else` statement. Example 7.6 illustrates the use of chained `if/else` statements in a program.

7.6 ChainedIfElseTest.cs

```
1    using System;
2
3    public class ChainedIfElseTest {
4      public static void Main(String[] args){
5
6       try{
7         int int_i = Convert.ToInt32(args[0]);
8         int int_j = Convert.ToInt32(args[1]);
9
10        if(int_i < int_j) {
11          Console.Write("Yes ");
12          Console.WriteLine(int_i + " is less than " + int_j);
13         } else if(int_i == int_j) {
14                 Console.Write("Exact match! ");
15                 Console.WriteLine(int_i + " is equal to " + int_j);
16               else{
17                  Console.Write("No ");
18                  Console.WriteLine(int_i + " is greater than " + int_j);
19               }
20       }catch(IndexOutOfRangeException){
21            Console.WriteLine("You must enter two integer numbers! Please try again.");
22       }catch(FormatException){
23            Console.Write("One of the arguments you entered was not a valid integer value!");
24            Console.WriteLine("Please try again.");
25       }
26     }
27   }
```

There are a couple of important points to note regarding Example 7.6. First, notice how the second `if/else` statement begins on line 13 immediately following the `else` keyword of the first `if/else` statement. Second, notice how indenting is used to aid readability. Figure 7-11 gives the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ChainedIfElseStatement>chainedifelsetest 1 2
Yes 1 is less than 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ChainedIfElseStatement>chainedifelsetest 1 1
Exact match! 1 is equal to 1

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ChainedIfElseStatement>chainedifelsetest 2 1
No 2 is greater than 1

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ChainedIfElseStatement>
```

Figure 7-11: Results of Running Example 7.6

Switch Statement

Use the `switch` statement, also referred to as the `switch/case` statement, in situations where you need to provide multiple execution paths based on the evaluation of a particular *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, *string*, or *enum* type value. Figure 7-12 gives the execution diagram for a `switch` statement.

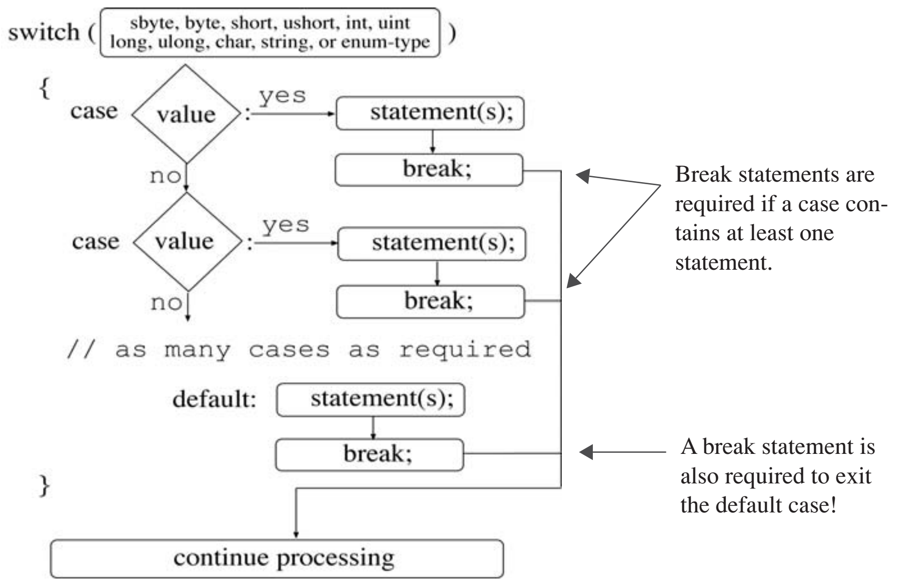

switch (
sbyte, byte, short, ushort, int, uint long, ulong, char, string, or enum-type
)
{
case
value
:
yes
statement(s);
break;
no
case
value
:
yes
statement(s);
break;
no
// as many cases as required
default:
statement(s);
break;
}
continue processing
Break statements are required if a case contains at least one statement.
A break statement is also required to exit the default case!

Figure 7-12: `switch` Statement Execution Diagram

When you write a `switch` statement, you will add one or more `case` clauses to it. When the `switch` statement executes, the program compares the value supplied in the parentheses of the `switch` statement to each `case` value. A match results in the execution of that `case`'s code block.

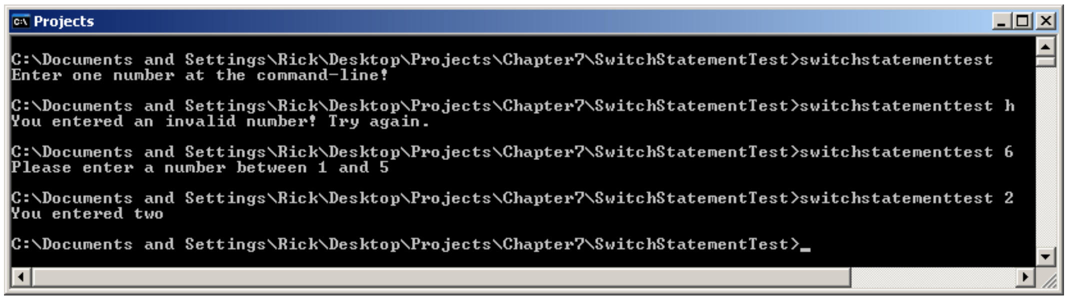
Notice in Figure 7-12 how each `case`'s statements are followed by a `break` statement. The `break` statement exits the `switch` and continues processing. The `break` statement is required because `case` statements are not allowed to implicitly continue processing (*i.e.*, fall through) to the next `case` after executing one or more statements. Example 7.7 gives an example of the `switch` statement in action.

7.7 SwitchStatementTest.cs

```
using System;

public class SwitchStatementTest {
  public static void Main(String[] args){
    try{
       int int_i = Convert.ToInt32(args[0]);

       switch(int_i){
         case 1  : Console.WriteLine("You entered one");
                   break;
         case 2  : Console.WriteLine("You entered two");
                   break;
         case 3  : Console.WriteLine("You entered three");
                   break;
         case 4  : Console.WriteLine("You entered four");
                   break;
```

```
            case 5  : Console.WriteLine("You entered five");
                      break;
            default : Console.WriteLine("Please enter a number between 1 and 5");
                      break;
            }
        }catch(IndexOutOfRangeException){
           Console.WriteLine("Enter one number at the command-line!");
        }catch(FormatException){
           Console.WriteLine("You entered an invalid number! Try again.");
        }
      }
    }
```

Referring to Example 7.7 — this program reads a string from the command line and converts it to an integer value using our friend the Convert.ToInt32() method. The integer variable int_i is then used in the `switch` statement. Its value is compared against the five cases. If there's a match, meaning its value is either 1, 2, 3, 4, or 5, then the related `case` executes and the appropriate message prints to the console. If there's no match, then the `default case` executes and prompts the user to enter a valid value. The `try/catch` statement handles anticipated exceptions. Figure 7-13 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTest>switchstatementtest
Enter one number at the command-line!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTest>switchstatementtest h
You entered an invalid number! Try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTest>switchstatementtest 6
Please enter a number between 1 and 5

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTest>switchstatementtest 2
You entered two

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTest>_
```

Figure 7-13: Results of Running Example 7.7

Implicit Case Fall-Through

You can rewrite the `switch` statement shown in Example 7.7 to take advantage of implicit `case` fall-through. In this version, the `switch` statement relies on the help of an array, as you will see from studying the code shown in Example 7.8.

7.8 SwitchStatementTest.cs (Mod 1)

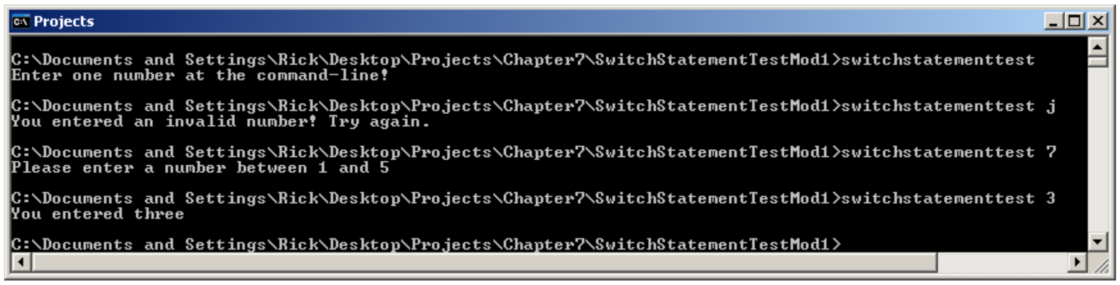
```
1    using System;
2
3    public class SwitchStatementTest {
4      public static void Main(String[] args){
5        try{
6          int int_i = Convert.ToInt32(args[0]);
7          string[] string_array = {"one", "two", "three", "four", "five"};
8
9          switch(int_i){
10           case 1  : // this works because these cases contain no statements...
11           case 2  :
12           case 3  :
13           case 4  :
14           case 5  : Console.WriteLine("You entered " + string_array[int_i-1]);
15                     break;
16           default : Console.WriteLine("Please enter a number between 1 and 5");
17                     break;
18           }
19         }catch(IndexOutOfRangeException){
20            Console.WriteLine("Enter one number at the command-line!");
21         }catch(FormatException){
22            Console.WriteLine("You entered an invalid number! Try again.");
23         }
24     }
25   }
```

Referring to Example 7.8 — although arrays are formally covered in Chapter 8, the use of one in this particular example should not be too confusing to you. A string array, similar to that used as an argument to the Main() method is declared on line 7. It is initialized to hold five string values: "one", "two", "three", "four", and "five". Each element of the array is accessed with an integer index value of between 0 and 4 where 0 represents the first element of the array and 4 represents the last element. The `switch` statement is then rewritten in a more streamlined fashion with the help of implicit `case` fall-through. Implicit `case` fall-through works as long as a `case` contains no statements.

The string_array variable on line 14 uses the variable int_i to provide the text representation of the numbers 1, 2, 3, 4, and 5. The variable int_i serves as the array index. Notice, however, that 1 must be subtracted from int_i to yield the proper array offset and refer to the correct string_array element.

If you are confused by the use of the array in this example don't panic. I cover arrays in detail in Chapter 8.

Figure 7-14 gives the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTestMod1>switchstatementtest
Enter one number at the command-line!

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTestMod1>switchstatementtest j
You entered an invalid number! Try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTestMod1>switchstatementtest 7
Please enter a number between 1 and 5

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTestMod1>switchstatementtest 3
You entered three

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\SwitchStatementTestMod1>
```

Figure 7-14: Results of Running Example 7.8

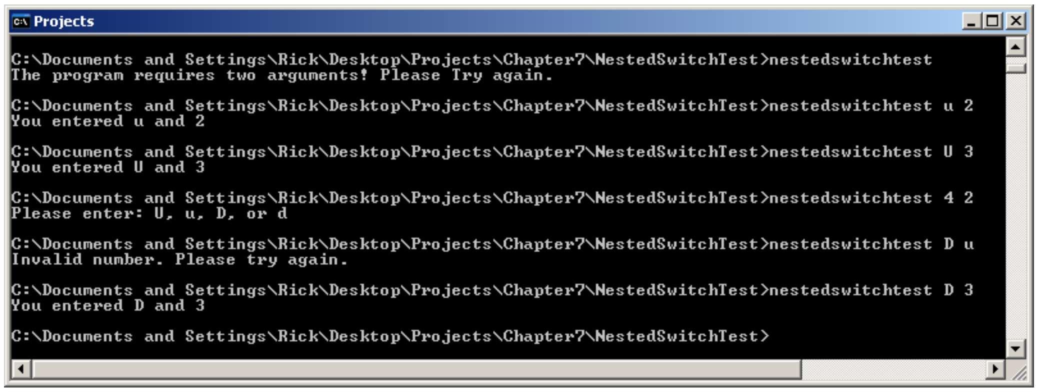
Nested Switch Statement

Switch statements can be nested to yield complex programming logic. Study Example 7.9.

7.9 NestedSwitchTest.cs

```
using System;

public class NestedSwitchTest {
  public static void Main(String[] args){
    try{
     char char_c = (args[0])[0];
     int int_i = Convert.ToInt32(args[1]);

     switch(char_c){
       case 'U' :
       case 'u' : switch(int_i){
                    case 1:
                    case 2:
                    case 3:
                    case 4:
                    case 5: Console.WriteLine("You entered " + char_c + " and " + int_i);
                            break;
                    default: Console.WriteLine("Please enter: 1, 2, 3, 4, or 5");
                             break;
                  }
                  break;
       case 'D' :
       case 'd' : switch(int_i){
                    case 1:
                    case 2:
                    case 3:
                    case 4:
                    case 5: Console.WriteLine("You entered " + char_c + " and " + int_i);
                            break;
                    default: Console.WriteLine("Please enter: 1, 2, 3, 4, or 5");
                             break;
                  }
                  break;
         default: Console.WriteLine("Please enter: U, u, D, or d");
                  break;
       }

     }catch(IndexOutOfRangeException){
         Console.WriteLine("The program requires two arguments! Please try again.");
     }catch(FormatException){
         Console.WriteLine("Invalid number. Please try again.");
     }
    }
  }
```

Referring to Example 7.9 — this program reads two string arguments from the command line. It then takes the first character of the first string and assigns it to the variable char_c. Next, it converts the second string into an integer value. The char_c variable is used in the first, or outer, `switch` statement. As you can see from examining the code, it is looking for the characters 'U', 'u', 'D', or 'd'. The nested `switch` statements are similar to the one used in the previous example. Notice again that indenting is used to help readers of the code distinguish between outer and inner `switch` statements. Figure 7-15 gives the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest
The program requires two arguments! Please Try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest u 2
You entered u and 2

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest U 3
You entered U and 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest 4 2
Please enter: U, u, D, or d

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest D u
Invalid number. Please try again.

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>nestedswitchtest D 3
You entered D and 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedSwitchTest>
```

Figure 7-15: Results of Running Example 7.9

Quick Review

Selection statements provide alternative program execution paths based on the evaluation of a conditional expression. There are three types of selection statements: `if`, `if/else`, and `switch`. The conditional expression of the `if` and `if/else` statements must evaluate to a boolean value of *true* or *false*. Any expression that evaluates to boolean *true* or *false* can be used. You can chain together `if` and `if/else` statements to form complex program logic.

The `switch` statement evaluates an *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, *string*, or *enum* value and executes a matching case and its associated statements. Use the `break` keyword to exit a `switch` statement and prevent case fall through. Always provide a default case. Implicit case fall-through is only allowed if a case contains no statements.

Iteration Statements

Iteration statements provide the capability to execute one or more program statements repeatedly based on the results of an expression evaluation. There are three flavors of iteration statement: `while`, `do/while`, and `for.` I discuss these statements in detail in this section. Iteration statements are also referred to as loops. So, when you hear other programmers talking about a `for` loop, `while` loop, or `do` loop, they are referring to the iteration statements.

While Statement

The `while` statement repeats one or more program statements based on the results of an expression evaluation. Figure 7-16 shows the execution diagram for the `while` statement.

Personality Of The While Statement

The `while` statement has the following personality:

- It evaluates the expression before executing its body statements or code block.
- The expression must eventually evaluate to false or the `while` loop will repeat forever. (Sometimes you want a `while` loop to repeat forever until some action inside it forces it to exit.)

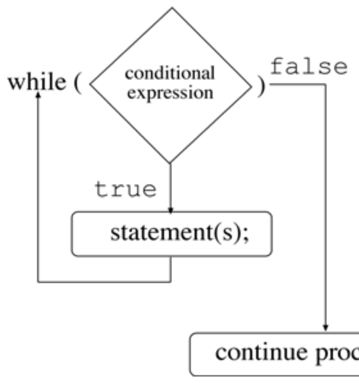


Figure 7-16: while Statement Execution Diagram

• The expression may evaluate to false on the first try and therefore not execute its body statements.

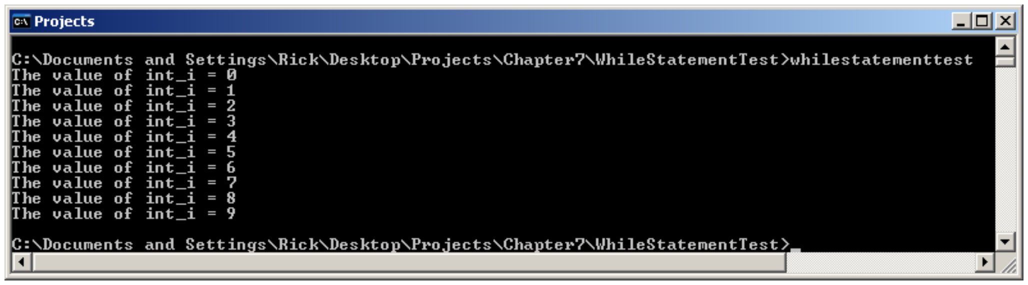
Essentially, the `while` loop performs the expression evaluation first, then executes its body statements. To prevent a `while` loop from looping indefinitely, a program statement somewhere within the body of the `while` loop must either perform an action that will make the expression evaluate to false when the time is right, or explicitly force an exit from the `while` loop.

Example 7.10 shows the `while` statement in action.

7.10 WhileStatementTest.cs

```
1     using System;
2
3     public class WhileStatementTest {
4        static void Main(){
5           int int_i = 0;
6
7           while(int_i < 10){
8             Console.WriteLine("The value of int_i = " + int_i);
9             int_i++;
10          }
11       }
12    }
```

Referring to Example 7.10 — on line 5, the integer variable int_i is declared and initialized to 0. The variable int_i is then compared to the integer literal value 10. So long as int_i is less than 10, the statements contained within the body of the `while` loop execute. (This includes all statements between the opening brace appearing at the end of line 7 and the closing brace on line 10.) Notice how the value of int_i is incremented with the '`++`' operator. This is an essential step. If int_i is not incremented, the expression will always evaluate to true, which would result in an infinite loop. Figure 7-17 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\WhileStatementTest>whilestatementtest
The value of int_i = 0
The value of int_i = 1
The value of int_i = 2
The value of int_i = 3
The value of int_i = 4
The value of int_i = 5
The value of int_i = 6
The value of int_i = 7
The value of int_i = 8
The value of int_i = 9

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\WhileStatementTest>_
```

Figure 7-17: Results of Running Example 7.10

Do/While Statement

The `do/while` statement repeats one or more body statements based on the result of a conditional expression evaluation. The `do/while` loop differs from the `while` loop in that its body statements execute at least once before the expression is evaluated. Figure 7-18 gives the execution diagram for a `do/while` statement.

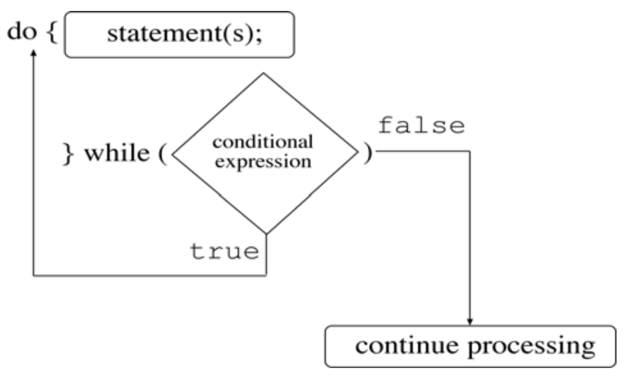


Figure 7-18: do/while Statement Execution Diagram

Personality Of The Do/While Statement

The `do/while` statement has the following personality:

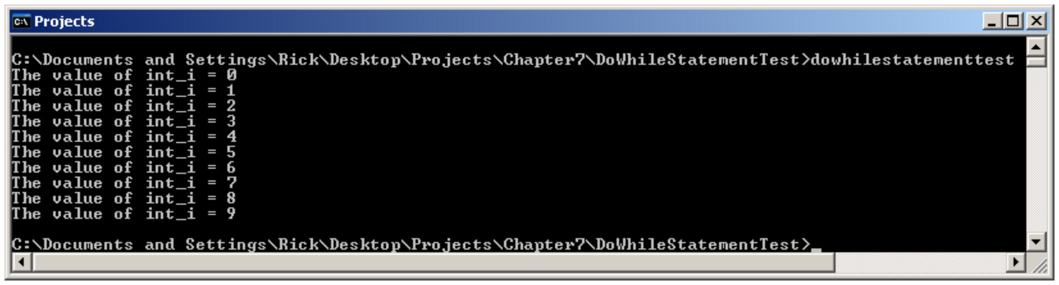
- It executes its body statements once before evaluating its expression.
- A statement within its body must either take some action that will make the expression evaluate to false or explicitly exit the loop, otherwise the `do/while` loop will repeat forever. (And, like the while loop, sometimes you may want it to repeat forever.)
- Use a `do/while` loop if the statements it contains must execute at least once.

So, the primary difference between the `while` and `do/while` statements is when the expression evaluation occurs. The `while` statement evaluates it at the beginning — the `do/while` statement evaluates it at the end. Example 7.11 shows the `do/while` statement in action.

7.11 DoWhileStatementTest.cs

```
1    using System;
2
3    public class DoWhileStatementTest {
4      static void Main(){
5        int int_i = 0;
6
7         do {
8           Console.WriteLine("The value of int_i = " + int_i);
9           int_i++;
10        }while(int_i < 10);
11     }
12   }
```

Referring to Example 7.11 — on line 5, the integer variable int_i is declared and initialized to 0. The `do/while` statement begins on line 7, and its body statements are contained between the opening brace appearing at the end of line 7 and the closing brace on line 10. Notice that the `while` keyword and its expression are terminated with a semicolon. Figure 7-19 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\DoWhileStatementTest>dowhilestatementtest
The value of int_i = 0
The value of int_i = 1
The value of int_i = 2
The value of int_i = 3
The value of int_i = 4
The value of int_i = 5
The value of int_i = 6
The value of int_i = 7
The value of int_i = 8
The value of int_i = 9

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\DoWhileStatementTest>_
```

Figure 7-19: Results of Running Example 7-11

For Statement

The `for` statement repeats a set of program statements, just like the `while` loop and the `do/while` loop. However, the `for` loop provides a more convenient way to combine the actions of counter variable initialization, expression evaluation, and counter variable incrementing. Figure 7.20 gives the execution diagram for the `for` statement.

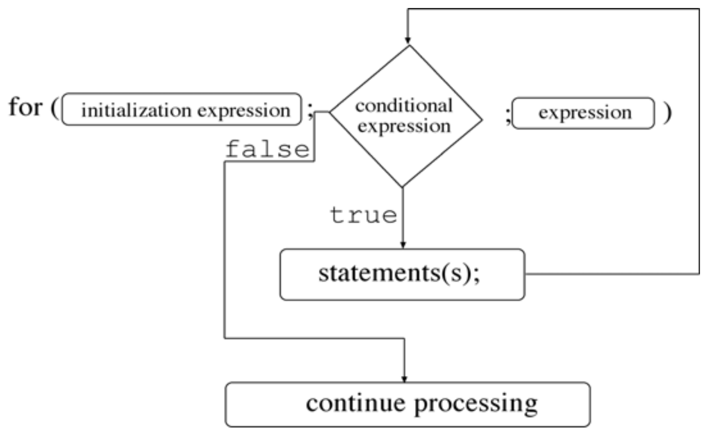


Figure 7-20: for Statement Execution Diagram

How The For Statement Is Related To The While Statement

The `for` statement is more closely related to the `while` statement than to the `do/while` statement. This is because the `for` statement's middle expression (*i.e.*, the one used to decide whether or not to repeat its body statements) is evaluated before its body statements execute.

Personality Of The For Statement

The `for` statement has the following personality:

- It provides a convenient way to initialize counter variables, perform conditional expression evaluation, and increment loop-control variables.
- The conditional expression is evaluated up front before its code block executes, just like the `while` statement
- The `for` statement is the statement of choice to process arrays (You will become an expert at using the `for` statement in Chapter 8).

Example 7.12 shows the `for` statement in action.

7.12 ForStatementTest.cs

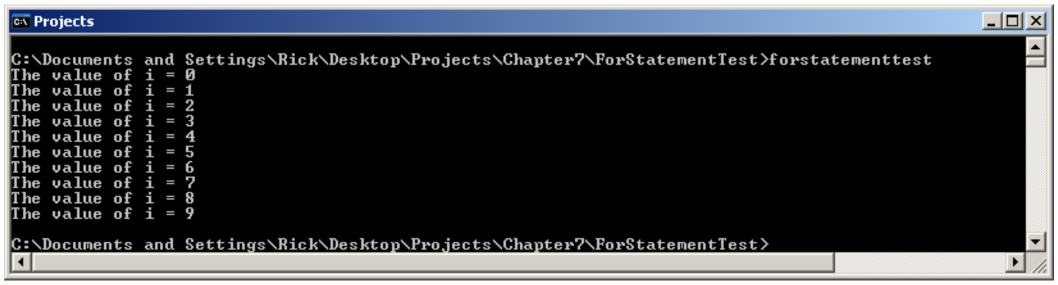
```
1    using System;
2
3    public class ForStatementTest {
4      static void Main(){
5
6          for(int i = 0; i < 10; i++){
7            Console.WriteLine("The value of i = " + i);
8          }
9       }
10   }
```

Referring to Example 7.12 — the `for` statement begins on line 6. Notice how the integer variable i is declared and initialized in the first expression. The second expression compares the value of i to the integer literal value 10. The third expression increments i using the '++' operator.

In this example, I have enclosed the single body statement in a code block. (Remember, a code block is denoted by a set of braces.) However, if a `for` statement only executes one statement, you can omit the braces. (This is true for the `while` and `do` loops as well.) For example, the `for` statement shown in Example 7.12 could be rewritten in the following manner:

```
for(int i = 0; i < 10; i++)
  Console.WriteLine(“The value of i = “ + i );
```

Figure 7-21 shows the results of running example 7.12.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ForStatementTest>forstatementtest
The value of i = 0
The value of i = 1
The value of i = 2
The value of i = 3
The value of i = 4
The value of i = 5
The value of i = 6
The value of i = 7
The value of i = 8
The value of i = 9

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ForStatementTest>
```

Figure 7-21: Results of Running Example 7.12

Nesting Iteration Statements

Iteration statements can be nested to implement complex programming logic. For instance, you can use a nested `for` loop to calculate the following summation:

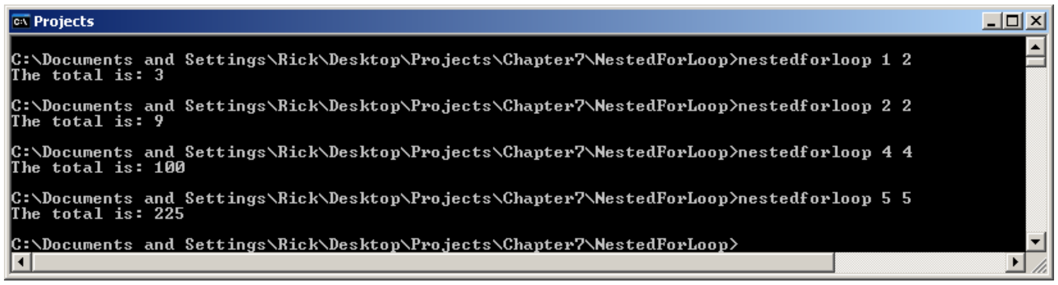
$$\sum_{i=1}^{m}\left(\sum_{j=1}^{n}(i \times j)\right)$$

Example 7.13 offers one possible solution for this particular problem.

7.13 NestedForLoop.cs

```
using System;

public class NestedForLoop {
  static void Main(String[] args){
    try{
      int limit_i = Convert.ToInt32(args[0]);
      int limit_j = Convert.ToInt32(args[1]);
      int total = 0;

      for(int i = 1; i <= limit_i; i++){
        for(int j = 1; j <= limit_j; j++){
          total += (i*j);
        }
      }
      Console.WriteLine("The total is: " + total);
    }catch(IndexOutOfRangeException){
       Console.WriteLine("Two arguments are required to run this program!");
    }catch(FormatException){
       Console.WriteLine("Both arguments must be integers!");
    }
  }
}

```

Referring to Example 7.13 — this program takes two strings as command-line arguments, converts them into integer values, and assigns them to the variables limit_i and limit_j. These variables are then compared against the values of i and j, respectively, in the middle expression of each `for` loop. Notice, too, that indenting makes it easier to distinguish between the outer and inner `for` statements. Figure 7-22 shows the results of running this program using various inputs.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedForLoop>nestedforloop 1 2
The total is: 3

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedForLoop>nestedforloop 2 2
The total is: 9

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedForLoop>nestedforloop 4 4
The total is: 100

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedForLoop>nestedforloop 5 5
The total is: 225

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\NestedForLoop>
```

Figure 7-22: Results of Running Example 7.13

Mixing Selection And Iteration Statements: A Powerful Combination

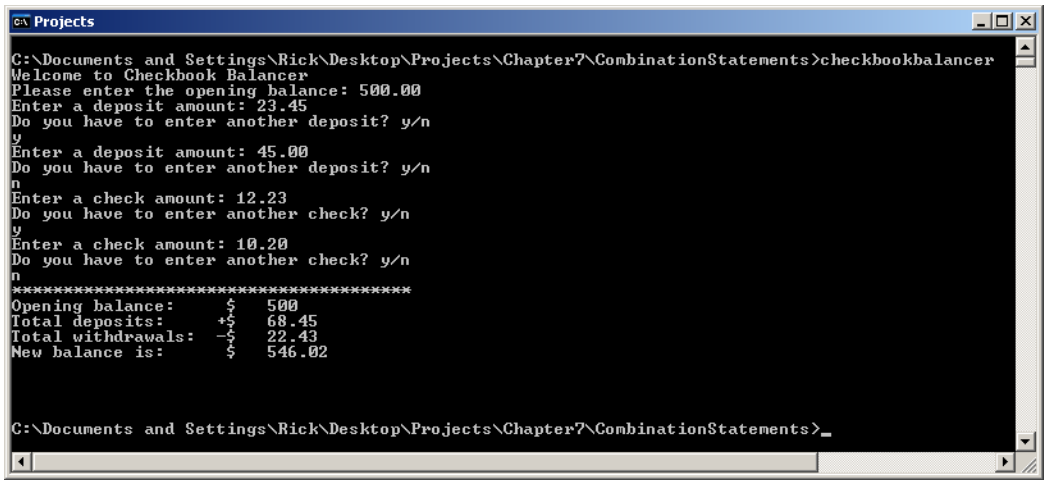
You can combine selection and iteration statements in practically any fashion to solve your particular programming problem. You can place selection statements inside the body of iteration statements or vice versa. Refer to the Robot Rat program developed in Chapter 3 for an example of how to combine control-flow statements to form complex programming logic. Example 7.14 offers another example. The CheckBookBalancer class below implements a simple program that will help you balance your check book. It reads string input from the console and converts it into the appropriate form using the Convert class. It also uses `try/catch` blocks like the previous examples.

7.14 CheckBookBalancer.cs

```
using System;

public class CheckBookBalancer {
    static void Main(){
      /**** Initialize Program Variables ******/

      char keep_going = 'Y';
      double balance = 0.0;
      double deposits = 0.0;
      double withdrawals = 0.0;
      bool good_double = false;

      /**** Display Welcome Message ****/
      Console.WriteLine("Welcome to Checkbook Balancer");


      /**** Get Starting Balance *****/
      do{
       try{
         Console.Write("Please enter the opening balance: ");
         balance = Convert.ToDouble(Console.ReadLine());
         good_double = true;
         }catch(FormatException){ Console.WriteLine("Please enter a valid balance!");}
        }while(!good_double);


      /**** Add All Deposits ****/
      while((keep_going == 'y') || (keep_going == 'Y')){
          good_double = false;
          do{
            try{
               Console.Write("Enter a deposit amount: ");
               deposits += Convert.ToDouble(Console.ReadLine());
               good_double = true;
               }catch(FormatException){ Console.WriteLine("Please enter a valid deposit value!");}
             }while(!good_double);
           Console.WriteLine("Do you have to enter another deposit? y/n");
           try{
             keep_going = (Console.ReadLine())[0];
              }catch(IndexOutOfRangeException){ Console.WriteLine("Problem reading input!");}
      }

     /**** Subtract All Checks Written ****/
      keep_going = 'y';
      while((keep_going == 'y') || (keep_going == 'Y')){
          good_double = false;
          do{
            try{
               Console.Write("Enter a check amount: ");
               withdrawals += Convert.ToDouble(Console.ReadLine());
```

```
                 good_double = true;
                 }catch(FormatException){ Console.WriteLine("Please enter a valid check amount!");}
               }while(!good_double);
             Console.WriteLine("Do you have to enter another check? y/n");
             try{
               keep_going = (Console.ReadLine())[0];
                }catch(IndexOutOfRangeException){ Console.WriteLine("Problem reading input!");}
        }

        /**** Display Final Tally ****/

        Console.WriteLine("****************************************");
        Console.WriteLine("Opening balance:     $    " + balance);
        Console.WriteLine("Total deposits:     +$    " + deposits);
        Console.WriteLine("Total withdrawals:  -$    " + withdrawals);
        balance = balance + (deposits - withdrawals);
        Console.WriteLine("New balance is:      $    " + balance);
        Console.WriteLine("\n\n");
    }
  }
```

Figure 7-23 shows the results of running the CheckBookBalancer program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\CombinationStatements>checkbookbalancer
Welcome to Checkbook Balancer
Please enter the opening balance: 500.00
Enter a deposit amount: 23.45
Do you have to enter another deposit? y/n
y
Enter a deposit amount: 45.00
Do you have to enter another deposit? y/n
n
Enter a check amount: 12.23
Do you have to enter another check? y/n
y
Enter a check amount: 10.20
Do you have to enter another check? y/n
n
****************************************
Opening balance:     $   500
Total deposits:     +$   68.45
Total withdrawals:  -$   22.43
New balance is:      $   546.02



C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\CombinationStatements>_
```

Figure 7-23: Results of Running CheckBookBalancer

Quick Review

Iteration statements repeat blocks of program code based on the result of a conditional expression evaluation. There are three types of iteration statements: `while`, `do/while`, and `for`. The `while` statement evaluates its conditional expression before executing its code block. The `do/while` statement executes its code block first and then evaluates the conditional expression. The `for` statement provides a convenient way to write a `while` statement as it combines loop-counter variable declaration and initialization, conditional expression evaluation, and loop-counter variable incrementing in one statement.

Break, Continue, And Goto

The `break`, `continue`, and `goto` statements belong to a class of C# statements known as *jump* statements. The `break` statement, as you have already learned, is used to exit a `switch` statement. It is also used to exit `for`, `while`, and `do` loops. The `continue` statement stops the processing of the current loop iteration and begins the next iteration. It is used in conjunction with `for`, `while`, and `do` loops. The `goto` statement is used with a label to jump to a specific point in a program.

Break Statement

An `break` statement looks like this:

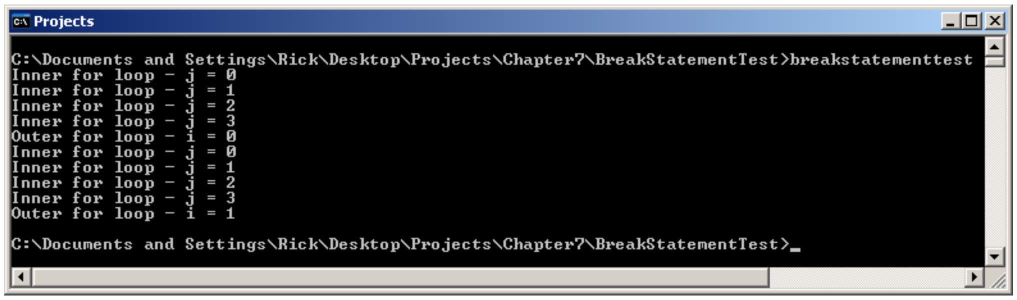
`break;`

Use a `break` statement in the body of `switch`, `for`, `while`, and `do/while` statements to immediately exit its containing statement. When used in a nested statement structure, the `break` exits the innermost containing statement. To exit nested statement structures, you must use the `goto` statement instead. Example 7.15 shows the `break` statement in action.

7.15 BreakStatementTest.cs

```
1    using System;
2
3    public class BreakStatementTest {
4      static void Main(){
5         for(int i = 0; i < 2; i++){
6             for(int j = 0; j < 1000; j++){
7               Console.WriteLine("Inner for loop - j = " + j);
8               if(j == 3) break;
9             }
10           Console.WriteLine("Outer for loop - i = " + i);
11        }
12     }
13   }
```

Referring to example 7.15 — here, a `break` statement is used to exit a nested `for` loop. The inner `for` loop is set to loop 1000 times, however, an `if` statement on line 8 checks the value of j. If j == 3, the `break` statement executes, otherwise the loop is allowed to continue. As soon as the expression j == 3 evaluates to *true*, the inner `for` loop terminates and the outer `for` loop executes the Console.WriteLine() method on line 10, and then begins another iteration. Figure 7-24 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\BreakStatementTest>breakstatementtest
Inner for loop - j = 0
Inner for loop - j = 1
Inner for loop - j = 2
Inner for loop - j = 3
Outer for loop - i = 0
Inner for loop - j = 0
Inner for loop - j = 1
Inner for loop - j = 2
Inner for loop - j = 3
Outer for loop - i = 1

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\BreakStatementTest>_
```

Figure 7-24: Results of Running Example 7.15

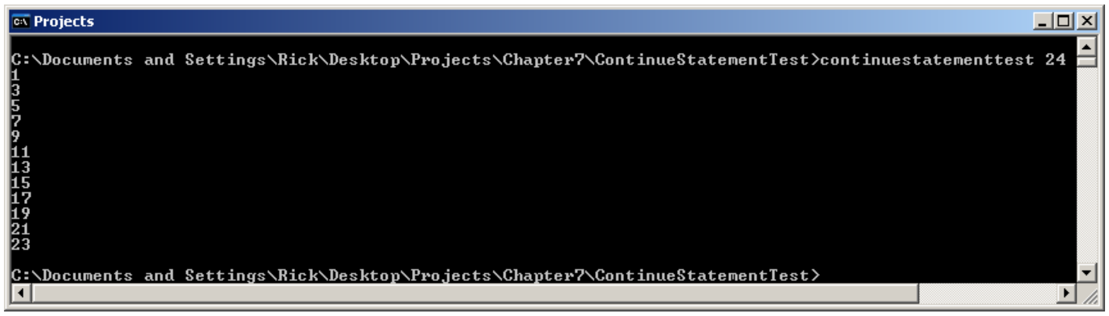
Continue Statement

The `continue` statement stops the current iteration of its containing loop and begins a new iteration. Example 7.16 shows the `continue` statement in action in a short program that prints odd integers.

7.16 ContinueStatementTest.cs

```
using System;

public class ContinueStatementTest {
  static void Main(String[] args){
    try{
      int limit_i = Convert.ToInt32(args[0]);
      for(int i = 0; i<limit_i; i++){
        if((i % 2) == 0) continue;
        Console.WriteLine(i);
      }
    }catch(IndexOutOfRangeException){
       Console.WriteLine("This program requires one integer argument!");
    }catch(FormatException){
       Console.WriteLine("Argument must be a valid integer!");
    }
  }
}
```

Referring to Example 7.16 — a string argument is entered on the command line, converted into an integer value, and assigned to the limit_i variable. The `for` statement uses the limit_i variable to determine how many loops it should perform. The `if` statement on line 8 uses the modulus operator '`%`' to see if the current loop index i is evenly divisible by 2. If so, it's not an odd number and the `continue` statement executes another iteration of the for loop. If the looping index i proves to be odd, then the `continue` statement is bypassed and the remainder of the for loop executes, resulting in the odd number being printed to the console. Figure 7-25 shows the results of running this program with the input 24.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ContinueStatementTest>continuestatementtest 24
1
3
5
7
9
11
13
15
17
19
21
23

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\ContinueStatementTest>
```

Figure 7-25: Results of Running Example 7.16

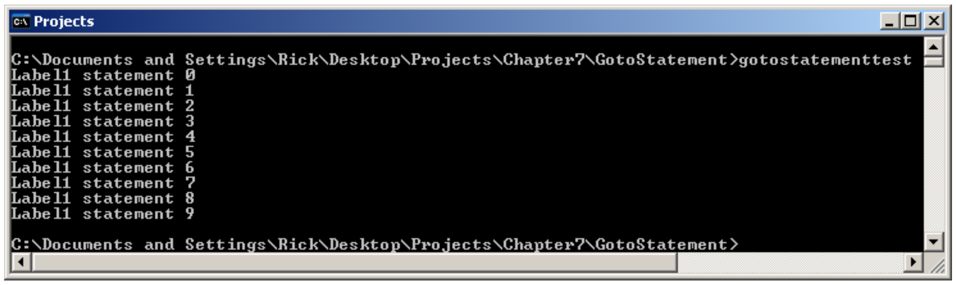
Goto Statement

The `goto` statement is used with a label to jump to the labeled point in the program. Example 7.17 shows the `goto` statement being used to implement a simple repetitive loop.

7.17 GotoStatementTest.cs

```
1     using System;
2
3     public class GotoStatementTest {
4       static void Main(){
5         int i = 0;
6         Label1: Console.WriteLine("Label1 statement " + i++);
7         if(i < 10) goto Label1;
8       }
9     }
```

Referring to Example 7.17 — this program shows how the `goto` statement can be used to implement a `do/while` statement. The statement on line 6 prints first and then increments the variable i. The `if` statement on line 7 compares i to the value 10. If i < 10, then execution jumps to Label1 on line 6. When i reaches 10, the program exits. Figure 7-26 shows the results of running this program.



```
C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\GotoStatement>gotostatementtest
Label1 statement 0
Label1 statement 1
Label1 statement 2
Label1 statement 3
Label1 statement 4
Label1 statement 5
Label1 statement 6
Label1 statement 7
Label1 statement 8
Label1 statement 9

C:\Documents and Settings\Rick\Desktop\Projects\Chapter7\GotoStatement>
```

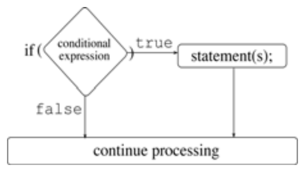
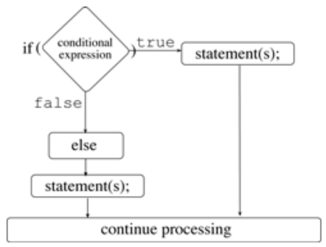
Figure 7-26: Results of Running Example 7.17

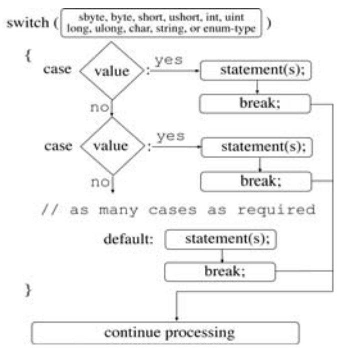
Quick Review

The `break` and `continue` statements provide fine-grained control over iteration statements. In addition to exiting `switch` statements, the `break` statement is used to exit `for`, `while`, and `do/while` loops. The `continue` statement terminates the current iteration of the loop it's embedded within and forces the start of a new iteration. The `goto` statement is used with a label to jump to the labeled spot within a program.

Selection And Iteration Statement Selection Table

The following table provides a quick summary of the C# selection and iteration statements. Feel free to copy it and keep it close by your computer until you've mastered their use.

Statement	Execution Diagram	Operation	When To Use
if	 if (conditional expression) true statement(s); false continue processing	Provides an alternative program execution path based on the results of a conditional expression. If the conditional expression evaluates to true its body statements execute. If it evaluates to false, they are skipped.	Use the `if` statement when you need to execute an alternative set of program statements based on some condition.
if/else	 if (conditional expression) true statement(s); false else statement(s); continue processing	Provides two alternative program execution paths based on the result of a conditional expression. If the conditional expression evaluates to true, the body of the `if` statement executes. If it evaluates to false, the statements associated with the `else` clause execute.	Use the `if/else` when you need to do one thing when the condition is true and another when the condition is false.

switch	 switch (
	sbyte, byte, short, ushort, int, uint long, ulong, char, string, or enum-type
	)
	{
	case
	value
	:
	yes
	statement(s);
	break;
	no
	case
	value
	:
	yes
	statement(s);
	break;
	no
	// as many cases as required
	default:
	statement(s);
	break;
	}
	continue processing
		The `switch` statement evaluates sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or enum values and executes a matching case and its associated statement block. Use the `break` keyword to exit each `case` statement. Always provide a `default` case.	Use the `switch` statement in place of chained `if/else` statements when you are evaluating sbyte, byte, short, ushort, int, uint, long, ulong, char, string, or enum values.

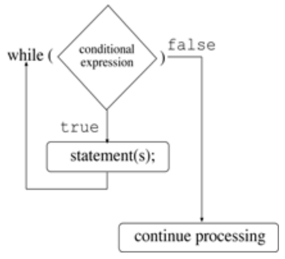
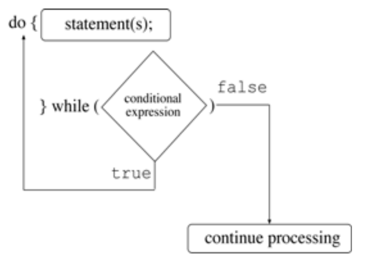
while	 while (conditional expression) false true statement(s); continue processing	The `while` statement repeatedly executes its statement block based on the results of its conditional expression evaluation. The conditional expression will be evaluated first. If true, the statement body executes and the conditional expression will again be evaluated. If it is false, the statement body is skipped and processing continues as normal.	Use the `while` loop when you want to do something over and over again while some condition is true.

Table 7-1: C# Selection And Iteration Statement Selection Guide

Statement	Execution Diagram	Operation	When To Use

do/while	 do {
	statement(s);
	} while (
	conditional expression
	)
	false
	true
	continue processing
		The `do/while` statement operates much like the `while` statement except its body statements are evaluated *at least once* before the conditional expression is evaluated.	Use the `do/while` statement when you want the body statements to be executed at least once.

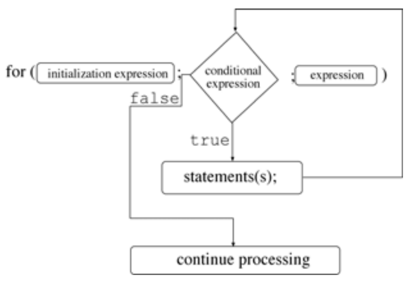
for	 for (initialization expression ; conditional expression ; expression) false true statements(s); continue processing	The `for` statement operates like the `while` statement but offers a more compact way of initializing, comparing, and incrementing counting variables.	Use the `for` statement when you want to iterate over a set of statements for a known number of times.

Table 7-1: C# Selection And Iteration Statement Selection Guide

Summary

Selection statements are used to provide alternative paths of program execution. There are three types of selection statements: `if`, `if/else`, and `switch`. The conditional expression of the `if` and `if/else` statements must evaluate to a boolean value of *true* or *false*. Any expression that evaluates to boolean true or false can be used. You can chain together `if` and `if/else` statements to form complex program logic.

The `switch` statement evaluates a *sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*, *char*, *string*, or *enum* value and executes a matching case and its associated statements. Use the `break` keyword to exit a `switch` statement and prevent case fall-through. Always provide a `default` case. Implicit case fall-through is only allowed if a case contains no statements.

Iteration statements repeat blocks of program code based on the result of a conditional expression evaluation. There are three types of iteration statements: `while`, `do/while`, and `for`. The `while` statement evaluates its conditional expression before executing its code block. The `do/while` statement executes its code block first and then evaluates the conditional expression. The `for` statement provides a convenient way to write a `while` statement as it combines loop-counter variable declaration and initialization, conditional expression evaluation, and loop-counter variable incrementing in a compact format.

The `break` and `continue` keywords provide fine-grained control over iteration statements. In addition to exiting `switch` statements, the `break` statement is used to exit `for`, `while`, and `do/while` loops. The `continue` statement terminates the current iteration statement loop and forces the start of a new iteration. The `goto` statement is used to jump to any labeled spot within a program.

Skill-Building Exercises

1. **If Statement:** Write a program that reads four string values from the console, converts them into int values using the `Console.ToInt32()` method, and assigns the int values to variables named `int_i`, `int_j`, `int_k`, and

`int_l`. Write a series of `if` statements that perform the conditional expressions shown in the first column of the following table and executes the operations described in the second column.

Conditional Expression	Operation
`int_i < int_j`	Print a text message to the console saying that int_i was less than int_j. Use the values of the variables in the message.
`(int_i + int_j) <= int_k`	Print a text message to the console showing the values of all the variables and the results of the addition operation.
`int_k == int_l`	Print a text message to the console saying that int_k was equal to int_l. Use the values of the variables in the text message.
`(int_k != int_i) && (int_j > 25)`	Print a text message to the console that shows the values of the variables.
`(++int_j) && (--int_l)`	Print a text message to the console that shows the values of the variables.

Run the program with different sets of input values to see if you can get all the conditional expressions to evaluate to true.

2. **If/Else Statement:** Write a program that reads two names from the command line. Assign the names to string variables named name_1 and name_2. Use an `if/else` statement to compare the text of name_1 and name_2 to each other. If the text is equal, print a text message to the console showing the names and stating that they are equal. If the text is not equal, print a text message to the console showing the names and stating they are not equal.

Hint: Use the == operator to perform the string value comparison. For example, given two String objects:

```
String name_1 = "Coralie";
String name_2 = "Coralie";
```

You can compare the text of one String object against the text of another String object by using the == operator in the following fashion:

```
name_1 == name_2
```

The == operator returns a boolean value. If the text of both String objects match it will return true, otherwise it will return false.

3. **Switch Statement:** Write a program that reads a string value from the command line and assigns the first character of the string to a character variable named char_val. Use a `switch` statement to check the value of char_val and execute a `case` based on the following table of cases and operations:

Case	Operation
`'A'`	Prompt the user to enter two numbers. Add the two numbers the user enters and print the sum to the console.
`'S'`	Prompt the user to enter two numbers. Subtract the first number from the second number and print the results to the console.
`'M'`	Prompt the user to enter two numbers. Multiply them and print the results to the console.
`'D'`	Prompt the user to enter two numbers. Divide the first number by the second and print the results to the console.
`default`	Prompt the user to enter two numbers, add them together, and print the sum.

Don’t forget to use the `break` keyword to exit each `case`. Study the CheckBookBalancer program given in Example 7.14 to see how to read input from the console using the Console.ReadLine() method.

4. **While Statement:** Write a program that prompts the user to enter a letter. Assign the letter to a character variable named char_c. Use a `while` statement to repeatedly print the following text to the console so long as the user does not enter the letter `‘Q’`:

```
“I love C#!”
```

5. **Do/While Statement:** Write a program that prompts the user to enter a number. Convert the user’s entry into an int and assign the value to an integer variable named int_i. Use a `do/while` loop to add the variable to itself five times. Print the results of each iteration of the `do/while` loop.

6. **For Statement:** Write a program that calculates the following summation using a `for` statement.

$$\sum_{i=1}^{n} i^2$$

7. **Chained If/Else Statements:** Rewrite skill-building exercise 1 using chained `if/else` statements.

8. **Mixed Selection and Iteration Statements:** Rewrite skill-building exercise 3. Use a `while` loop to repeatedly process the `switch` statement. Exit the `while` statement if the user enters the character `‘E’`.

9. **Mixed Selection and Iteration Statements:** Rewrite skill-building exercise 3 again. This time make the `while` loop execute forever using the following format:

```
while(true){

}
```

Add a `case` to the `switch` statement that exits the program when the user enters the character `‘E’`.

10. **Mixed Selection and Iteration Statements:** Rewrite skill-building exercise 6. Repeatedly prompt the user to enter a value for `n`, calculate the summation, and print the results of each step of the summation to the console. Keep prompting the user for numbers and perform the operation until they enter a zero.

Suggested Projects

1. **Weight Guesser Game:** Write a program that guesses the user’s weight. Have them enter their height and age as a starting point. When the program prints a number, it should also ask the user if it is too low, too high, or correct. If correct, the program terminates after writing the number of guesses to the console.

2. **Number Guesser Game:** Write a program that generates a random number between 1 and 100 and then asks the user to guess the number. The program must tell the user if their guess is too high or too low. Keep track of the number of user guesses and print the statistics when the user guesses the correct answer. Prompt the user to repeat the game and terminate when the user enters `‘N’`.

3. **Simple Calculator:** Write a program that implements a four function calculator that adds, subtracts, multiplies, and divides integer and floating point values.

4. **Calculate Area of Circles:** Write a program that calculates the area of circles using the following formula:

$$A_c = \pi r^2$$

Prompt the user for the value of r. After each iteration ask users if they wish to continue and terminate the program if they enter `'N'`.

5. **Temperature Converter:** Write a program that converts the temperature from Celsius to Fahrenheit and vice versa. Use the following formulas to perform your conversions:

$$T_f = \left(\left(\frac{9}{5}\right) \times T_c\right) + 32 \qquad T_c = \left(\frac{5}{9}\right) \times (T_f - 32)$$

6. **Continuous Adding Machine:** Write a program that repeatedly adds numbers entered by the user. Display the running total after each iteration. Exit the program when the user enters -9999.

7. **Create Multiplication Tables:** Write a program that prints the multiplication tables for the numbers 1 through 12 up to the 12th factor.

8. **Calculate Hypotenuse:** Write a program that calculates the hypotenuse of triangles using the Pythagorean theorem:

$$a^2 + b^2 = c^2$$

9. **Calculate Grade Averages:** Write a program that helps the instructor calculate test grade averages. Prompt the user for the number of students and then prompt for each test grade. Calculate the grade average and print the results.

Self-Test Questions

1. To what type must the conditional expression of a selection or iteration statement evaluate?

2. What's the purpose of a selection statement?

3. What's the purpose of an iteration statement?

4. What types can be used as `switch` statement evaluation values?

5. What's the primary difference between a `while` statement and a `do/while` statement?

6. Explain why, in some programming situations, you would choose to use a `do/while` statement vs. a `while` statement.

7. When would you use a `switch` statement vs. chained `if/else` statements?

8. For what purpose is the `break` keyword used in `switch` statements?

9. What's the effect of using the `break` keyword in an iteration statement?

10. What's the effect of using the `continue` keyword in an iteration statement?

11. What's the purpose of the `goto` statement?

References

Lawrence S. Leff. *Geometry The Easy Way*, Second Edition. Barron's Educational Series, Inc. ISBN: 0-8120-4287-5

ECMA-335 Common Language Infrastructure (CLI), 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-335.htm]

ECMA-334 C# Language Specification, 4th Edition, June 2006 [http://www.ecma-international.org/publications/standards/Ecma-334.htm]

Microsoft Developer Network (MSDN) [http://www.msdn.com]

Notes