Branching

Objectives

- Differentiate between conditional and unconditional branching.
- Understand how method calls interrupt the execution path unconditionally, and how methods return values.
- Understand how the if statement allows conditional branching.
- See how the else statement and nested branching add flexibility and how the switch statement adds clarity.
- Discover how looping works in C# and see how the various looping constructs facilitate writing code that is easy to understand.

Unconditional Branching

The "normal" course of events is for a program to execute in the order it was written. Each statement is executed, one after another. In tiny sample applications, that may be the entire story: the program begins, each statement is executed, and then the program ends, but if that were all there was, programs would be very limited and nearly useless.

In any real program, the logic of the program branches. That branching can happen unconditionally, when a method is called, or conditionally, based on the state of some object in the program.

Method Calls

There are a number of ways a program can branch conditionally, and many of these will be discussed later in this course. The most common unconditional branch is a *method*.

The normal course of events is for a program to execute in the order it is written. A method begins and each statement is executed in turn until the method ends.

NOTE The terms method and function can be used interchangeably. C programmers tend to use the term *function*, while C++ and Java programmers tend to use the term *method*.

When a method is called, or invoked, the processing of the program branches to the start of the new method. When the method returns (finishes), processing resumes on the line immediately following the invocation of the method, back in the original calling method.

Figure 1 illustrates that when the method returns, processing resumes on the line after the method call. Processing begins with the execution of Statement1 and then continues to the next statement, which is a method call for Function2. At this point, execution branches to Function2.

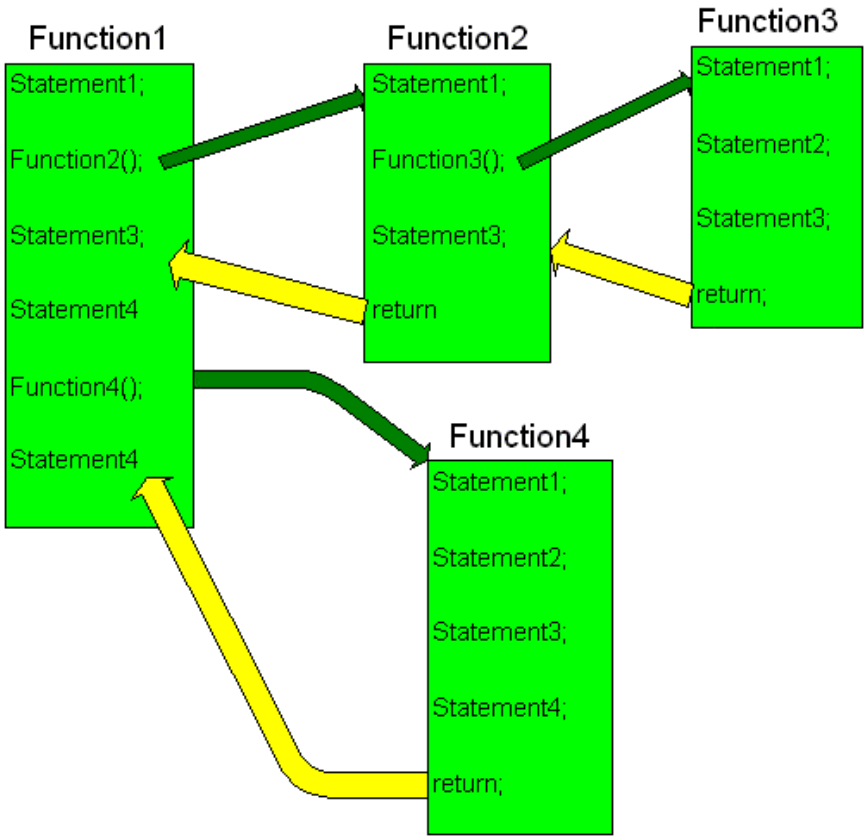

Figure 1. A branching diagram.

Within Function2 the first statement executes and then processing goes to the second statement, which again is a method call to Function3. In Function3 the first statement, Statement1 executes, followed by Statement2 and Statement3, in that order. When the function ends or when it hits a return statement, execution returns to the calling method, in this case Function2. Within Function2 execution resumes on the line immediately following the call to Function3; in this case Statement3.

When Function2 ends, again execution resumes in the calling method, in this case Function1. Statements 3 and 4 are executed and then processing branches on the call to Function4.

Each statement in Function4 executes in order. Statement1 is followed by Statement2, which in turn is followed by Statement3 and then Statement4. When Function4 completes, execution returns to Function1 at Statement4.

Try It Out!

You can see the order of execution illustrated with a simple example.

1. Open a new project in Visual Studio .NET.

2. Under Project Types choose **Visual C# Projects** and within Templates choose **Console Application**. Name the project **UnconditionalBranching** as shown in Figure 2.

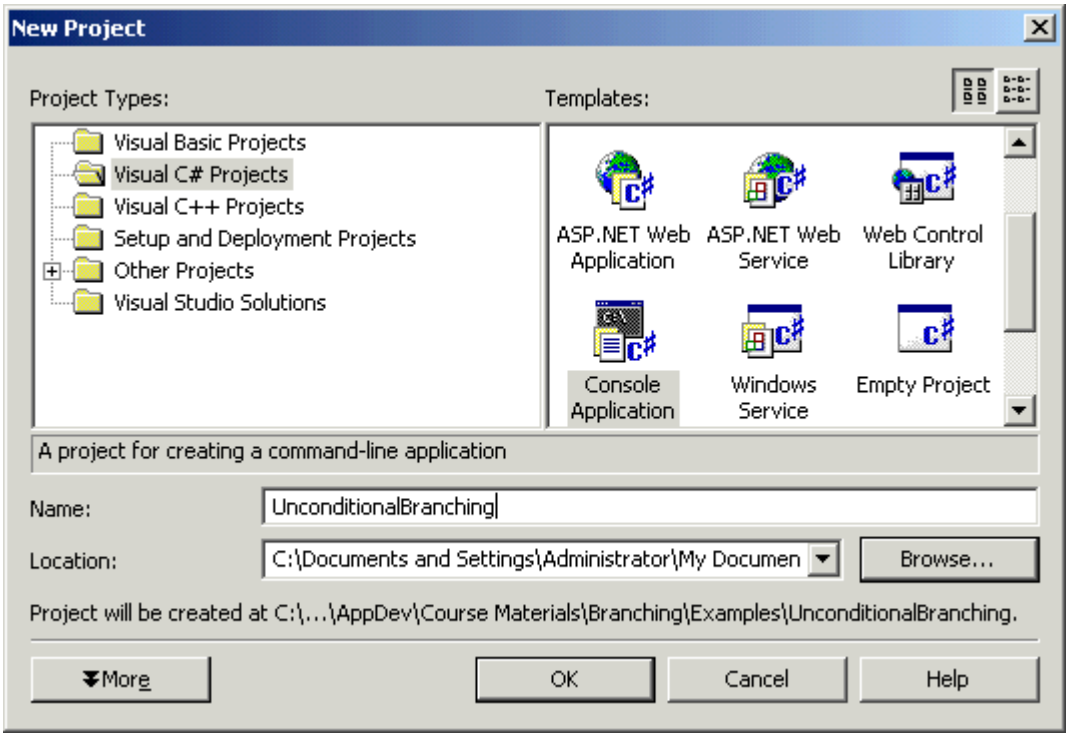

Figure 2. Creating the new project.

3. Replace the comment in Main with the following two lines of code:

```
Console.WriteLine("I'm in Main!");
FunctionOne();
```

4. Add the following methods to the program, just below Main, but within the braces for Class1:

```
static void FunctionOne()
{
   Console.WriteLine("I'm in functionOne!");
   FunctionTwo();
}

static void FunctionTwo()
{
   Console.WriteLine("I'm in functionTwo!");
}
```

5. Compile the program with **SHIFT+CTRL+B**. You should not receive any errors.

6. Execute the program with **CTRL+F5**. Each method is invoked in turn, as shown in Figure 3.

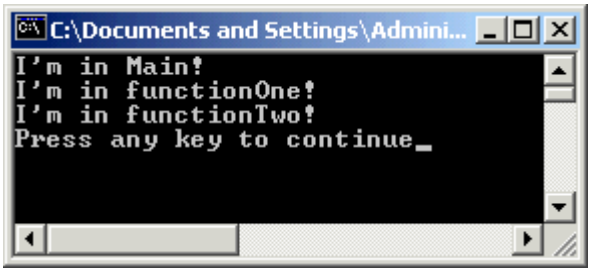

Figure 3. Running the test branching program.

7. You can see the actual steps of invoking methods by putting a breakpoint next to the first executable line in Main(). You set the breakpoint by clicking in the margin next to the line you want to break on. A red dot appears, and the line is highlighted as shown in Figure 4.

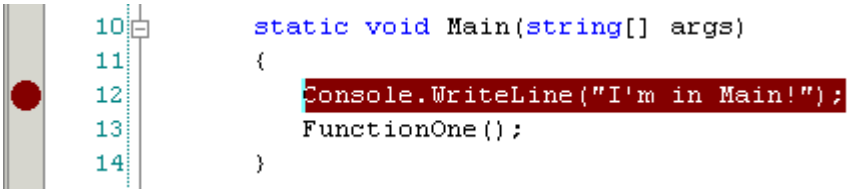

Figure 4. Setting the breakpoint.

8. Execute to the breakpoint by pressing **F5**.

TIP:	**F5** runs the program in the debugger, while **CTRL+F5** runs without the debugger.

9. Program executes to the breakpoint and then stops. Depending on how your environment is set up, you should now see a set of windows similar to those shown in Figure 5. In the main code window you see the current line (about to be executed) indicated with a yellow arrow.

At the bottom of the screen may be a pair of useful windows. The Locals window shows the value of local variables. You'll learn more about this window later in the course.

The lower right-hand corner has the Call Stack window, which shows the list of methods that led to the current method (more about this shortly).

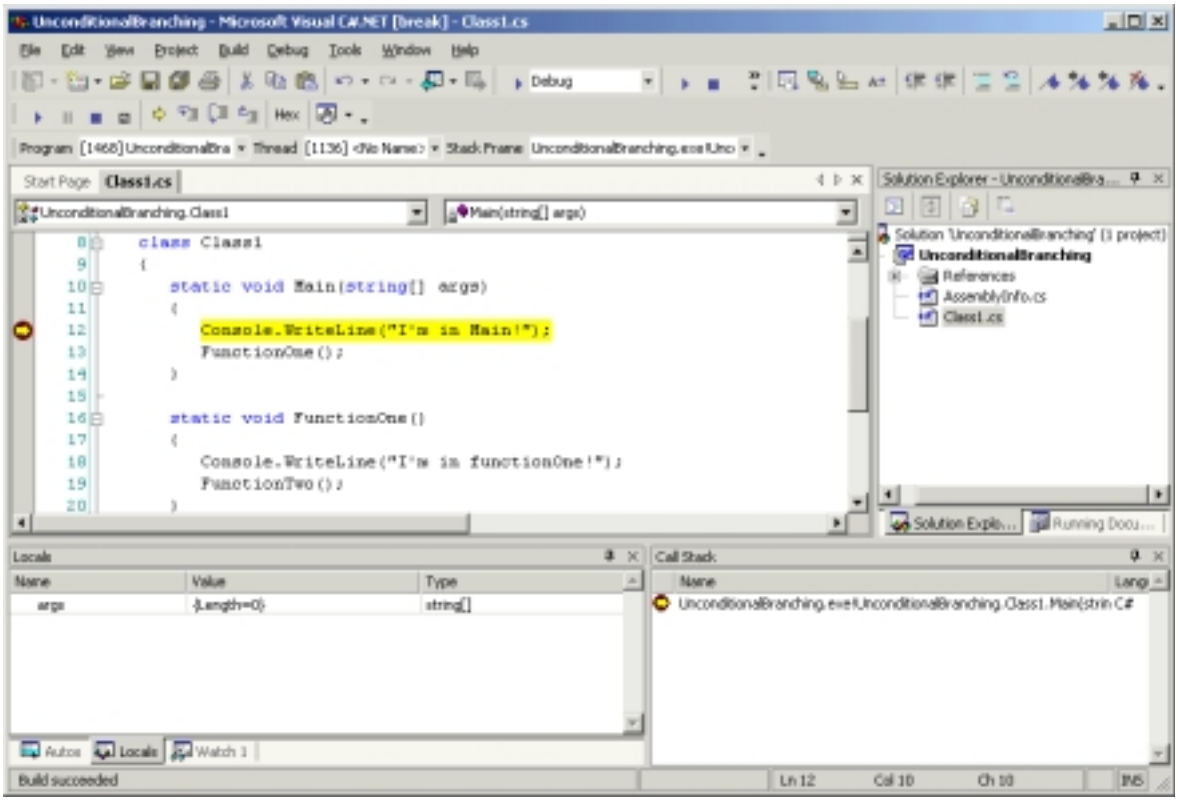

Figure 5. Running to the breakpoint.

10. You can step through this program using **F11**. Pressing once executes the first statement, which writes to the output console. If you switch to the output console (click on it on the taskbar, or use **ALT+TAB** to switch among your applications) you'll see that the line has been displayed to the console.

11. Step into the call to FunctionOne by pressing **F11**. The yellow current statement pointer jumps down into FunctionOne and the CallStack now shows two methods listed, FunctionOne() and Main(), as shown in Figure 6. This indicates that you are currently in FunctionOne, called by Main.

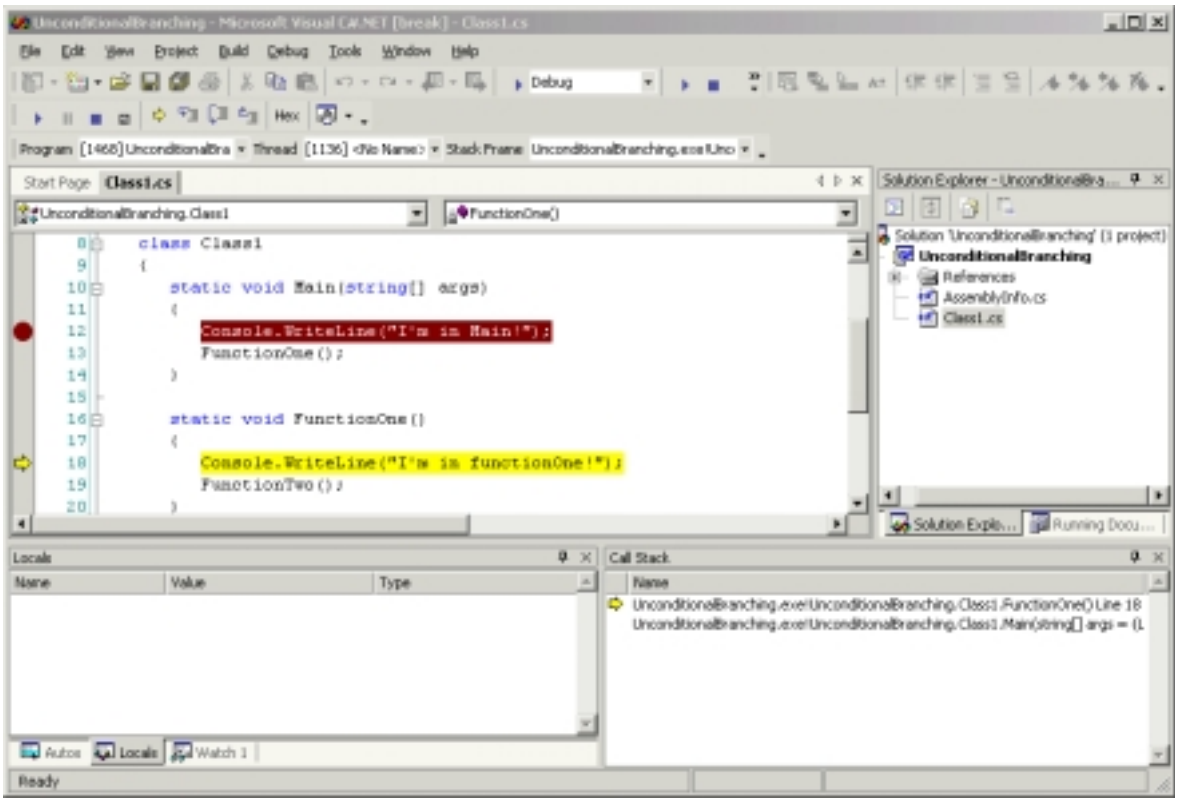

Figure 6. Step into FunctionOne.

12. You can continue stepping through the program, executing each line in turn, and branching unconditionally on the method calls.

Returning Values from Method Calls

Methods can return a value to the calling method. To do this, you must declare the type of value the method will return. Void indicates that the method does not return any value at all. So far, all the methods you've seen have returned void.

Try It Out!

Follow these steps to create a method that returns an integer value, and you'll use that value in the calling method.

1. Create a new console application and call it ReturnValues.

2. Replace the code in Main with the following:

```
Console.WriteLine("I'm in Main!");
FunctionOne();
```

3. Add two methods. FunctionOne returns void, but Doubler returns an int value:

```
static void FunctionOne()
{
   Console.WriteLine("I'm in functionOne!");
   int x = 5;
   int y;
   y = Doubler(x);
   Console.WriteLine("x was {0} and y is {1}", x,y);
   return;
}

static int Doubler(int originalValue)
{
   int newValue =  originalValue * 2;
   return newValue;
}
```

4. Put a breakpoint in Main() where you call FunctionOne and run to that breakpoint, as shown in Figure 7.

5. Step into FunctionOne. You declare a local variable x and initialize its value to 5. You then declare a variable y and assign to y the result of calling Doubler, passing in x. Step until that line is current.

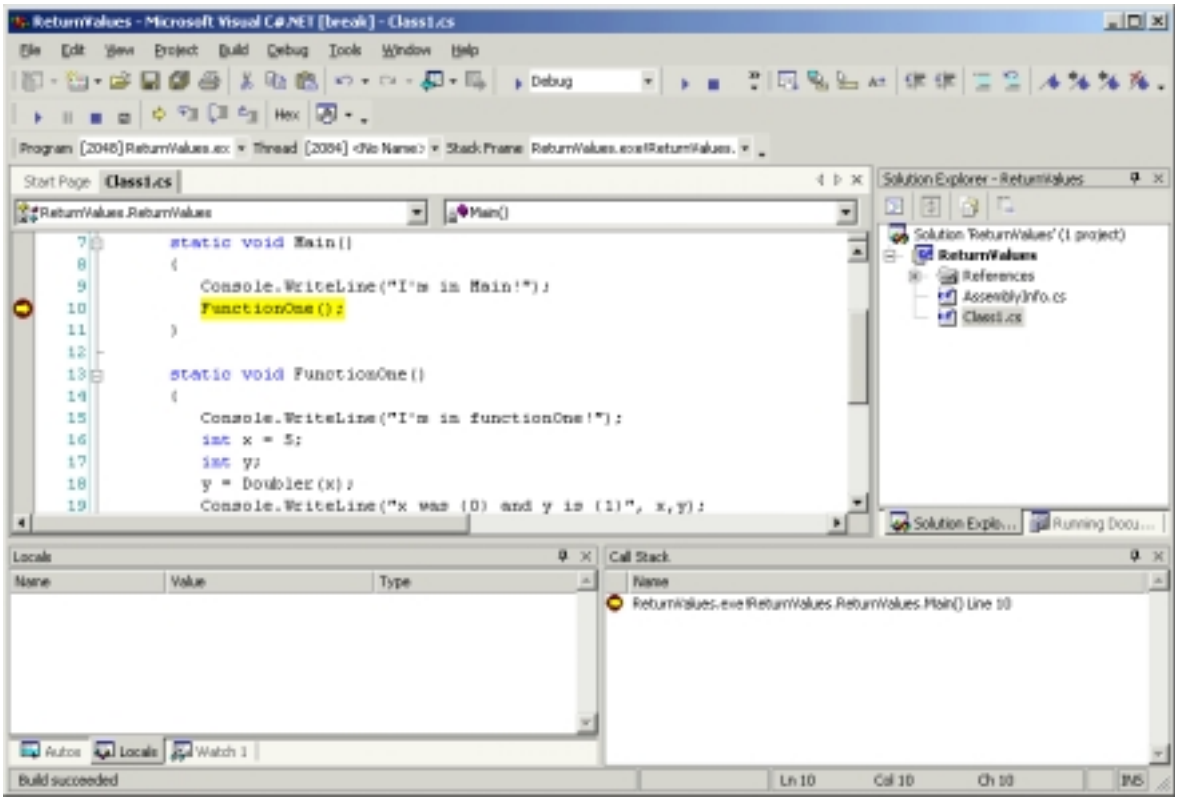

Figure 7. In ReturnValues ready to call FunctionOne.

6. Hover the cursor over the x. You'll see that the value of x is displayed in a small window, and is also shown in the Locals window, as shown in Figure 8.

7. Step into Doubler, and examine the value of the parameter, originalValue. You can see in the Locals window that the value is 5, as you'd expect. This is the value of x, the argument to Doubler as called from FunctionOne.

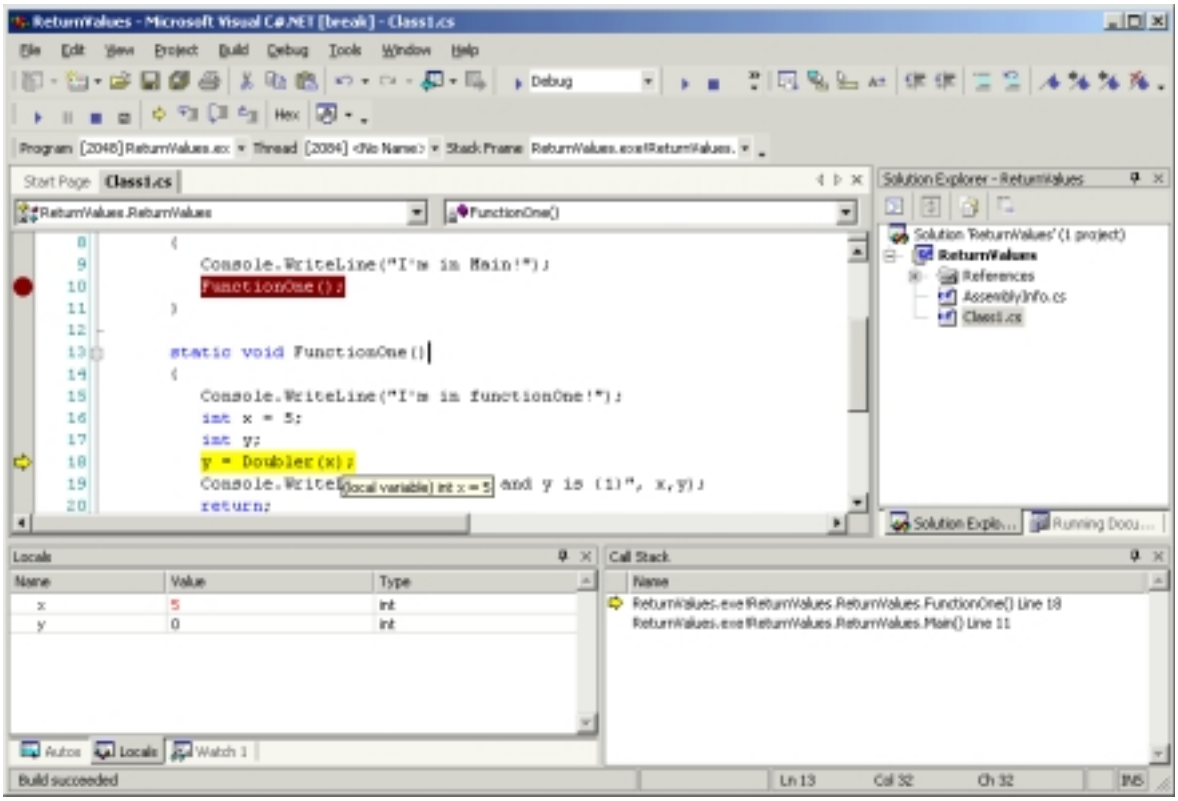

Figure 8. Examining the value of x before invoking Doubler.

8. Within Doubler a new value is created and initialized to twice the parameter. It is this value that is returned in the return statement.

9. Examine the value of *newValue* before the method returns. You'll see in the Locals window that it is 10.

10. Continue stepping out of Doubler and back into FunctionOne. Step through the assignment to y and then examine y; it is now set to 10.

The return value of Doubler was assigned to the local variable y. This is a common idiom with method calls.

Conditional Branching

Far more interesting than unconditional branching, is conditional branching. With conditional branching, your program learns to make decisions based on the conditions of the moment.

In real life you decide whether or not to open your umbrella based on local conditions. For example, if it is raining, you open the umbrella. If it is not raining, you do not (yes, there are always exceptions, but this is the general rule with umbrellas).

In C# you might express this idea with the following code:

```
if (isRaining)
   OpenUmbrella();
```

The if Statement

The if statement tests a Boolean value or an expression that evaluates to a Boolean. Often that expression is a method call that returns a Boolean value. If the value evaluates true, the body of the if statement executes, if not, the statement is skipped.

The body of an if statement can be a single statement, or as noted earlier, can be a block of statements enclosed within braces.

else

If statements can also have an *else* clause. If the expression evaluates true, then the body of the if statement is executed and the body of the else statement is skipped. If the expression evaluates false, then the body of the if statement is skipped and the body of the else statement is executed.

```
if ( myAge > 40 )
   Console.WriteLine("You are over 40");
else
   Console.WriteLine("Youth abounds!");
```

In this example, if the variable myAge is 50 then the Boolean expression "myAge > 40" will evaluate true, and the statement *You are over 40* will be displayed. If the value of myAge is 40 or less, however, then the if statement

will fail, and the else statement will execute, displaying *Youth abounds!* on the console.

A Nested if

You can nest an if statement within another if statement to test multiple conditions. For example, you can test the ambient temperature to see if water is solid, liquid, or gaseous with the following code:

```
if ( temperature > freezingPoint )
{
   if ( temperature < boilingPoint )
      Console.WriteLine("Water is liquid");
   else
      Console.WriteLine("Water is gaseous");
}
else
   ConsoleWriteLine("Water is solid");
```

The outer if statement tests whether the current temperature is greater than the freezingPoint. If it is not, the else statement executes and the string, *Water is solid* is displayed.

If the outer if statement evaluates true (that is, if the current temperature is greater than the freezing point), then the block of statement executes. Within that block, you see an inner, nested if statement. This inner if tests whether the current temperature is less than the boiling point, and if so it displays the message *Water is liquid*. Otherwise, if the temperature is greater than the boiling point, the string *Water is gaseous* is displayed.

Nested if statements work well for situations where one condition depends on another. For this particular example, however, you can write code that is easier to understand by using another construct: the *switch* statement.

Switch

The switch statement allows you to choose among a series of values, taking action on whichever value matches. Switch statements are typically used when you branch, or take other actions, based on one of a series of values.

The formal syntax for a switch statement is as follows:

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

*See **Switch Statements.sln***

The easiest way to see how a switch statement is used is with an example. In the following example, you will switch on the value entered by a user.

```
using System;

namespace SwitchStatements
{
   class SwitchStatements
   {
      static void Main()
      {
         // print the menu of choices
         Console.Write(
            "(1) Walk (2) Run  (3) Crawl  (4) Falling
[1,2,3,4]: ");

         // read the user's choice from the keyboard
         string choice = Console.ReadLine();

         // convert the choice to an integer
         int menuChoice = Convert.ToInt32(choice);

         // switch on the choice and choose
         // the appropriate action
         switch (menuChoice)
         {
            case 1:
               Console.WriteLine("Walking...");
               break;
            case 2:
```

```
                Console.WriteLine("Running...");
                goto case 4;
            case 3:
                Console.WriteLine("Crawling...");
                break;
            case 4:
                Console.WriteLine("Falling...");
                break;
        }               // end switch
    }                   // end Main
}                       // end class SwitchStatements
}                       // end namespace SwitchStatements
```

The example begins by printing a menu to the console:

```
Console.Write(
"(1) Walk (2) Run  (3) Crawl  (4) Falling [1,2,3,4]: ")
```

The user chooses a value of 1, 2, 3, or 4. You read the user's choice with the ReadLine() method of Console, which reads a single line (ended by pressing enter) from the console.

You then assign that string value to an integer by calling the ToInt32 method of the Convert object:

```
string choice = Console.ReadLine();
int menuChoice = Convert.ToInt32(choice);
```

NOTE ToInt32 is a static method of the Convert object, as explained later in this course. For now, you can just treat this as a magic ability to convert strings to integers.

You are ready to switch on the menuChoice value, taking the appropriate action depending on the user's selection. If the user entered 1 to indicate that he wants to walk, you will match the case where the value is 1. In that case you will take whatever action is appropriate. To keep this example simple, you'll print a message to the console:

```
switch (menuChoice)
|{
   case 1:
      Console.WriteLine("Walking...");
      break;
```

The keyword *break* indicates that the handling of the case is completed and processing can fall out of the switch statement.

If the user entered 2 to indicate that they want to run, then the first case would be skipped and the second case would match:

```
case 2:
   Console.WriteLine("Running...");
   goto case 4;
```

This case also prints a message to the console. This time, however, rather than ending with *break* this statement ends with *goto*. Goto indicates that you want to continue as if another case had matched, in this case 4. The result of choosing 2 (run) is that both case 2 and case 4 will execute (in that order). Thus, the output will be *Running… Falling…* which is not inappropriate.

Default

The switch statement may have a default case that will execute if none of the other choices do. You can add a default case to the code shown just below the fourth case:

```
switch (menuChoice)
{
   case 1:
      Console.WriteLine("Walking...");
      break;
   case 2:
      Console.WriteLine("Running...");
      goto case 4;
   case 3:
      Console.WriteLine("Crawling...");
      break;
   case 4:
      Console.WriteLine("Falling...");
      break;
   default:
      Console.WriteLine(
         "I'm sorry, that is not a valid choice");
      break;
}              // end switch
```

If the user enters an invalid value (e.g., 6), the default case executes and the string *I'm sorry, that is not a valid choice* displays.

Looping

One of the principal mechanisms for writing useful programs is the ability to loop through a set of instructions, repeating each instruction until a condition is met.

C# supports a number of looping instructions, including:

- goto
- while
- do…while
- for
- foreach

Each of these constructs, except the foreach loop is illustrated in the following example. You'll learn more about the foreach loop later in this course when collections are covered.

```
using System;

namespace Looping
{
   class Looping
   {
      static void Main()
      {

         // goto
         Console.WriteLine("goto...");
         int i = 0;
         repeat:
         Console.Write("{0} ", i);
         i++;
         if (i < 10)
            goto repeat;

         // while
         Console.WriteLine("\n\nWhile...");
```

```
            i = 0;
            while (i < 10)
            {
               Console.Write("{0} ", i);
               i++;
            }

            // do...while
            Console.WriteLine("\n\nDo While...");
            i = 0;
            do
            {
               Console.Write("{0} ", i);
               i++;
            } while (i < 10);

            // for
            Console.WriteLine("\n\nFor...");
            for (i = 0; i < 10; i++)
            {
               Console.Write("{0} ", i);
            }
            Console.WriteLine("\n");
         }
      }
   }
}
```

The best way to analyze this demonstration program is section by section.

goto

The venerable goto statement is not much respected these days (except when used within a switch statement as shown above), but it is the root of all looping. Using goto is quite straightforward, and requires only three steps:

1. Create a label.
2. Perform some work.
3. Goto the label if some condition is met.

Set up your state by initializing the integer variable i to zero. Then create the label: *repeat* and begin work. In this case, the work is nothing more than printing out the value of i and then incrementing i. The ++ operator adds one to the value of i. Test the value of i, and if it is less than ten, go to the label and loop through printing and incrementing the value again:

```
// goto
 Console.WriteLine("goto...");
 int i = 0;
 repeat:
 Console.Write("{0} ", i);
 i++;
 if (i < 10)
    goto repeat;
```

The result of running just this block of code is shown in Figure 9.

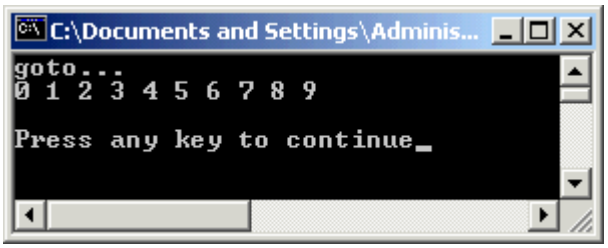

Figure 9. The goto loop.

To see how this works, put a breakpoint at the initialization of i and step through the loop. Each time through, you'll see i incremented and then the test in the if statement. Each time that i is less than 10 the goto executes, returning you to the *repeat* label. Once i is 10, the if statement fails and the program ends.

The Trouble with goto

Goto has a terrible reputation with computer scientists, and for good reason. If you imagine that there is a line drawn in your code indicating the order of execution, goto statements can cause the line of execution to loop over itself. Repeated goto statements, sprinkled liberally throughout the code can create loops within loops that look not unlike a plate of spaghetti.

This is why goto statements are said to create "spaghetti code." The truth is somewhat more complicated: goto statements are not inherently evil. You can write fine, structured code containing goto. The problem is that goto

statements also allow you to write horrible, unstructured code. The solution is to use more structured constructs, such as while, do while, and for loops.

while

The while loop shown in the listing is a good example of how you can accomplish the same goal you did with the goto statement, but in a more structured statement.

The semantics of a while statement are: *While this condition remains true, take this action*. *While i is less than ten, print its value*.

```
i = 0;
while (i < 10)
{
   Console.Write("{0} ", i);
   i++;
}
```

NOTE We use the value i for counting variables in this and many other examples. This is a long tradition in computer programming, and in fact it goes all the way back to an early programming language: Fortran. In Fortran the only legal counting variables were i, j, k, and l. Many programmers continue this tradition of using i, j, k, and l for simple counting variables, even if they don't know why.

What is appealing about this code is that the braces enclose the entire action, and the while statement sets the condition. You can see how this works and there is no jumping all over the code willy-nilly. The condition is tested, and if it evaluates true (i is less than ten) the action within the braces is executed. The result of running the while loop is shown in Figure 10.

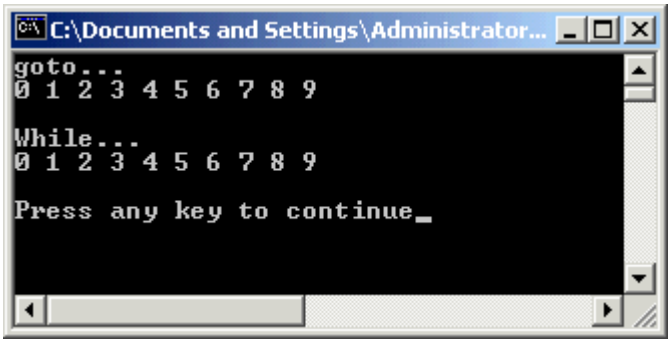

Figure 10. Executing the while loop.

You can see that the output from the while loop is identical to the goto, but the code is somewhat cleaner. In large programs the code is infinitely cleaner.

do…while

The while loop has one feature that may represent a limitation in certain circumstances. The condition (i < 10) is tested *before* the loop is executed.

Notice that just before testing the while loop, you've assigned the value 0 to i. This is necessary because the previous test of goto left i with the value of 10. If you comment out the assignment just before running the while loop, the results are dramatically different, as shown in Figure 11.

```
// goto
Console.WriteLine("goto...");
int i = 0;
repeat:
Console.Write("{0} ", i);
i++;
if (i < 10)
   goto repeat;

// while
Console.WriteLine("\n\nWhile...");
// i = 0;
while (i < 10)
{
   Console.Write("{0} ", i);
   i++;
}
```

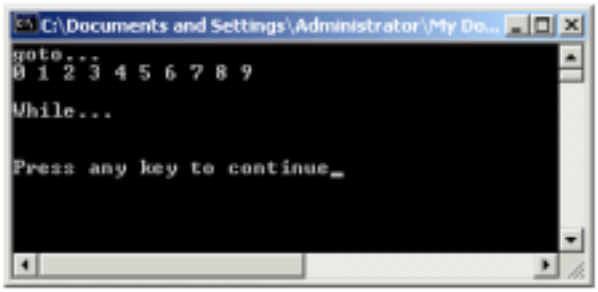

Figure 11. Commenting out the assignment to i.

This time the while loop does not print any values at all, because the test (i < 10) fails, and the body never executes.

There are times when you want to ensure that your loop runs at least once. For this, use a do..while loop. A do..while loop works just like a while loop, except

that the test is executed *after* the statement executes. In most circumstances, this won't make any difference. If you initialize the i to 0 each time, the loops will run identically, as shown in Figure 12.

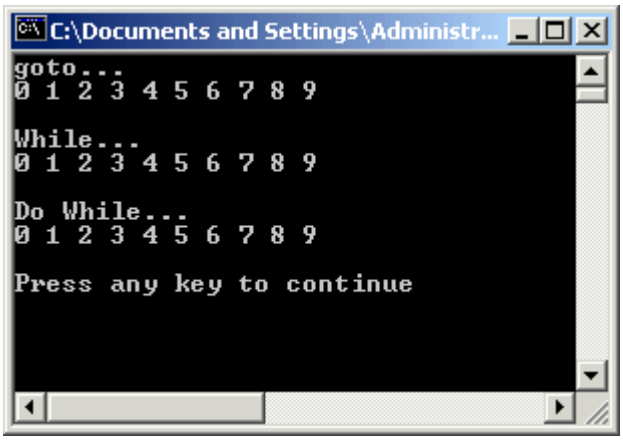

Figure 12. While and DoWhile when i = 0.

If, however, you remove the initialization of i to 0 both in the while and the do..while loops, the difference is shown, as seen in Figure 13.

```
// goto
Console.WriteLine("goto...");
int i = 0;
repeat:
Console.Write("{0} ", i);
i++;
if (i < 10)
   goto repeat;

// while
Console.WriteLine("\n\nWhile...");
// i = 0;
while (i < 10)
{
   Console.Write("{0} ", i);
   i++;
}

// do...while
Console.WriteLine("\n\nDo While...");
// i = 0;
do
{
   Console.Write("{0} ", i);
   i++;
} while (i < 10);
```

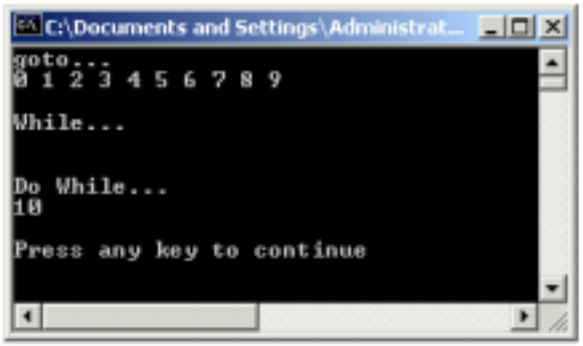

Figure 13. While and do while when i = 10.

This time i is not set to 0, it is left at 10. The while loop test fails, so nothing is printed. The do..while loop executes once and then the value is tested. Since the test fails, the do..while loop is not executed again.

for Loops

Most loops, in one way or another, replicate these steps:

1. Set an initial condition.
2. Test the condition and if the test evaluates true, execute the body of the loop.
3. Change the condition.
4. Test the condition and if the test evaluates true, execute the body of the loop.
5. Repeat Steps 3 and 4 while the condition remains true.

The for loop combines Steps 2 through 4 into a single statement.

The head of the for loop (within the parentheses) is divided into three sections, separated by semicolons. The first section sets the initial conditions. The second section is the test, and the third section is the action to take after the body of the for statement executes.

```
for (i = 0; i < 10; i++)
{
   Console.Write("{0} ", i);
}
```

In the code shown, the variable i is initialized to 0 in the first section. This section only executes once; the first time the loop is run. The second section is the test; the variable is tested to ensure its value is less than 10. If so, the body of the for loop (within the braces) is run.

When the body completes, the third section executes (i++) and then the test is executed again. If the test evaluates true, the body is run again. This repeats until the test fails.

You can leave any of the sections blank and can move the initialization out of the for loop:

```
i = 0;
for (; i < 10; i++)
{
   Console.Write("{0} ", i);
}
```

You can move the action into the body of the loop:

```
Console.WriteLine("\n\nFor...");
 i = 0;
 for (; i < 10; )
 {
    Console.Write("{0} ", i);
    i++;
 }
```

You can, if you are particularly ornery, even move the test into the body of the loop:

```
i = 0;
for (;; )
{
   Console.Write("{0} ", i);
   i++;
   if (i >= 10)
      break;
}
```

This does somewhat undermine the point of the for loop, but it also illustrates quite dramatically that each part of the for loop is optional. You must hold its place with the semicolons, and you can break out of a for loop (or any loop) with the break keyword.

Summary

- Program execution continues in the order of the program statements, unless interrupted by a branching statement.
- Branching can be conditional or unconditional. Unconditional branching is most often accomplished with a method call.
- When a method is invoked, program execution branches to the top of the. When the method returns, execution resumes with the statement in the calling method that immediately follows the method invocation.
- A value may be returned by a method and that value may be used locally within the calling method.
- Conditional branching can be accomplished with the if statement. If statements may have an else clause and may also have other if statements nested within them.
- The switch statement may be used to selectively branch based on a particular value.
- C# programs can loop using a number of constructs. The most primitive looping construct is the if statement, but C# also supports the while, do..while, and for loop.

Questions

1. What is the difference between a conditional and an unconditional branch?
2. What is the return type to indicate that no value will be returned by a method?
3. When is an else statement executed?
4. What is the disadvantage of goto?
5. Why would you use do..while rather than while?
6. What is the advantage of using a for loop rather than a while loop?

Answers

1. What is the difference between a conditional and an unconditional branch?
   A conditional branch depends on the Boolean state of an expression being tested. An unconditional branch occurs no matter what the state of the program.

2. What is the return type to indicate that no value will be returned by a method?
   The return type to indicate that no value is returned is *void*.

3. When is an else statement executed?
   An else statement is executed when the expression tested by its if statement evaluates false. You cannot have an else statement without an if statement.

4. What is the disadvantage of goto?
   Programs using goto branch n ways that are hard to trace when examining the code, and create code that is difficult to understand and maintain.

5. Why would you use do..while rather than while?
   The do..while loop is guaranteed to run at least once, since the test executes after the body of the loop. Use do..while anytime you must ensure that the body of the loop will execute at least once.

6. What is the advantage of using a for loop rather than a while loop?
   The for loop combines the initialization, test, and update of the trackng value into a single header statement. for loops are generally cleaner and easier to understand.

Lab 3: Branching

Lab 3 Overview

In this lab you'll learn how to work with conditional statements and how to work with loops.

To complete this lab, you'll need to work through two exercises:

- Branching Your Program Using Conditionals
- Repeating Work with Looping

Each exercise includes an "Objective" section that describes the purpose of the exercise. You are encouraged to try to complete the exercise from the information given in the Objective section. If you require more information to complete the exercise, the Objective section is followed by detailed step-by-step instructions.

Branching Your Program Using Conditionals

Objective

In this exercise, you'll create code that branches based on a condition.

Things to Consider

- When would you use if/else vs. switch?
- How might you exit a program quickly if you cannot continue?

Step-by-Step Instructions

1. Create a new console application.
2. Within Main, ask the user for his/her age. If the user is under 21, write a message and exit the program:

```
using System;

namespace Conditionals
{
   class GuessIt
   {
      static void Main()
      {

         Console.Write("Enter your age: ");

         // read the user's age from the keyboard
         string strAge = Console.ReadLine();

         // convert it to an integer
         int age = Convert.ToInt32(strAge);
```

```
            // don't let them continue if they're not of age
            if (age < 21)
            {
               Console.WriteLine(
                "Oh, you are just too young to play this
game!");
               return;
            }
            else
               Console.WriteLine(
               "Step right up, and watch carefully....");
```

3. Prompt the user to guess a number between 1 and 10. Convert the value to an integer and switch on the value, printing appropriate messages based on how close they are to the value that the computer is thinking of (4).

```
            Console.Write(
            "Enter a number between 1 and 10: ");

            // read the user's guess from the keyboard
            string strGuess = Console.ReadLine();

            // convert the guess to an integer
            int guess = Convert.ToInt32(strGuess);

            // switch on the guess and choose
            // the appropriate action
            switch (guess)
            {
               case 1:
               case 2:
                  Console.WriteLine("Nope, way too low...");
                  break;
               case 3:
                  Console.WriteLine("Very close...");
                  break;
               case 4:
```

```
                    Console.WriteLine("You got it!!");
                    break;
                case 5:
                    Console.WriteLine("You just missed it!");
                    break;
                case 6:
                case 7:
                    Console.WriteLine("A bit high...");
                    break;
                case 8:
                case 9:
                case 10:
                    Console.WriteLine("Way too high!");
                    break;
                default:
                    Console.WriteLine(
                       "I'm sorry, that is not a valid guess");
                    break;
            }               // end switch
        }                   // end Main
    }                       // end class GuessIt
}                           // end namespace
```

Repeating Work with Looping

Objective

In this exercise, you'll add loops to the previous game to make it more interesting.

Things to Consider

- What kind of loop will work best to allow the user to keep playing after each guess?
- What is the condition for ending the loop?

Step-by-Step Instructions

1. Reopen the previous exercise.

2. Modify the code to declare the integer variable *guess* before you prompt for the number, and initialize it to 0.

```
int guess = 0;
```

3. Create the while loop, testing the new value *guess*, and continuing the loop until guess is equal to 4.

```
int guess = 0;
while (guess != 4)
{
  Console.Write("Enter a number between 1 and 10: ");

  // read the user's guess from the keyboard
  string strGuess = Console.ReadLine();

  // note that you no longer declare guess here
  guess = Convert.ToInt32(strGuess);
```

4. The rest of the code is unchanged. Remember to close the while loop (with a closing brace).

```
                default:
                    Console.WriteLine(
                    "I'm sorry, that is not a valid guess");
                    break;
                }           // end while
            }               // end switch
        }                   // end Main
    }                       // end class GuessIt
}                           // end namespace
```

5. Compile and run your program as shown in Figure 14.

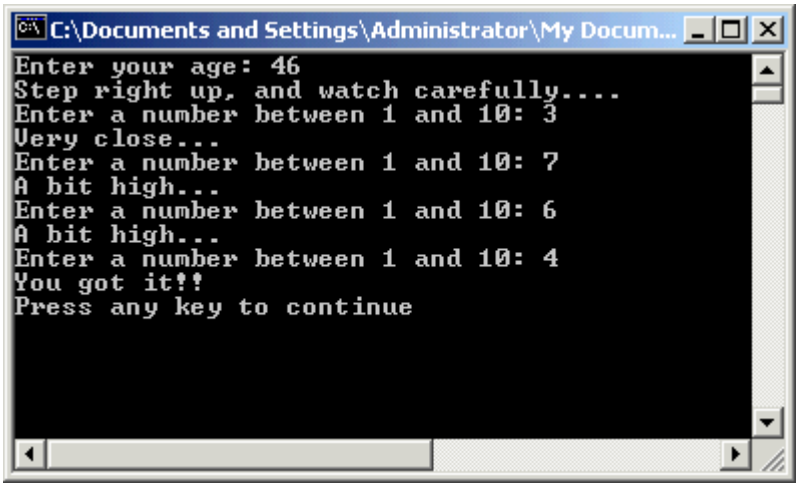

Figure 14. Running the looping application.