**Operators, Types, and Variables**

This lesson introduces C# operators, types, and variables. Its goal is to meet the following objectives:

* Understand what a variable is.
* Familiarization with C# built-in types.
* Get an introduction to C# operators.
* Learn how to use Arrays.

**Variables and Types**

"Variables" are simply storage locations for data. You can place data into them and retrieve their contents as part of a C# expression. The interpretation of the data in a variable is controlled through "Types".

C# is a "Strongly Typed" language. Thus all operations on variables are performed with consideration of what the variable's "Type" is. There are rules that define what operations are legal in order to maintain the integrity of the data you put in a variable.

The C# simple types consist of the Boolean type and three numeric types - Integrals, Floating Point, Decimal, and String. The term "Integrals", which is defined in the C# Programming Language Specification, refers to the classification of types that include sbyte, byte, short, ushort, int, uint, long, ulong, and char. More details are available in the Integral Types section later in this lesson. The term "Floating Point" refers to the float and double types, which are discussed, along with the decimal type, in more detail in the Floating Point and Decimal Types section later in this lesson. The string type represents a string of characters and is discussed in The String Type section, later in this lesson. The next section introduces the boolean type.

**The Boolean Type**

Boolean types are declared using the keyword, *bool*. They have two values: *true* or *false*. In other languages, such as C and C++, boolean conditions can be satisfied where 0 means false and anything else means true. However, in C# the only values that satisfy a boolean condition is *true* and *false*, which are official keywords. Listing 2-1 shows one of many ways that boolean types can be used in a program.

**Listing 2-1. Displaying Boolean Values: Boolean.cs**

using System;  
  
class Booleans  
{  
   publicstaticvoid Main()  
    {  
        bool content = true;  
        bool noContent = false;  
  
        Console.WriteLine("It is {0} that C# Station provides C# programming language content.", content);  
        Console.WriteLine("The statement above is not {0}.", noContent);  
    }  
}

In Listing 2-1, the boolean values are written to the console as a part of a sentence. The only legal values for the *bool* type are either *true* or *false*, as shown by the assignment of *true* to *content* and *false* to *noContent*. When run, this program produces the following output:

It is True that C# Station provides C# programming language content.

The statement above is not False.

**Integral Types**

In C#, an *integral* is a category of types. For anyone confused because the word Integral sounds like a mathematical term, from the perspective of C# programming, these are actually defined as Integral types in the C# programming language specification. They are whole numbers, either signed or unsigned, and the char type. The char type is a Unicode character, as defined by the Unicode Standard. For more information, visit [The Unicode Home Page](http://www.unicode.org/). table 2-1 shows the integral types, their size, and range.

**Table 2-1. The Size and Range of C# Integral Types**

|  |  |  |
| --- | --- | --- |
| **Type** | **Size (in bits)** | **Range** |
| sbyte | 8 | -128 to 127 |
| byte | 8 | 0 to 255 |
| short | 16 | -32768 to 32767 |
| ushort | 16 | 0 to 65535 |
| int | 32 | -2147483648 to 2147483647 |
| uint | 32 | 0 to 4294967295 |
| long | 64 | -9223372036854775808 to 9223372036854775807 |
| ulong | 64 | 0 to 18446744073709551615 |
| char | 16 | 0 to 65535 |

Integral types are well suited for those operations involving whole number calculations. The *char* type is the exception, representing a single Unicode character. As you can see from the table above, you have a wide range of options to choose from, depending on your requirements.

**Floating Point and Decimal Types**

A C# floating point type is either a float or double. They are used any time you need to represent a real number, as defined by IEEE 754. For more information on IEEE 754, visit the [IEEE Web Site](http://www.ieee.org/). Decimal types should be used when representing financial or money values. table 2-2 shows the floating point and decimal types, their size, precision, and range.

**Table 2-2. The Floating Point and Decimal Types with Size, precision, and Range**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Size (in bits)** | **precision** | **Range** |
| float | 32 | 7 digits | 1.5 x 10-45 to 3.4 x 1038 |
| double | 64 | 15-16 digits | 5.0 x 10-324 to 1.7 x 10308 |
| decimal | 128 | 28-29 decimal places | 1.0 x 10-28 to 7.9 x 1028 |

Floating point types are used when you need to perform operations requiring fractional representations. However, for financial calculations, the *decimal* type is the best choice because you can avoid rounding errors.

**The string Type**

A string is a sequence of text characters. You typically create a string with a string literal, enclosed in quotes: "This is an example of a string." You've seen strings being used in Lesson 1, where we used the *Console.WriteLine* method to send output to the console.

Some characters aren't printable, but you still need to use them in strings. Therefore, C# has a special syntax where characters can be escaped to represent non-printable characters. For example, it is common to use newlines in text, which is represented by the '\n' char. The backslash, '\', represents the escape. When preceded by the escape character, the 'n' is no longer interpreted as an alphabetical character, but now represents a newline.

You may be now wondering how you could represent a backslash character in your code. We have to escape that too by typing two backslashes, as in '\\'. table 2-3 shows a list of common escape sequences.

**Table 2-3. C# Character Escape Sequences**

|  |  |
| --- | --- |
| **Escape Sequence** | **Meaning** |
| \' | Single Quote |
| \" | Double Quote |
| \\ | Backslash |
| \0 | Null, not the same as the C# *null* value |
| \a | Bell |
| \b | Backspace |
| \f | form Feed |
| \n | Newline |
| \r | Carriage Return |
| \t | Horizontal Tab |
| \v | Vertical Tab |

Another useful feature of C# strings is the verbatim literal, which is a string with a @ symbol prefix, as in *@"Some string"*. Verbatim literals make escape sequences translate as normal characters to enhance readability. To appreciate the value of verbatim literals, consider a path statement such as *"c:\\topdir\\subdir\\subdir\\myapp.exe"*. As you can see, the backslashes are escaped, causing the string to be less readable. You can improve the string with a verbatim literal, like this: *@"c:\topdir\subdir\subdir\myapp.exe"*.

That is fine, but now you have the problem where quoting text is not as easy. In that case, you would specify double double quotes. For example, the string *"copy \"c:\\source file name with spaces.txt\" c:\\newfilename.txt"* would be written as the verbatim literal *@"copy ""c:\source file name with spaces.txt"" c:\newfilename.txt"*.

**C# Operators**

Results are computed by building expressions. These expressions are built by combining variables and operators together into statements. The following table describes the allowable operators, their precedence, and associativity.

**Table 2-4. Operators with their precedence and Associativity**

|  |  |  |
| --- | --- | --- |
| **Category (by precedence)** | **Operator(s)** | **Associativity** |
| Primary | x.y  f(x)  a[x]  x++  x--  new  typeof  default  checked  unchecked delegate | left |
| Unary | +  -  !  ~  ++x  --x  (T)x | right |
| Multiplicative | \*  /  % | left |
| Additive | +  - | left |
| Shift | <<  >> | left |
| Relational | <  >  <=  >=  is as | left |
| Equality | ==  != | right |
| Logical AND | & | left |
| Logical XOR | ^ | left |
| Logical OR | | | left |
| Conditional AND | && | left |
| Conditional OR | || | left |
| Null Coalescing | ?? | left |
| Ternary | ?: | right |
| Assignment | =  \*=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=  => | right |

Left associativity means that operations are evaluated from left to right. Right associativity mean all operations occur from right to left, such as assignment operators where everything to the right is evaluated before the result is placed into the variable on the left.

Most operators are either unary or binary. Unary operators form expressions on a single variable, but binary operators form expressions with two variables. Listing 2-2 demonstrates how unary operators are used.

**Listing 2-2. Unary Operators: Unary.cs**

using System;  
  
class Unary  
{  
   publicstaticvoid Main()  
    {  
        int unary = 0;  
        int preIncrement;  
        int preDecrement;  
        int postIncrement;  
        int postDecrement;  
        int positive;  
        int negative;  
        sbyte bitNot;  
        bool logNot;  
  
        preIncrement = ++unary;  
        Console.WriteLine("pre-Increment: {0}", preIncrement);  
  
        preDecrement = --unary;  
        Console.WriteLine("pre-Decrement: {0}", preDecrement);  
  
        postDecrement = unary--;  
        Console.WriteLine("Post-Decrement: {0}", postDecrement);  
  
        postIncrement = unary++;  
        Console.WriteLine("Post-Increment: {0}", postIncrement);  
  
        Console.WriteLine("Final Value of Unary: {0}", unary);  
  
        positive = -postIncrement;  
        Console.WriteLine("Positive: {0}", positive);  
  
        negative = +postIncrement;  
        Console.WriteLine("Negative: {0}", negative);  
  
        bitNot = 0;  
        bitNot = (sbyte)(~bitNot);  
        Console.WriteLine("Bitwise Not: {0}", bitNot);  
  
        logNot = false;  
        logNot = !logNot;  
        Console.WriteLine("Logical Not: {0}", logNot);  
    }  
}

When evaluating expressions, post-increment *(x++)* and post-decrement *(x--)* operators return their current value and then apply the operators. However, when using pre-increment *(++x)* and pre-decrement *(--x)* operators, the operator is applied to the variable prior to returning the final value.

In Listing 2-2, the *unary* variable is initialized to zero. When the pre-increment *(++x)* operator is used, *unary* is incremented to 1 and the value 1 is assigned to the *preIncrement* variable. The pre-decrement *(--x)* operator turns *unary* back to a 0 and then assigns the value to the *preDecrement* variable.

When the post-decrement *(x--)* operator is used, the value of *unary*, 0, is placed into the *postDecrement* variable and then *unary* is decremented to -1. Next the post-increment *(x++)*operator moves the current value of *unary*, -1, to the *postIncrement* variable and then increments *unary* to 0.

The variable *bitNot* is initialized to 0 and the bitwise not *(~)* operator is applied. The bitwise not *(~)* operator flips the bits in the variable. In this case, the binary representation of 0, "00000000", was transformed into -1, "11111111".

While the *(~)* operator works by flipping bits, the logical negation operator *(!)* is a logical operator that works on *bool* values, changing *true* to *false* or *false* to *true*. In the case of the *logNot* variable in Listing 2-2, the value is initialized to *false*, and the next line applies the logical negation operator, *(!)*, which returns *true* and reassigns the new value, *true*, to *logNot*. Essentially, it is toggling the value of the *bool* variable, *logNot*.

The setting of *positive* is a little tricky. At the time that it is set, the *postIncrement* variable is equal to -1. Applying the minus *(-)* operator to a negative number results in a positive number, meaning that *positive* will equal 1, instead of -1. The minus operator *(-)*, which is not the same as the pre-decrement operator (--), doesn't change the value of *postInc* - it just applies a sign negation. The plus operator *(+)* doesn't affect the value of a number, assigning *negative* with the same value as *postIncrement*, -1.

Notice the expression *(sbyte)(~bitNot)*. Any operation performed on types *sbyte*, *byte*, *short*, or *ushort* return *int* values. To assign the result into the *bitNot* variable we had to use a cast, *(Type)*, operator, where *Type* is the type you wish to convert to (in this case - *sbyte*). The cast operator is shown as the Unary operator, *(T)x,* in table 2-4. Cast operators must be performed explicity when you go from a larger type to a smaller type because of the potential for lost data. Generally speaking, assigning a smaller type to a larger type is no problem, since the larger type has room to hold the entire value. Also be aware of the dangers of casting between signed and unsigned types. You want to be sure to preserve the integrity of your data. Many basic programming texts contain good descriptions of bit representations of variables and the dangers of explicit casting.

Here's the output from the Listing 2-2:

pre-Increment: 1

pre-Decrement 0

Post-Decrement: 0

Post-Increment: -1

Final Value of Unary: 0

Positive: 1

Negative: -1

Bitwise Not: -1

Logical Not: true

In addition to unary operators, C# has binary operators that form expressions of two variables. Listing 2-3 shows how to use the binary operators.

**Listing 2-3. Binary Operators: Binary.cs**

using System;  
  
class Binary  
{  
   publicstaticvoid Main()  
    {  
        int x, y, result;  
        float floatresult;  
  
        x = 7;  
        y = 5;  
  
        result = x+y;  
        Console.WriteLine("x+y: {0}", result);  
  
        result = x-y;  
        Console.WriteLine("x-y: {0}", result);  
  
        result = x\*y;  
        Console.WriteLine("x\*y: {0}", result);  
  
        result = x/y;  
        Console.WriteLine("x/y: {0}", result);  
  
        floatresult = (float)x/(float)y;  
        Console.WriteLine("x/y: {0}", floatresult);  
  
        result = x%y;  
        Console.WriteLine("x%y: {0}", result);  
  
        result += x;  
        Console.WriteLine("result+=x: {0}", result);  
    }  
}

And here's the output:

x+y: 12

x-y: 2

x\*y: 35

x/y: 1

x/y: 1.4

x%y: 2

result+=x: 9

Listing 2-3 shows several examples of binary operators. As you might expect, the results of addition (+), subtraction (-), multiplication (\*), and division (/) produce the expected mathematical results.

The *floatresult* variable is a floating point type. We explicitly cast the integer variables *x* and *y* to calculate a floating point value.

There is also an example of the remainder(%) operator. It performs a division operation on two values and returns the remainder.

The last statement shows another form of the assignment with operation (+=) operator. Any time you use the assignment with operation operator, it is the same as applying the binary operator to both the left hand and right hand sides of the operator and putting the results into the left hand side. The example could have been written as *result = result + x;* and returned the same value.

**The Array Type**

Another data type is the Array, which can be thought of as a container that has a list of storage locations for a specified type. When declaring an Array, specify the type, name, dimensions, and size.

**Listing 2-4. Array Operations: Array.cs**

using System;  
  
class Array  
{  
   publicstaticvoid Main()  
    {  
        int[] myInts = { 5, 10, 15 };  
        bool[][] myBools = newbool[2][];  
        myBools[0] = newbool[2];  
        myBools[1] = newbool[1];  
        double[,] myDoubles = newdouble[2, 2];  
        string[] myStrings = newstring[3];  
  
        Console.WriteLine("myInts[0]: {0}, myInts[1]: {1}, myInts[2]: {2}", myInts[0], myInts[1], myInts[2]);  
  
        myBools[0][0] = true;  
        myBools[0][1] = false;  
        myBools[1][0] = true;  
        Console.WriteLine("myBools[0][0]: {0}, myBools[1][0]: {1}", myBools[0][0], myBools[1][0]);  
  
        myDoubles[0, 0] = 3.147;  
        myDoubles[0, 1] = 7.157;  
        myDoubles[1, 1] = 2.117;  
        myDoubles[1, 0] = 56.00138917;  
        Console.WriteLine("myDoubles[0, 0]: {0}, myDoubles[1, 0]: {1}", myDoubles[0, 0], myDoubles[1, 0]);  
  
        myStrings[0] = "Joe";  
        myStrings[1] = "Matt";  
        myStrings[2] = "Robert";  
        Console.WriteLine("myStrings[0]: {0}, myStrings[1]: {1}, myStrings[2]: {2}", myStrings[0], myStrings[1], myStrings[2]);  
  
    }  
}

And here's the output:

myInts[0]: 5, myInts[1]: 10, myInts[2]: 15

myBools[0][0]: true, myBools[1][0]: true

myDoubles[0, 0]: 3.147, myDoubles[1, 0]: 56.00138917

myStrings[0]: Joe, myStrings[1]: Matt, myStrings[2]: Robert

Listing 2-4 shows different implementations of Arrays. The first example is the *myInts* Array, which is a single-dimension array. It is initialized at declaration time with explicit values.

Next is a jagged array, *myBools*. It is essentially an array of arrays. We needed to use the *new* operator to instantiate the size of the primary array and then use the *new* operator again for each sub-array.

The third example is a two dimensional array, *myDoubles*. Arrays can be multi-dimensional, with each dimension separated by a comma. It must also be instantiated with the *new* operator.

One of the differences between jagged arrays, *myBools[][]*, and multi-dimension arrays, *myDoubles[,]*, is that a multi-dimension array will allocate memory for every element of each dimension, whereas a jagged array will only allocate memory for the size of each array in each dimension that you define. Most of the time, you'll be using multi-dimension arrays, if you need multiple dimensions, and will only use jagged arrays in very special circumstances when you are able to save significant memory by explicitly specifying the sizes of the arrays in each dimension.

Finally, we have the single-dimensional array of *string* types, *myStrings*.

In each case, you can see that array elements are accessed by identifying the integer index for the item you wish to refer to. Arrays sizes can be any *int* type value. Their indexes begin at 0.

**Summary**

A variable is an identifier with a type that holds a value of that type. Simple types include the integrals, floating points, decimal, and bool. C# has several mathematical and logical operators that participate in forming expressions. C# also offers the single dimension, multi-dimension and jagged array types.

**Control Statements - Selection**

In the last couple of lessons, every program you saw contained a limited amount of sequential steps and then stopped. There were no decisions you could make with the input and the only constraint was to follow straight through to the end. The information in this lesson will help you branch into separate logical sequences based on decisions you make. More specifically, the goals of this lesson are as follows:

* Learn the *if* statements.
* Learn the *switch* statement.
* Learn how *break* is used in *switch* statements.
* Understand proper use of the *goto* statement.

**The *if* Statement**

An *if* statement allows you to take different paths of logic, depending on a given condition. When the condition evaluates to a boolean *true*, a block of code for that true condition will execute. You have the option of a single *if* statement, multiple *else if* statements, and an optional *else* statement. Listing 3-1 shows how each of these types of if statements work.

**Listing 3-1. forms of the *if* statement: IfSelection.cs**

using System;  
  
class IfSelect  
{  
   publicstaticvoid Main()  
    {  
        string myInput;  
        int myInt;

        Console.Write("Please enter a number: ");  
        myInput = Console.ReadLine();  
        myInt = Int32.Parse(myInput);

        // Single Decision and Action with braces  
        if (myInt > 0)  
        {  
            Console.WriteLine("Your number {0} is greater than zero.", myInt);  
        }

        // Single Decision and Action without brackets  
        if (myInt < 0)   
            Console.WriteLine("Your number {0} is less than zero.", myInt);

        // Either/Or Decision  
        if (myInt != 0)  
        {  
            Console.WriteLine("Your number {0} is not equal to zero.", myInt);  
        }  
        else  
       {  
            Console.WriteLine("Your number {0} is equal to zero.", myInt);  
        }

        // Multiple Case Decision  
        if (myInt < 0 || myInt == 0)  
        {  
            Console.WriteLine("Your number {0} is less than or equal to zero.", myInt);  
        }  
        elseif (myInt > 0 && myInt <= 10)  
        {  
            Console.WriteLine("Your number {0} is in the range from 1 to 10.", myInt);  
        }  
        elseif (myInt > 10 && myInt <= 20)  
        {  
            Console.WriteLine("Your number {0} is in the range from 11 to 20.", myInt);  
        }  
        elseif (myInt > 20 && myInt <= 30)  
        {  
            Console.WriteLine("Your number {0} is in the range from 21 to 30.", myInt);  
        }  
        else  
       {  
            Console.WriteLine("Your number {0} is greater than 30.", myInt);  
        }  
    }  
}

The statements in Listing 3-1 use the same input variable, *myInt* as a part of their evaluations. This is another way of obtaining interactive input from the user. Here's the pertinent code:

        Console.Write("Please enter a number: ");  
        myInput = Console.ReadLine();  
        myInt = Int32.Parse(myInput);

We first print the line "Please enter a number: " to the console. The *Console.ReadLine()* statement causes the program to wait for input from the user, who types a number and then presses Enter. This number is returned in the form of a string into the *myInput* variable, which is a string type. Since we must evaluate the user's input in the form of an int, *myInput* must be converted. This is done with the command *Int32.Parse(myInput)*. (*Int32* and similar types will be covered in another lesson on advanced types) The result is placed into the *myInt* variable, which is an *int* type.

Now that we have a variable in the type we wanted, we will evaluate it with *if* statements. The first statement is of the form *if (boolean expression) { statements }*, as shown below:

        // Single Decision and Action with braces  
        if (myInt > 0)  
        {  
            Console.WriteLine("Your number {0} is greater than zero.", myInt);  
        }

You must begin with the keyword *if*. Next is the boolean expression between parenthesis. This boolean expression must evaluate to a *true* or *false* value. In this case, we are checking the user's input to see if it is greater than (>) 0. If this expression evaluates to *true*, we execute the statements within the curly braces. (We refer to the structure with curly braces as a "block") There could be one or more statements within this block. If the boolean expression evaluates to *false*, we ignore the statements inside the block and continue program execution with the next statement after the block.

**Note:** In other languages, such as C and C++, conditions can be evaluated where a result of 0 is false and any other number is true. In C#, the condition must evaluate to a boolean value of either true or false. If you need to simulate a numeric condition with C#, you can do so by writing it as (myInt != 0), which means that the expression evaluates to true if myInt is not 0.

The second *if* statement is much like the first, except it does not have a block, as shown here:

        // Single Decision and Action without braces  
        if (myInt < 0)   
            Console.WriteLine("Your number {0} is less than zero.", myInt);

If its boolean expression evaluates to *true*, the first statement after the boolean expression will be executed. When the boolean expression evaluates to *false*, the first statement after the boolean expression will be skipped and the next program statement will be executed. This form of *if* statement is adequate when you only have a single statement to execute. If you want to execute two or more statements when the boolean expression evaluates to *true*, you must enclose them in a block.

Most of the time, you'll want to make an either/or kind of decision. This is called an if/else statement. The third *if* statement in Listing 3-1 presents this idea, as shown below:

        // Either/Or Decision  
        if (myInt != 0)  
        {  
            Console.WriteLine("Your number {0} is not equal to zero.", myInt);  
        }  
        else  
         {  
            Console.WriteLine("Your number {0} is equal to zero.", myInt);  
        }

When the boolean expression evaluates to *true*, the statement(s) in the block immediately following the *if* statement are executed. However, when the boolean expression evaluates to *false*, the statements in the block following the *else* keyword are executed.

When you have multiple expressions to evaluate, you can use the if/else if/else form of the *if* statement. We show this form in the fourth *if* statement of Listing 3-1, and repeated below:

        // Multiple Case Decision  
        if (myInt < 0 || myInt == 0)  
        {  
            Console.WriteLine("Your number {0} is less than or equal to zero.", myInt);  
        }  
        elseif (myInt > 0 && myInt <= 10)  
        {  
            Console.WriteLine("Your number {0} is in the range from 1 to 10.", myInt);  
        }  
        elseif (myInt > 10 && myInt <= 20)  
        {  
            Console.WriteLine("Your number {0} is in the range from 11 to 20.", myInt);  
        }  
        elseif (myInt > 20 && myInt <= 30)  
        {  
            Console.WriteLine("Your number {0} is in the range from 21 to 30.", myInt);  
        }  
        else  
         {  
            Console.WriteLine("Your number {0} is greater than 30.", myInt);  
        }

This example begins with the *if* keyword, again executing the following block if the boolean expression evaluates to *true*. However, this time you can evaluate multiple subsequent conditions with the *else if* keyword combination. the *else if* statement also takes a boolean expression, just like the *if* statement. The rules are the same, when the boolean expression for the *else if* statement evaluates to *true*, the block immediately following the boolean expression is executed. When none of the other *if* or *else if* boolean expressions evaluate to *true*, the block following the *else* keyword will be executed. Only one section of an *if/else if/else* statement will be executed.

One difference in the last statement from the others is the boolean expressions. The boolean expression, *(myInt < 0 || myInt == 0)*, contains the conditional OR (||) operator. In both the regular OR (|) operator and the conditional OR (||) operator, the boolean expression will evaluate to *true* if either of the two sub-expressions on either side of the operator evaluate to *true*. The primary difference between the two OR forms are that the regular OR operator will evaluate both sub-expressions every time. However, the conditional OR will evaluate the second sub-expression only if the first sub-expression evaluates to false.

The boolean expression, *(myInt > 0 && myInt <= 10)*, contains the conditional AND operator.  Both the regular AND (&) operator and the conditional AND (&&) operator will return *true* when both of the sub-expressions on either side of the operator evaluate to *true*.  The difference between the two is that the regular AND operator will evaluate both expressions every time. However, the conditional AND operator will evaluate the second sub-expression only when the first sub-expression evaluates to *true*.

The conditional operators (&& and ||) are commonly called short-circuit operators because they do not always evaluate the entire expression. Thus, they are also used to produce more efficient code by ignoring unnecessary logic.

**The *switch* Statement**

Another form of selection statement is the *switch* statement, which executes a set of logic depending on the value of a given parameter. The types of the values a switch statement operates on can be booleans, enums, integral types, and strings. [Lesson 2: Operators, Types, and Variables](http://www.csharp-station.com/Tutorial/CSharp/Lesson02) discussed the bool type, integral types and strings and [Lesson 17: Enums](http://www.csharp-station.com/Tutorial/CSharp/Lesson17) will teach you what an *enum* type is. Listing 3-2 shows how to use the *switch* statement with both *int* and *string* types.

**Listing 3-2. Switch Statements: SwitchSelection.cs**

using System;  
  
class SwitchSelect  
{  
   publicstaticvoid Main()  
    {  
        string myInput;  
        int myInt;  
  
        begin:  
  
        Console.Write("Please enter a number between 1 and 3: ");  
        myInput = Console.ReadLine();  
        myInt = Int32.Parse(myInput);  
  
        // switch with integer type  
        switch (myInt)  
        {  
            case 1:  
                Console.WriteLine("Your number is {0}.", myInt);  
                break;  
            case 2:  
                Console.WriteLine("Your number is {0}.", myInt);  
                break;  
            case 3:  
                Console.WriteLine("Your number is {0}.", myInt);  
                break;  
            default:  
                Console.WriteLine("Your number {0} is not between 1 and 3.", myInt);  
                break;  
        }  
  
        decide:  
  
        Console.Write("Type \"continue\" to go on or \"quit\" to stop: ");  
        myInput = Console.ReadLine();  
  
        // switch with string type  
        switch (myInput)  
        {  
            case "continue":  
                goto begin;  
            case "quit":  
                Console.WriteLine("Bye.");  
                break;  
            default:  
                Console.WriteLine("Your input {0} is incorrect.", myInput);  
                goto decide;  
        }  
    }  
}

**Note:** Listing 3-2 will throw an exception if you enter any value other than an int. i.e. the letter 'a' would be an error.

Listing 3-2 shows a couple of *switch* statements. The *switch* statement begins with the *switch* keyword followed by the *switch* expression. In the first *switch* statement in listing 3-2, the *switch* expression evaluates to an *int* type, as follows:

        // switch with integer type  
        switch (myInt)  
        {  
            case 1:  
                Console.WriteLine("Your number is {0}.", myInt);  
                break;  
            case 2:  
                Console.WriteLine("Your number is {0}.", myInt);  
                break;  
            case 3:  
                Console.WriteLine("Your number is {0}.", myInt);  
                break;  
            default:  
                Console.WriteLine("Your number {0} is not between 1 and 3.", myInt);  
                break;  
        }

The *switch* block follows the *switch* expression, where one or more choices are evaluated for a possible match with the *switch* expression. Each choice is labeled with the *case* keyword, followed by an example that is of the same type as the *switch* expression and followed by a colon (:). In the example we have *case 1:*, *case 2:*, and *case 3:*. When the result evaluated in the *switch* expression matches one of these choices, the statements immediately following the matching choice are executed, up to and including a branching statement, which could be either a *break*, *continue*, *goto* , *return*, or *throw* statement. table 3-1 summarizes the branching statements.

**Table 3-1. C# Branching Statements**

|  |  |
| --- | --- |
| **Branching statement** | **Description** |
| break | Leaves the switch block |
| continue | Leaves the switch block, skips remaining logic in enclosing loop, and goes back to loop condition to determine if loop should be executed again from the beginning. Works only if switch statement is in a loop as described in [Lesson 04: Control Statements - Loops](http://www.csharp-station.com/Tutorial/CSharp/Lesson04). |
| goto | Leaves the switch block and jumps directly to a label of the form "<labelname>:" |
| return | Leaves the current method. Methods are described in more detail in [Lesson 05: Methods](http://www.csharp-station.com/Tutorial/CSharp/Lesson05). |
| throw | Throws an exception, as discussed in [Lesson 15: Introduction to Exception Handling](http://www.csharp-station.com/Tutorial/CSharp/Lesson15). |

You may also include a *default* choice following all other choices. If none of the other choices match, then the *default* choice is taken and its statements are executed. Although use of the *default* label is optional, I highly recommend that you always include it. This will help catch unforeseen circumstances and make your programs more reliable.

Each *case* label must end with a branching statement, as described in table 3-1, which is normally the *break* statement. The *break* statement will cause the program to exit the *switch* statement and begin execution with the next statement after the *switch* block. There are two exceptions to this: adjacent *case* statements with no code in between or using a *goto* statement. Here's an example that shows how to combine case statements:

        switch (myInt)  
        {  
            case 1:  
            case 2:  
            case 3:  
                Console.WriteLine("Your number is {0}.", myInt);  
                break;  
            default:  
                Console.WriteLine("Your number {0} is not between 1 and 3.", myInt);  
                break;  
        }

By placing *case* statements together, with no code in-between, you create a single case for multiple values. A case without any code will automatically fall through to the next case. The example above shows how the three cases for *myInt* equal to 1, 2, or 3, where *case 1* and *case 2* will fall through and execute code for *case 3.*

A *case* statement can only be an exact match and you can't use logical conditions. If you need to use logical conditions, you can use an *if/else if/else* statement.

Another way to control the flow of logic in a *switch* statement is by using the *goto* statement. You can either jump to another case statement, or jump out of the switch statement. The second *switch* statement in Listing 3-2 shows the use of the *goto* statement, as shown below:

        // switch with string type  
        switch (myInput)  
        {  
            case "continue":  
                goto begin;  
            case "quit":  
                Console.WriteLine("Bye.");  
                break;  
            default:  
                Console.WriteLine("Your input {0} is incorrect.", myInput);  
                goto decide;  
        }

**Note:** in the current example, "continue", is a case of the switch statement -- not the keyword.

The *goto* statement causes program execution to jump to the label following the *goto* keyword. During execution, if the user types in "continue", the *switch* statement matches this input (a *string* type) with the *case "continue":* label and executes the *"goto begin:"* instruction. The program will then leave the *switch* statement and start executing the first program statement following the *begin:* label. This is effectively a loop, allowing you to execute the same code multiple times. The loop will end when the user types the *string* "quit". This will be evaluated with the *case "quit":* choice, which will print "Bye." to the console, break out of the *switch* statement and end the program.

**Warning:** You should not create loops like this. It is \*bad\* programming style. The only reason it is here is because I wanted to show you the syntax of the *goto* statement. Instead, use one of the structured looping statements, described When neither the "continue" nor "quit" strings are entered, the *"default:"case* will be entered. It will print an error message to the console and then execute the *goto decide:* command. This will cause program execution to jump to the first statement following the *decide:* label, which will ask the user if they want to continue or quit. This is effectively another loop.

Clearly, the *goto* statement is powerful and can, under controlled circumstances, be useful. However, I must caution you strongly on its use. The *goto* statement has great potential for misuse. You could possibly create a very difficult program to debug and maintain. Imagine the spaghetti code that could be created by random *goto* statements throughout a program. In the next lesson, I'll show you a better way to create loops in your program.

**Summary**

The *if* statement can be written in multiple ways to implement different branches of logic. The *switch* statement allows a choice among a set of *bool*, *enum*, integral, or *string* types. You use *break*, *continue*, *goto*, *return*, or *throw* statements to leave a case statement. Be sure to avoid the *goto* statement in your code unless you have an extremely good reason for using it.

**Control Statements - Loops**

In the last lesson, you learned how to create a simple loop by using the *goto* statement. I advised you that this is not the best way to perform loops in C#. The information in this lesson will teach you the proper way to execute iterative logic with the various C# looping statements. Its goal is to meet the following objectives:

* Learn the *while* loop.
* Learn the *do* loop.
* Learn the *for* loop.
* Learn the *foreach* loop.
* Complete your knowledge of the *break* statement.
* Teach you how to use the *continue* statement.

**The *while* Loop**

A *while* loop will check a condition and then continues to execute a block of code as long as the condition evaluates to a boolean value of *true*. Its syntax is as follows: *while (<boolean expression>) { <statements> }.* The statements can be any valid C# statements. The boolean expression is evaluated before any code in the following block has executed. When the boolean expression evaluates to *true*, the statements will execute. Once the statements have executed, control returns to the beginning of the *while* loop to check the boolean expression again.

When the boolean expression evaluates to *false*, the *while* loop statements are skipped and execution begins after the closing brace of that block of code. Before entering the loop, ensure that variables evaluated in the loop condition are set to an initial state. During execution, make sure you update variables associated with the boolean expression so that the loop will end when you want it to. Listing 4-1 shows how to implement a *while* loop.

**Listing 4-1. The While Loop: WhileLoop.cs**

using System;  
  
class WhileLoop  
{  
   publicstaticvoid Main()  
    {  
        int myInt = 0;  
  
        while (myInt < 10)  
        {  
            Console.Write("{0} ", myInt);  
            myInt++;  
        }  
        Console.WriteLine();  
    }  
}

Listing 4-1 shows a simple *while* loop. It begins with the keyword *while*, followed by a boolean expression. All control statements use boolean expressions as their condition for entering/continuing the loop. This means that the expression must evaluate to either a *true* or *false* value. In this case we are checking the *myInt* variable to see if it is less than (<) 10. Since *myInt* was initialized to 0, the boolean expression will return *true* the first time it is evaluated. When the boolean expression evaluates to *true*, the block immediately following the boolean expression will be executed.

Within the *while* block we print the number and a space to the console. Then we increment (++) *myInt* to the next integer. Once the statements in the *while* block have executed, the boolean expression is evaluated again. This sequence will continue until the boolean expression evaluates to *false*. Once the boolean expression is evaluated as *false*, program control will jump to the first statement following the *while* block. In this case, we will write the numbers 0 through 9 to the console, exit the *while* block, and print a new line to the console.

**The *do* Loop**

A *do* loop is similar to the *while* loop, except that it checks its condition at the end of the loop. This means that the *do* loop is guaranteed to execute at least one time. On the other hand, a *while* loop evaluates its boolean expression at the beginning and there is generally no guarantee that the statements inside the loop will be executed, unless you program the code to explicitly do so. One reason you may want to use a *do* loop instead of a *while* loop is to present a message or menu such as the one in Listing 4-2 and then retrieve input from a user.

**Listing 4-2. The Do Loop: DoLoop.cs**

using System;  
  
class DoLoop  
{  
   publicstaticvoid Main()  
    {  
        string myChoice;  
  
        do  
       {  
            // Print A Menu  
            Console.WriteLine("My Address Book\n");  
  
            Console.WriteLine("A - Add New Address");  
            Console.WriteLine("D - Delete Address");  
            Console.WriteLine("M - Modify Address");  
            Console.WriteLine("V - View Addresses");  
            Console.WriteLine("Q - Quit\n");  
  
            Console.WriteLine("Choice (A,D,M,V,or Q): ");  
  
            // Retrieve the user's choice  
            myChoice = Console.ReadLine();  
  
            // Make a decision based on the user's choice  
            switch(myChoice)  
            {  
                case "A":  
                case "a":  
                    Console.WriteLine("You wish to add an address.");  
                    break;  
                case "D":  
                case "d":  
                    Console.WriteLine("You wish to delete an address.");  
                    break;  
                case "M":  
                case "m":  
                    Console.WriteLine("You wish to modify an address.");  
                    break;  
                case "V":  
                case "v":  
                    Console.WriteLine("You wish to view the address list.");  
                    break;  
                case "Q":  
                case "q":  
                    Console.WriteLine("Bye.");  
                    break;  
                default:  
                    Console.WriteLine("{0} is not a valid choice", myChoice);  
                    break;  
            }  
  
            // Pause to allow the user to see the results  
            Console.Write("press Enter key to continue...");  
            Console.ReadLine();  
            Console.WriteLine();  
        } while (myChoice != "Q" && myChoice != "q"); // Keep going until the user wants to quit  
   }  
}

Listing 4-2 shows a *do* loop in action. The syntax of the *do* loop is *do { <statements> } while (<boolean expression>);*. The statements can be any valid C# programming statements you like. The boolean expression is the same as all others we've encountered so far. It returns either *true* or *false*.

In the *Main* method, we declare the variable *myChoice* of type *string*. Then we print a series of statements to the console. This is a menu of choices for the user. We must get input from the user, which is in the form of a *Console.ReadLine* method which returns the user's value into the *myChoice* variable. We must take the user's input and process it. A very efficient way to do this is with a *switch* statement. Notice that we've placed matching upper and lower case letters together to obtain the same functionality. This is the only legal way to have automatic fall through between cases. If you were to place any statements between two cases, you would not be able to fall through. Another point is that we used the *default:* case, which is a very good habit for the reasons stated in [Lesson 3: Control Statements - Selection](http://www.csharp-station.com/Tutorial/CSharp/Lesson03).

**The *for* Loop**

A *for* loop works like a *while* loop, except that the syntax of the *for* loop includes initialization and condition modification. *for* loops are appropriate when you know exactly how many times you want to perform the statements within the loop. The contents within the *for* loop parentheses hold three sections separated by semicolons *(<initializer list>; <boolean expression>; <iterator list>) { <statements> }*.

The initializer list is a comma separated list of expressions. These expressions are evaluated only once during the lifetime of the *for* loop. This is a one-time operation, before loop execution. This section is commonly used to initialize an integer to be used as a counter.

Once the initializer list has been evaluated, the *for* loop gives control to its second section, the boolean expression. There is only one boolean expression, but it can be as complicated as you like as long as the result evaluates to *true* or *false*. The boolean expression is commonly used to verify the status of a counter variable.

When the boolean expression evaluates to *true*, the statements within the curly braces of the *for* loop are executed. After executing *for* loop statements, control moves to the top of loop and executes the iterator list, which is normally used to increment or decrement a counter. The iterator list can contain a comma separated list of statements, but is generally only one statement. Listing 4-3 shows how to implement a *for* loop. The purpose of the program is to  print only odd numbers less than 10.

**Listing 4-3. The For Loop: ForLoop.cs**

using System;  
  
class ForLoop  
{  
   publicstaticvoid Main()  
    {  
        for (int i=0; i < 20; i++)  
        {  
            if (i == 10)  
                break;  
  
            if (i % 2 == 0)  
                continue;  
  
            Console.Write("{0} ", i);  
        }  
        Console.WriteLine();  
    }  
}

Normally, *for* loop statements execute from the opening curly brace to the closing curly brace without interruption. However, in Listing 4-3, we've made a couple exceptions. There are a couple *if* statements disrupting the flow of control within the *for* block.

The first *if* statement checks to see if *i* is equal to 10. Now you see another use of the *break* statement. Its behavior is similar to the selection statements, as discussed in [Lesson 3: Control Statements - Selection](http://www.csharp-station.com/Tutorial/CSharp/Lesson03). It simply breaks out of the loop at that point and transfers control to the first statement following the end of the *for* block.

The second *if* statement uses the remainder operator to see if *i* is a multiple of 2. This will evaluate to *true* when *i* is divided by 2 with a remainder equal to zero, (0). When *true*, the *continue* statement is executed, causing control to skip over the remaining statements in the loop and transfer back to the iterator list. By arranging the statements within a block properly, you can conditionally execute them based upon whatever condition you need.

When program control reaches either a *continue* statement or end of block, it transfers to the third section within the *for* loop parentheses, the iterator list. This is a comma separated list of actions that are executed after the statements in the *for* block have been executed. Listing 4-3 is a typical action, incrementing the counter. Once this is complete, control transfers to the boolean expression for evaluation.

Similar to the *while* loop, a *for* loop will continue as long as the boolean expression is *true*. When the boolean expression becomes *false*, control is transferred to the first statement following the *for* block.

For this tutorial, I chose to implement *break* and *continue* statements in Listing 4-3 only. However, they may be used in any of the loop statements.

**The *foreach* Loop**

A *foreach* loop is used to iterate through the items in a list. It operates on arrays or collections such as ArrayList, which can be found in the System.Collections namespace. The syntax of a foreach loop is *foreach (<type><iteration variable> in <list>) { <statements> }*. The type is the type of item contained in the list. For example, if the type of the list was *int[]* then the type would be *int*.

The iteration variable is an identifier that you choose, which could be anything but should be meaningful. For example, if the list contained an array of people's ages, then a meaningful name for item name would be age.

The *in* keyword is required.

As mentioned earlier, the list could be either an array or a collection. You learned about arrays in [Lesson 02: Operators, Types, and Variables](http://www.csharp-station.com/Tutorial/CSharp/Lesson02). You can also iterate over C# generic collections also, described in [Lesson 20: Introduction to Generic Collections](http://www.csharp-station.com/Tutorial/CSharp/Lesson20).

While iterating through the items of a list with a *foreach* loop, the list is read-only. This means that you can't modify the iteration variable within a *foreach* loop. There is a subtlety here; Later, you'll learn how to create custom types, called class and struct, that can contain multiple fields.  You can change the fields of the class or struct, but not the iteration variable for the class or struct itself in a *foreach* loop.

On each iteration through a *foreach* loop the list is queried for a new value. As long as the list can return a value, this value will be put into the read-only iteration variable, causing the statements in the *foreach* block to be executed. When the collection has been fully traversed, control will transfer to the first executable statement following the end of the *foreach* block. Listing 4-4 demonstrates how to use a *foreach* loop.

**Listing 4-4. The ForEach Loop: ForEachLoop.cs**

using System;  
  
class ForEachLoop  
{  
   publicstaticvoid Main()  
    {  
        string[] names = {"Cheryl", "Joe", "Matt", "Robert"};  
  
        foreach (string person in names)  
        {  
            Console.WriteLine("{0} ", person);  
        }  
    }  
}

In Listing 4-4, the first thing we've done inside the *Main* method is declare and initialize the *names* array with 4 *strings*. This is the list used in the *foreach* loop.

In the *foreach* loop, we've used a *string* variable, *person*, as the item name, to hold each element of the *names* array. As long as there are names in the array that have not been returned, the *Console.WriteLine* method will print each value of the *person* variable to the screen.

**Summary**

Loops allow you to execute a block of statements repeatedly. C# offers several statements to construct loops with, including the *while*, *do*, *for*, and *foreach* loops. *while* loops execute a block of statements as long as an expression is *true*, *do* loops execute a block of statements at least once and then keep going as long as a condition is *true*, *for* loops execute a block of statements a specified amount of times, and *foreach* loops execute a block of statements for each item in a collection. Normally a block of statements will execute from beginning to end. However, the normal flow of a loop can be changed with the *break* and *continue* statements.

**Methods**

In previous lessons of this tutorial, all of our functionality for each program resided in the *Main()* method. While this was adequate for the simple programs we used to learn earlier concepts, there is a better way to organize your program, using methods. A method helps you separate your code into modules that perform a given task. The objectives of this lesson are as follows:

* Understand the structure of a method.
* Know the difference between *static* and instance methods.
* Learn to instantiate objects.
* Learn how to call methods of an instantiated object.
* Understand the 4 types of parameters.
* Learn how to use the *this* reference.

**Method Structure**

Methods are extremely useful because they allow you to separate your logic into different units. You can pass information to methods, have it perform one or more statements, and retrieve a return value. The capability to pass parameters and return values is optional and depends on what you want the method to do. Here's a description of the syntax required for creating a method:

*attributes modifiers return-type method-name(parameters )*

*{*

*statements*

*}*

We defer discussion of attributes and modifiers to a later lesson. The return-type can be any C# type. It can be assigned to a variable for use later in the program. The method name is a unique identifier for what you wish to call a method. To promote understanding of your code, a method name should be meaningful and associated with the task the method performs. Parameters allow you to pass information to and from a method. They are surrounded by parenthesis. Statements within the curly braces carry out the functionality of the method.

**Listing 5-1. One Simple Method: OneMethod.cs**

using System;  
  
class OneMethod  
{  
   publicstaticvoid Main()  
    {  
        string myChoice;  
  
        OneMethod om = new OneMethod();  
  
        do  
       {  
            myChoice = om.getChoice();  
  
            // Make a decision based on the user's choice  
            switch(myChoice)  
            {  
                case "A":  
                case "a":  
                    Console.WriteLine("You wish to add an address.");  
                    break;  
                case "D":  
                case "d":  
                    Console.WriteLine("You wish to delete an address.");  
                    break;  
                case "M":  
                case "m":  
                    Console.WriteLine("You wish to modify an address.");  
                    break;  
                case "V":  
                case "v":  
                    Console.WriteLine("You wish to view the address list.");  
                    break;  
                case "Q":  
                case "q":  
                    Console.WriteLine("Bye.");  
                    break;  
                default:  
                    Console.WriteLine("{0} is not a valid choice", myChoice);  
                    break;  
            }  
  
            // Pause to allow the user to see the results  
            Console.WriteLine();  
            Console.Write("press Enter key to continue...");  
  
            Console.ReadLine();  
            Console.WriteLine();  
  
        } while (myChoice != "Q" && myChoice != "q"); // Keep going until the user wants to quit  
   }  
  
    string getChoice()  
    {  
        string myChoice;  
  
        // Print A Menu  
        Console.WriteLine("My Address Book\n");  
  
        Console.WriteLine("A - Add New Address");  
        Console.WriteLine("D - Delete Address");  
        Console.WriteLine("M - Modify Address");  
        Console.WriteLine("V - View Addresses");  
        Console.WriteLine("Q - Quit\n");  
  
        Console.Write("Choice (A,D,M,V,or Q): ");  
  
        // Retrieve the user's choice  
        myChoice = Console.ReadLine();  
        Console.WriteLine();  
  
        return myChoice;   
    }  
}

The program in Listing 5-1 is similar to the *DoLoop* program from Lesson 4, except for one difference. Instead of printing the menu and accepting input in the *Main()* method, this functionality has been moved to a new method called *getChoice()*. The return type is a *string*. This *string* is used in the *switch* statement in *Main()*. The method name "getChoice" describes what happens when it is invoked. Since the parentheses are empty, no information will be transferred to the *getChoice()* method.

Within the method block we first declare the variable *myChoice*. Although this is the same name and type as the *myChoice* variable in *Main()*, they are both unique variables. They are local variables and they are visible only in the block they are declared. In other words, the *myChoice* in *getChoice()* knows nothing about the existence of the *myChoice* in *Main()*, and vice versa.

The *getChoice()* method prints a menu to the console and gets the user's input. The *return* statement sends the data from the *myChoice* variable back to the caller, *Main()*, of *getChoice()*. Notice that the type returned by the *return* statement must be the same as the return-type in the function declaration. In this case it is a *string*.

In the *Main()* method we must instantiate a new *OneMethod* object before we can use *getChoice()*. This is because of the way *getChoice()* is declared. Since we did not specify a *static* modifier, as for *Main()*, *getChoice()* becomes an instance method. The difference between instance methods and *static* methods is that multiple instances of a class can be created (or instantiated) and each instance has its own separate *getChoice()* method. However, when a method is *static*, there are no instances of that method, and you can invoke only that one definition of the *static* method.

So, as stated, *getChoice()* is not *static* and therefore, we must instantiate a new object to use it. This is done with the declaration *OneMethod om = new OneMethod()*. On the left hand side of the declaration is the object reference *om* which is of type *OneMethod*. The distinction of *om* being a reference is important. It is not an object itself, but it is a variable that can refer (or point ) to an object of type *OneMethod*. On the right hand side of the declaration is an assignment of a new *OneMethod* object to the reference *om*. The keyword *new* is a C# operator that creates a new instance of an object on the heap. What is happening here is that a new *OneMethod* instance is being created on the heap and then being assigned to the *om* reference. Now that we have an instance of the *OneMethod* class referenced by *om*, we can manipulate that instance through the *om* reference.

Methods, fields, and other class members can be accessed, identified, or manipulated through the "." (dot) operator. Since we want to call *getChoice()*, we do so by using the dot operator through the *om* reference: *om.getChoice()*. The program then executes the statements in the *getChoice()* block and returns. To capture the value *getChoice()* returns, we use the "=" (assignment) operator. The returned *string* is placed into *Main()'s* local *myChoice* variable. From there, the rest of the program executes as expected, using concepts from earlier lessons.

**Listing 5-2. Method Parameters: MethodParams.cs**

using System;  
  
class Address  
{  
   publicstring name;  
    publicstring address;  
}  
  
class MethodParams  
{  
   publicstaticvoid Main()  
    {  
        string myChoice;  
  
        MethodParams mp = new MethodParams();  
  
        do  
       {  
            // show menu and get input from user  
            myChoice = mp.getChoice();  
  
            // Make a decision based on the user's choice  
            mp.makeDecision(myChoice);  
  
            // Pause to allow the user to see the results  
            Console.Write("press Enter key to continue...");  
            Console.ReadLine();  
            Console.WriteLine();  
        } while (myChoice != "Q" && myChoice != "q"); // Keep going until the user wants to quit  
   }  
  
    // show menu and get user's choice  
   string getChoice()  
    {  
        string myChoice;  
  
        // Print A Menu  
        Console.WriteLine("My Address Book\n");  
  
        Console.WriteLine("A - Add New Address");  
        Console.WriteLine("D - Delete Address");  
        Console.WriteLine("M - Modify Address");  
        Console.WriteLine("V - View Addresses");  
        Console.WriteLine("Q - Quit\n");  
  
        Console.WriteLine("Choice (A,D,M,V,or Q): ");  
  
        // Retrieve the user's choice  
        myChoice = Console.ReadLine();  
  
        return myChoice;   
    }  
  
    // make decision  
   void makeDecision(string myChoice)  
    {  
        Address addr = new Address();  
  
        switch(myChoice)  
        {  
            case "A":  
            case "a":  
                addr.name = "Joe";  
                addr.address = "C# Station";  
                this.addAddress(ref addr);  
                break;  
            case "D":  
            case "d":  
                addr.name = "Robert";  
                this.deleteAddress(addr.name);  
                break;  
            case "M":  
            case "m":  
                addr.name = "Matt";  
                this.modifyAddress(out addr);  
                Console.WriteLine("Name is now {0}.", addr.name);  
                break;  
            case "V":  
            case "v":  
                this.viewAddresses("Cheryl", "Joe", "Matt", "Robert");  
                break;  
            case "Q":  
            case "q":  
                Console.WriteLine("Bye.");  
                break;  
            default:  
                Console.WriteLine("{0} is not a valid choice", myChoice);  
                break;  
        }  
    }  
  
    // insert an address  
   void addAddress(ref Address addr)  
    {  
        Console.WriteLine("Name: {0}, Address: {1} added.", addr.name, addr.address);   
    }  
  
    // remove an address  
   void deleteAddress(string name)  
    {  
        Console.WriteLine("You wish to delete {0}'s address.", name);   
    }  
  
    // change an address  
   void modifyAddress(out Address addr)  
    {  
        //Console.WriteLine("Name: {0}.", addr.name); // causes error!  
        addr = new Address();  
        addr.name = "Joe";  
        addr.address = "C# Station";  
    }  
  
    // show addresses  
   void viewAddresses(paramsstring[] names)  
    {  
        foreach (string name in names)  
        {  
            Console.WriteLine("Name: {0}", name);  
        }  
    }  
}

Listing 5-2 is a modification of Listing 5-1, modularizing the program and adding more implementation to show parameter passing. There are 4 kinds of parameters a C# method can handle: *out*, *ref*, *params*, and *value*. To help illustrate usage of parameters, we created an *Address* class with two string fields.

In *Main()* we call *getChoice()* to get the user's input and put that *string* in the *myChoice* variable. Then we use *myChoice* as an argument to *makeDecision()*. In the declaration of *makeDecision()* you'll notice its one parameter is declared as a *string* with the name *myChoice*. Again, this is a new *myChoice*, separate from the caller's argument and local only to this method. Since *makeDecision()'s myChoice* parameter does not have any other modifiers, it is considered a *value* parameter. The actual value of the argument is copied on the stack. Variables given by *value* parameters are local and any changes to that local variable do not affect the value of the variable used in the caller's argument.

The *switch* statement in *makeDecision()* calls a method for each case. These method calls are different from the ones we used in *Main()*. Instead of using the *mp* reference, they use the *this* keyword. *this* is a reference to the current object. We know the current object has been instantiated because *makeDecision()* is not a *static* method. Therefore, we can use the *this* reference to call methods within the same instance.

The *addAddress()* method takes a *ref* parameter. This means that a reference to the parameter is copied to the method. This reference still refers to the same object on the heap as the original reference used in the caller's argument. This means any changes to the local reference's object also changes the caller reference's object. The code can't change the reference, but it can make changes to the object being referenced. You can think of this as a way to have an input/output parameter.

As you know, methods have return values, but sometimes you'll want to return more than one value from a method. An *out* parameter allows you to return additional values from a method.

*modifyAddress()* has an *out* parameter. *out* parameters are only passed back to the calling function. Because of definite assignment rules, you cannot use this variable until it has a valid value assigned. The first line in *modifyAddress()* is commented on purpose to illustrate this point. Uncomment it and compile to see what happens. Once assigned and the program returns, the value of the *out* parameter will be copied into the caller's argument variable. You must assign a value to an *out* parameter before your method returns.

A very useful addition to the C# language is the *params* parameter, which lets you define a method that can accept a variable number of arguments. The *params* parameter must be a single dimension or jagged array. When calling *viewAddresses()*, we pass in four string arguments. The number of arguments is variable and will be converted to a *string[]* automatically. In *viewAddresses()* we use a *foreach* loop to print each of these *strings*. Instead of the list of *string* arguments, the input could have also been a *string* array. The *params* parameter is considered an input only parameter and any changes affect the local copy only.

In summary, you understand the structure of a method. The four types of paramters are *value*, *ref*, *out*, and *params*. When you wish to use an instance method, you must instantiate its object as opposed to *static* methods that can be called any time. The *this* reference refers to its containing object and may be used to refer to its containing object's members, including methods.

**Namespaces**

This lesson introduces you to C# Namespaces.  Our objectives are as follows:

* Understand what Namespace is.
* Learn how to implement the *using* directive.
* Learn to use *alias* directive.
* Understand what are namespace members.

In Lesson 1, you saw the *using System;* directive in the *SimpleHello* program. This directive allowed you to use members of the *Systemnamespace*. Because of the narrow focus of that lesson, we needed to delay explanation until now. When you've completed this lesson you will understand the *using* directive and more.

Namespaces are C# program elements designed to help you organize your programs. They also provide assistance in avoiding name clashes between two sets of code. Implementing Namespaces in your own code is a good habit because it is likely to save you from problems later when you want to reuse some of your code. For example, if you created a class named Console, you would need to put it in your own namespace to ensure that there wasn't any confusion about when the System.Console class should be used or when your class should be used. Generally, it would be a bad idea to create a class named Console, but in many cases your classes will be named the same as classes in either the .NET Framework Class Library or a third party library and namespaces help you avoid the problems that identical class names would cause.

*Namespaces* don't correspond to file or directory names. If naming directories and files to correspond to *namespaces* helps you organize your code, then you may do so, but it is not required.

**Listing 6-1. The C# Station Namespace: NamespaceCSS.cs**

// Namespace Declaration  
using System;  
  
// The C# Station Namespace  
namespace csharp\_station   
{  
    // Program start class  
    class NamespaceCSS   
    {  
        // Main begins program execution.  
        publicstaticvoid Main()   
        {  
            // Write to console  
            Console.WriteLine("This is the new C# Station Namespace.");   
        }  
    }  
}

Listing 6-1 shows how to create a *namespace*.  We declare the new *namespace* by putting the word *namespace* in front of *csharp\_station*.  Curly braces surround the members inside the *csharp\_stationnamespace*.

**Listing 6-2. Nested Namespace 1: NestedNamespace1.cs**

// Namespace Declaration  
using System;  
  
// The C# Station Tutorial Namespace  
namespace csharp\_station   
{  
    namespace tutorial   
    {  
        // Program start class  
        class NamespaceCSS   
        {  
            // Main begins program execution.  
            publicstaticvoid Main()   
            {  
                // Write to console  
                Console.WriteLine("This is the new C# Station Tutorial Namespace.");  
            }  
        }  
    }  
}

*Namespaces* allow you to create a system to organize your code. A good way to organize your *namespaces* is via a hierarchical system. You put the more general names at the top of the hierarchy and get more specific as you go down. This hierarchical system can be represented by nested *namespaces*. Listing 6-2 shows how to create a nested *namespace*. By placing code in different sub-namespaces, you can keep your code organized.

**Listing 6-3. Nested Namespace 2:  NestedNamespace2.cs**

// Namespace Declaration  
using System;  
  
// The C# Station Tutorial Namespace  
namespace csharp\_station.tutorial   
{  
    // Program start class  
    class NamespaceCSS   
    {  
        // Main begins program execution.  
        publicstaticvoid Main()   
        {  
            // Write to console  
            Console.WriteLine("This is the new C# Station Tutorial Namespace.");   
        }  
    }  
}

Listing 6-3 shows another way of writing nested *namespaces*. It specifies the nested *namespace* with the dot operator between *csharp\_station* and *tutorial*. The result is exactly the same as Listing 6-2. However, Listing 6-3 is easier to write.

**Listing 6-4. Calling Namespace Members: NamespaceCall.cs**

// Namespace Declaration  
using System;  
  
namespace csharp\_station   
{  
    // nested namespace  
    namespace tutorial   
    {  
        class myExample1   
        {  
            publicstaticvoid myPrint1()   
            {  
                Console.WriteLine("First Example of calling another namespace member.");  
            }  
        }  
    }  
  
    // Program start class  
    class NamespaceCalling   
    {  
        // Main begins program execution.  
        publicstaticvoid Main()   
        {  
            // Write to console  
            tutorial.myExample1.myPrint1();   
            tutorial.myExample2.myPrint2();   
        }  
    }  
}  
  
// same namespace as nested namespace above  
namespace csharp\_station.tutorial   
{  
    class myExample2   
    {  
        publicstaticvoid myPrint2()   
        {  
            Console.WriteLine("Second Example of calling another namespace member.");  
        }  
    }  
}

Listing 6-4 provides an example of how to call *namespace* members with fully qualified names. A fully qualified name contains every language element from the *namespace* name down to the method call. At the top of the listing there is a nested *namespacetutorial* within the *csharp-stationnamespace* with *classmyExample1* and method *myPrint1*. *Main()* calls this method with the fully qualified name of *tutorial.myExample1.myPrint1()*. Since *Main()* and the *tutorialnamespace* are located in the same *namespace*, using *csharp\_station* in the fully qualified name is unnecessary.

At the bottom of Listing 6-4 is an addition to the *csharp\_station.tutorialnamespace*. The classes *myExample1* and *myExample2* both belong to the same *namespace*. Additionally, they could be written in separate files and still belong to the same *namespace*. In *Main()*, the *myPrint2()* method is called with the fully qualified name *tutorial.myExample2.myPrint2()*. Although the class *myExample2* is outside the bounding braces of where the method *myPrint2* is called, the *namespacecsharp\_station* does not need to be a part of the fully qualified name. This is because both classes belong to the same *namespace*, *csharp\_station*.

Notice that I used different names for the two classes *myExample1* and *myExample2*. This was necessary because every *namespace* member of the same type must have a unique name. Remember, they are both in the same *namespace* and you wouldn't want any ambiguity about which class to use. The methods *myPrint1()* and *myPrint2()* have different names only because it would make the lesson a little easier to follow. They could have had the same name with no effect, because their classes are different, thus avoiding any ambiguity.

**Listing 6-5. The using Directive: UsingDirective.cs**

// Namespace Declaration  
using System;  
using csharp\_station.tutorial;  
  
// Program start class  
class UsingDirective   
{  
    // Main begins program execution.  
    publicstaticvoid Main()   
    {  
        // Call namespace member  
        myExample.myPrint();   
    }  
}  
  
// C# Station Tutorial Namespace  
namespace csharp\_station.tutorial   
{  
    class myExample   
    {  
        publicstaticvoid myPrint()   
        {  
            Console.WriteLine("Example of using a using directive.");  
        }  
    }  
}

If you would like to call methods without typing their fully qualified name, you can implement the *using* directive. In Listing 6-5, we show two *using* directives. The first, *using System*, is the same *using* directive you have seen in every program in this tutorial. It allows you to type the method names of members of the *Systemnamespace* without typing the word *System* every time. In *myPrint()*, *Console* is a class member of the *Systemnamespace* with the method *WriteLine()*. Its fully qualified name is *System.Console.WriteLine(...)*.

Similarly, the *using* directive *using csharp\_station.tutorial* allows us to call members of the *csharp\_station.tutorialnamespace* without typing the fully qualified name. This is why we can type *myExample.myPrint()*. Without the *using* directive, we would have to type *csharp\_station.tutorial.myExample.myPrint()* every time we wanted to call that method.

**Listing 6-6. The Alias Directive: AliasDirective.cs**

// Namespace Declaration  
using System;  
using csTut = csharp\_station.tutorial.myExample; // alias  
  
// Program start class  
class AliasDirective   
{  
    // Main begins program execution.  
    publicstaticvoid Main()   
    {  
        // Call namespace member  
        csTut.myPrint();  
        myPrint();  
    }  
  
    // Potentially ambiguous method.  
    staticvoid myPrint()   
    {  
        Console.WriteLine("Not a member of csharp\_station.tutorial.myExample.");  
    }  
}  
  
// C# Station Tutorial Namespace  
namespace csharp\_station.tutorial   
{  
    class myExample   
    {  
        publicstaticvoid myPrint()   
        {  
            Console.WriteLine("This is a member of csharp\_station.tutorial.myExample.");  
        }  
    }  
}

Sometimes you may encounter a long namespace and wish to have it shorter. This could improve readability and still avoid name clashes with similarly named methods. Listing 6-6 shows how to create an alias with the alias directive *using csTut = csharp\_station.tutorial.myExample*. Now the expression *csTut* can be used anywhere, in this file, in place of *csharp\_station.tutorial.myExample*. We use it in *Main()*.

Also in *Main()* is a call to the *myPrint()* method of the *AliasDirectiveclass*. This is the same name as the *myPrint()* method in the *myExampleclass* . The reason both of these methods can be called in the same method call is because the *myPrint()* method in the *myExampleclass* is qualified with the *csTut* alias. This lets the compiler know exactly which method is to be executed. Had we mistakenly omitted *csTut* from the method call, the compiler would have set up the *myPrint()* method of the *AliasDirectiveclass* to run twice.

So far, all we've shown in our namespaces are classes. However, namespaces can hold other types as follows:

* Classes
* Structures
* Interfaces
* Enumerations
* Delegates

Future chapters will cover what these types are in more detail.

In summary, you know what a *namespace* is and you can declare your own namespaces. If you don't want to type a fully qualified name, you know how to implement the *using* directive. When you want to shorten a long namespace declaration, you can use the alias directive. Also, you have been introduced to some of the other *namespace* members in addition to the *class* type.

**Introduction to Classes**

This lesson introduces you to C# Classes. Our objectives are as follows:

* Implement Constructors.
* Know the difference between instance and *static* members.
* Understand Destructors.
* Familiarization with Class Members.

Since the beginning of this tutorial, you have been using classes. By now, you should have a sense of what a *class* is for and how to specify one. This lesson will build upon what you already know and introduce the various *class* members.

Classes are declared by using the keyword *class* followed by the *class* name and a set of *class* members surrounded by curly braces. Every *class* has a constructor, which is called automatically any time an instance of a *class* is created. The purpose of constructors is to initialize *class* members when an instance of the *class* is created. Constructors do not have return values and always have the same name as the *class*. Listing 7-1 is an example of a *class*.

**Listing 7-1. Example C# Classes: Classes.cs**

// Namespace Declaration  
using System;  
  
// helper class  
class OutputClass   
{  
    string myString;  
  
    // Constructor  
    public OutputClass(string inputString)   
    {  
        myString = inputString;  
    }  
  
    // Instance Method  
    publicvoid printString()   
    {  
        Console.WriteLine("{0}", myString);  
    }  
  
    // Destructor  
    ~OutputClass()   
    {  
        // Some resource cleanup routines  
    }  
}  
  
// Program start class  
class ExampleClass   
{  
    // Main begins program execution.  
    publicstaticvoid Main()   
    {  
        // Instance of OutputClass  
        OutputClass outCl = new OutputClass("This is printed by the output class.");  
  
        // Call Output class' method  
        outCl.printString();   
    }  
}  
<%--

[Get Setup Instructions For How to Run this Program](http://www.csharp-station.com/Tutorial/CSharp/SmartConsoleSetup.aspx)

--%>

Listing 7-1 shows two classes. The top *class*, *OutputClass*, has a constructor, instance method, and a destructor. It also had a field named *myString*. Notice how the *OutputClass* constructor is used to initialize data members of the *class*. In this case, the *OutputClass* constructor accepts a *string* argument, *inputString*. This *string* is copied to the *class* field *myString*.

Constructors are not mandatory, as indicated by the implementation of *ExampleClass*. In this case, a default constructor is provided. A default constructor is simply a constructor with no arguments. However, a constructor with no arguments is not always useful. To make default constructors more useful, you can implement them with initializers. Here is an example:

    public OutputClass() : this("Default Constructor String") { }

Imagine this constructor was included in *classOutputClass* from Listing 7-1. This default constructor is followed by an initializer. The colon, ":", marks the beginning of the initializer, followed by the *this* keyword. The *this* keyword refers to this particular object. It effectively makes a call to the constructor of the same object it is defined in. After the *this* keyword is a parameter list with a *string*. The action taken by the initializer above is to invoke the *OutputClass* constructor that takes a *string* type as an argument. The initializer helps you to ensure your *class* fields are initialized when a *class* is instantiated.

The example above illustrates how a class can have multiple constructors. The specific constructor called depends on the number of parameters and the type of each parameter.

In C#, there are two types of *class* members, instance and *static*. Instance *class* members belong to a specific occurrence of a *class*. Every time you declare an object of a certain *class*, you create a new instance of that *class*. The *ExampleClassMain()* method creates an instance of the *OutputClass* named *outCl*. You can create multiple instances of *OutputClass* with different names. Each of these instances are separate and stand alone. For example, if you create two *OutputClass* instances as follows:

    OutputClass oc1 = new OutputClass("OutputClass1");  
    OutputClass oc2 = new OutputClass("OutputClass2");

You create two separate instances of *OutputClass* with separate *myString* fields and separate *printString()* methods. On the other hand, if a *class* member is *static*, you can access it simply by using the syntax *<classname>.<static class member>*. The instance names are *oc1* and *oc2*.

Suppose *OutputClass* had the following *static* method:

    publicstaticvoid staticPrinter()   
    {  
        Console.WriteLine("There is only one of me.");  
    }

Then you could call that function from *Main()* like this:

OutputClass.staticPrinter();

You must call *static* class members through their *class* name and not their instance name. This means that you don't need to instantiate a class to use its *static* members. There is only ever one copy of a *static* class member. A good use of *static* members is when there is a function to be performed and no intermediate state is required, such as math calculations. Matter of fact, the .NET Frameworks Base Class Library includes a *Mathclass* that makes extensive use of *static* members.

Another type of constructor is the *static* constructor. Use *static* constructor to initialize *static* fields in a *class*. You declare a *static* constructor by using the keyword *static* just in front of the constructor name. A *static* constructor is called before an instance of a *class* is created, before a *static* member is called, and before the *static* constructor of a derived class (covered in a later chapter). They are called only once.

*OutputClass* also has a destructor. Destructors look just like constructors, except they have a tilde, "~", in front of them. They don't take any parameters and do not return a value. Destructors are places where you could put code to release any resources your class was holding during its lifetime. They are normally called when the C# garbage collector decides to clean your object from memory.

**Note:** You've probably noticed the use of the *public* modifier (an access modifier), meaning that a class member can be accessed from other classes. When used on a class, it means that the class can be accessed by DLLs outside of the Assembly (which is commonly a \*.exe or \*.dll file). [Lesson 19: Encapsulation](http://www.csharp-station.com/Tutorial/CSharp/Lesson19) discusses access modifiers in more depth.

So far, the only class members you've seen are Fields, Methods, Constructors, and Destructors. Here is a complete list of the types of members you can have in your classes:

* Constructors
* Destructors
* Fields
* Methods
* Properties
* Indexers
* Delegates
* **Eve nts**
* Nested Classes

Those items not covered in this lesson will be covered in later lessons.

In summary, you can declare instance and *static* constructors. You know how to initialize class fields. When there is no need to instantiate an object, you can create *static* class members. You can also declare destructors for cleaning up resources.

**Class Inheritance**

This lesson teaches about C# Inheritance. Our objectives are as follows:

* Implement Base Classes.
* Implement Derived Classes.
* Initialize Base Classes from Derived Classes.
* Learn How to Call Base Class Members.
* Learn How to Hide Base Class Members.

Inheritance is one of the primary concepts of object-oriented programming. It allows you to reuse existing code. Through effective employment of reuse, you can save time in your programming.

**Listing 8-1. Inheritance: BaseClass.cs**

using System;  
  
publicclass ParentClass  
{  
    public ParentClass()  
    {  
        Console.WriteLine("Parent Constructor.");  
    }  
  
    publicvoid print()  
    {  
        Console.WriteLine("I'm a Parent Class.");  
    }  
}  
  
publicclass ChildClass : ParentClass  
{  
    public ChildClass()  
    {  
        Console.WriteLine("Child Constructor.");  
    }  
  
    publicstaticvoid Main()  
    {  
        ChildClass child = new ChildClass();  
  
        child.print();  
    }  
}

Output:

Parent Constructor.

Child Constructor.

I'm a Parent Class.

Listing 8-1 shows two classes. The top class is named *ParentClass* and the main class is called *ChildClass*. What we want to do is create a child class, using existing code from *ParentClass*.

First we must declare our intention to use *ParentClass* as the base class of *ChildClass*. This is accomplished through the *ChildClass* declaration public class *ChildClass : ParentClass*. The base class is specified by adding a colon, ":", after the derived class identifier and then specifying the base class name.

**Note:** C# supports single class inheritance only. Therefore, you can specify only one base class to inherit from. However, it does allow multiple *interface* inheritance, a subject covered in a later lesson.

*ChildClass* has exactly the same capabilities as *ParentClass*. Because of this, you can also say *ChildClass* "is" a *ParentClass*. This is shown in the *Main()* method of *ChildClass* when the *print()* method is called. *ChildClass* does not have its own *print()* method, so it uses the *ParentClassprint()* method. You can see the results in the 3rd line of output.

Base classes are automatically instantiated before derived classes. Notice the output from Listing 8-1. The *ParentClass* constructor executed before the *ChildClass* constructor.

**Listing 8-2. Derived Class Communicating with Base Class: BaseTalk.cs**

using System;

publicclass Parent

{

    string parentString;

    public Parent()

    {

        Console.WriteLine("Parent Constructor.");

    }

    public Parent(string myString)

    {

        parentString = myString;

        Console.WriteLine(parentString);

    }

    publicvoid print()

    {

        Console.WriteLine("I'm a Parent Class.");

    }

}

public class Child : Parent

{

    public Child() : base("From Derived")

    {

        Console.WriteLine("Child Constructor.");

    }

    publicnewvoid print()

    {

        base.print();

        Console.WriteLine("I'm a Child Class.");

    }

    publicstaticvoid Main()

    {

        Child child = new Child();

        child.print();

        ((Parent)child).print();

    }

}

Output:

From Derived

Child Constructor.

I'm a Parent Class.

I'm a Child Class.

I'm a Parent Class.

Derived classes can communicate with base classes during instantiation. Listing 8-2 shows how this is done at the child constructor declaration. The colon, ":", and keyword *base* call the base class constructor with the matching parameter list. If the code had not appended *base("From Derived")* to the *Derived* constructor, the code would have automatically called *Parent()*. The first line of output shows the base class constructor being called with the *string* "From Derived".

Sometimes you may want to create your own implementation of a method that exists in a base class. The *Child* class does this by declaring its own *print()* method. The *Childprint()* method hides the *Parentprint()* method. The effect is the *Parentprint()* method will not be called, unless we do something special to make sure it is called.

Inside the *Childprint()* method, we explicitly call the *Parentprint()* method. This is done by prefixing the method name with "*base."*. Using the *base* keyword, you can access any of a base class *public* or *protected* class members. The output from the *Childprint()* method is on output lines 3 and 4.

Another way to access base class members is through an explicit cast. This is done in the last statement of the *Child* class *Main()* method. Remember that a derived class is a specialization of its base class. This fact allows us to perform a cast on the derived class, making it an instance of its base class. The last line of output from Listing 8-2 shows the *Parentprint()* method was indeed executed.

Notice the *new* modifier on the *Child* class *print()* method. This enables this method to hide the *Parent* class *print()* method and explicitly states your intention that you don't want polymorphism to occur. Without the *new* modifier, the compiler will produce a warning to draw your attention to this. See the next lesson for a detailed discussion of polymorphism.

In summary, you know how to create a derived/base class relationship. You can control instantiation of your base class and call its methods either implicitly or explicitly. You also understand that a derived class is a specialization of its base class.

**Polymorphism**

This lesson teaches about Polymorphism in C#. Our objectives are as follows:

* Learn What Polymorphism Is.
* Implement a Virtual Method.
* Override a Virtual Method.
* Use Polymorphism in a Program.

Another primary concept of object-oriented programming is Polymorphism. It allows you to invoke derived class methods through a base class reference during run-time. This is handy when you need to assign a group of objects to an array and then invoke each of their methods. They won't necessarily have to be the same object type. However, if they're related by inheritance, you can add them to the array as the inherited type. Then if they all share the same method name, that method of each object can be invoked. This lesson will show you how to accomplish this.

**Listing 9-1. A Base Class With a Virtual Method: DrawingObject.cs**

using System;  
  
publicclass DrawingObject  
{  
   publicvirtualvoid Draw()  
    {  
        Console.WriteLine("I'm just a generic drawing object.");  
    }  
}

Listing 9-1 shows the *DrawingObject* class. This will be the base class for other objects to inherit from. It has a single method named *Draw()*. The *Draw()* method has a *virtual* modifier. The *virtual* modifier indicates to derived classes that they can override this method. The *Draw()* method of the *DrawingObject* class performs a single action of printing the statement, "I'm just a generic drawing object.", to the console.

**Listing 9-2. Derived Classes With Override Methods: Line.cs, Circle.cs, and Square.cs**

using System;  
  
publicclass Line : DrawingObject  
{  
   publicoverridevoid Draw()  
    {  
        Console.WriteLine("I'm a Line.");  
    }  
}  
  
publicclass Circle : DrawingObject  
{  
   publicoverridevoid Draw()  
    {  
        Console.WriteLine("I'm a Circle.");  
    }  
}  
  
publicclass Square : DrawingObject  
{  
   publicoverridevoid Draw()  
    {  
        Console.WriteLine("I'm a Square.");  
    }  
}

Listing 9-2 shows three classes. These classes inherit the *DrawingObject* class. Each class has a *Draw()* method and each *Draw()* method has an *override* modifier. The *override* modifier allows a method to *override* the *virtual* method of its base class at run-time. The *override* will happen only if the class is referenced through a base class reference. Overriding methods must have the same signature, name and parameters, as the *virtual* base class method it is overriding.

**Listing 9-3. Program Implementing Polymorphism: DrawDemo.cs**

using System;  
  
publicclass DrawDemo  
{  
   publicstaticint Main( )  
    {  
        DrawingObject[] dObj = new DrawingObject[4];  
  
        dObj[0] = new Line();  
        dObj[1] = new Circle();  
        dObj[2] = new Square();  
        dObj[3] = new DrawingObject();  
  
        foreach (DrawingObject drawObj in dObj)  
        {  
            drawObj.Draw();  
        }  
  
        return 0;  
    }  
}

Listing 9-3 shows a program that uses the classes defined in Listing 9-1 and Listing 9-2. This program implements polymorphism. In the *Main()* method of the *DrawDemo* class, there is an array being created. The type of object in this array is the *DrawingObject* class. The array is named *dObj* and is being initialized to hold four objects of type *DrawingObject*.

Next the *dObj* array is initialized. Because of their inheritance relationship with the *DrawingObject* class, the *Line*, *Circle*, and *Square* classes can be assigned to the *dObj* array. Without this capability, you would have to create an array for each type. Inheritance allows derived objects to act like their base class, which saves work.

After the array is initialized, there is a *foreach* loop that looks at each element of the array. Within the *foreach* loop the *Draw()* method is invoked on each element of the *dObj* array. Because of polymorphism, the run-time type of each object is invoked. The type of the reference object from the *dObj* array is a *DrawingObject*. However, that doesn't matter because the derived classes *override* the *virtualDraw()* method of the *DrawingObject* class. This makes the overriden *Draw()* methods of the derived classes execute when the *Draw()* method is called using the *DrawingObject* base class reference from the *dObj* array. Here's what the output looks like:

Output:

I'm a Line.

I'm a Circle.

I'm a Square.

I'm just a generic drawing object.

The *overrideDraw()* method of each derived class executes as shown in the *DrawDemo* program. The last line is from the *virtualDraw()* method of the *DrawingObject* class. This is because the actual run-time type of the fourth array element was a *DrawingObject* object.

The code in this lesson can be compiled with the following command line:

csc DrawDemo.cs DrawingObject.cs Circle.cs Line.cs Square.cs

It will create the file *DrawDemo.exe*, which defaulted to the name of the first file on the command line.

**Summary**

You should now have a basic understanding of polymorphism. You know how to define a *virtual* method. You can implement a derived class method that overrides a *virtual* method. This relationship between *virtual* methods and the derived class methods that *override* them enables polymorphism. This lesson showed how to use this relationship between classes to implement polymorphism in a program.

**Properties**

This lesson teaches C# Properties. Our objectives are as follows:

* Understand What Properties Are For.
* Implement a Property.
* Create a Read-Only Property.
* Create a Write-Only Property.
* Create an auto-implemented property.

**Overview of Properties**

Properties provide the opportunity to protect a field in a class by reading and writing to it through the property. In other languages, this is often accomplished by programs implementing specialized getter and setter methods. C# properties enable this type of protection while also letting you access the property just like it was a field.

Another benefit of properties over fields is that you can change their internal implementation over time. With a public field, the underlying data type must always be the same because calling code depends on the field being the same. However, with a property, you  can change the implementation. For example, if a customer has an ID that is originally stored as an int, you might have a requirements change that made you perform a validation to ensure that calling code could never set the ID to a negative value. If it was a field, you would never be able to do this, but a property allows you to make such a change without breaking code. Now, lets see how to use properties.

**Traditional Encapsulation Without Properties**

Languages that don't have properties will use methods (functions or procedures) for encapsulation. The idea is to manage the values inside of the object, state, avoiding corruption and misuse by calling code. Listing 10-1 demonstrates how this traditional method works, encapsulating *Customer* information via accessor methods.

**Listing 10-1. An Example of Traditional Class Field Access**

using System;

publicclass Customer

{

privateint m\_id = -1;

publicint GetID()

{

return m\_id;

}

publicvoid SetID(int id)

{

m\_id = id;

}

privatestring m\_name = string.Empty;

publicstring GetName()

{

return m\_name;

}

publicvoid SetName(string name)

{

m\_name = name;

}

}

publicclass CustomerManagerWithAccessorMethods

{

publicstaticvoid Main()

{

Customer cust = new Customer();

cust.SetID(1);

cust.SetName("Amelio Rosales");

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.GetID(),

cust.GetName());

Console.ReadKey();

}

}

Listing 10-1 shows the traditional method of accessing class fields. The *Customer* class has four methods, two for each private field that the class encapsulates: *m\_id* and *m\_name*. As you can see, *SetID* and *SetName* assign a new values and *GetID* and *GetName* return values.

Observe how *Main* calls the *SetXxx* methods, which sets *m\_id* to *1* and *m\_name* to "Amelio Rosales" in the *Customer* instance, *cust*.  The call to *Console.WriteLine* demonstrates how to read *m\_id* and *m\_name* from *cust*, via *GetID* and *GetName* method calls, respectively.

This is such a common pattern, that C# has embraced it in the form of a language feature called properties, which you'll see in the next section.

**Encapsulating Type State with Properties**

The practice of accessing field data via methods was good because it supported the object-oriented concept of encapsulation. For example, if the type of *m\_id* or *m\_name* changed from an *int* type to *byte*, calling code would still work. Now the same thing can be accomplished in a much smoother fashion with properties, as shown in Listing 10-2.

**Listing 10-2. Accessing Class Fields With Properties**

using System;

publicclass Customer

{

privateint m\_id = -1;

publicint ID

{

get

{

return m\_id;

}

set

{

m\_id = value;

}

}

privatestring m\_name = string.Empty;

publicstring Name

{

get

{

return m\_name;

}

set

{

m\_name = value;

}

}

}

publicclass CustomerManagerWithProperties

{

publicstaticvoid Main()

{

Customer cust = new Customer();

cust.ID = 1;

cust.Name = "Amelio Rosales";

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.ID,

cust.Name);

Console.ReadKey();

}

}

Listing 10-2 shows how to create and use a property. The *Customer* class has the *ID* and *Name* property implementations. There are also private fields named *m\_id* and *m\_name;* which *ID* and *Name*, respectively, encapsulate. Each property has two accessors, *get* and *set*. The *get* accessor returns the value of a field. The *set* accessor sets the value of a field with the contents of *value*, which is the value being assigned by calling code. The *value* shown in the accessor is a C# reserved word.

When setting a property, just assign a value to the property as if it were a field. The *CustomerManagerWithProperties* class uses the *ID* and *Name* properties in the *Customer* class. The first line of *Main* instantiates a *Customer* object named *cust*. Next the value of the *m\_id* and *m\_name* fields of *cust* are set by using the *ID* and *Name* properties.

To read from a property, use the property as if it were a field. *Console.WriteLine* prints the value of the *m\_id* and *m\_name* fields of *cust*. It does this by calling the *ID* and *Name* properties of *cust*.

This was a read/write property, but you can also create read-only properties, which you'll learn about next.

**Creating Read-Only Properties**

Properties can be made read-only. This is accomplished by having only a *get* accessor in the property implementation. Listing 10-3 demonstrates how you can create a read-only property.

**Listing 10-3. Read-Only Properties**

using System;

publicclass Customer

{

privateint m\_id = -1;

privatestring m\_name = string.Empty;

public Customer(int id, string name)

{

m\_id = id;

m\_name = name;

}

publicint ID

{

get

{

return m\_id;

}

}

publicstring Name

{

get

{

return m\_name;

}

}

}

publicclass ReadOnlyCustomerManager

{

publicstaticvoid Main()

{

Customer cust = new Customer(1, "Amelio Rosales");

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.ID,

cust.Name);

Console.ReadKey();

}

}

The *Customer* class in Listing 10-3 has two read-only properties, *ID* and *Name*. You can tell that each property is read-only because they only have *get* accessors. At some time, values for the *m\_id* and *m\_name* must be assigned, which is the role of the constructor in this example.

The *Main* method of the *ReadOnlyCustomerManager* class instantiates a new *Customer* object named *cust*. The instantiation of *cust* uses the constructor of *Customer* class, which takes *int* and *string* type parameters. In this case, the values are *1* and *"Amelio Rosales"*. This initializes the *m\_id* and *m\_name* fields of *cust*.

Since the *ID* and *Name* properties of the *Customer* class are read-only, there is no other way to set the value of the *m\_id* and *m\_name* fields. If you inserted *cust.ID = 7* into the listing, the program would not compile, because *ID* is read-only; the same goes for *Name*. When the *ID* and *Name* properties are used in *Console.WriteLine*, they work fine. This is because these are read operations which only invoke the *get* accessor of the *ID* and *Name* properties.

One question you might have now is "If a property can be read-only, can it also be write-only?" The answer is yes, and explained in the next section.

**Creating a Write-Only Property**

You can assign values to, but not read from, a write-only property. A write-only property only has a *set* accessor. Listing 10-4 shows you how to create and use write-only properties.

**Listing 10-4. Write-Only Properties**

using System;

publicclass Customer

{

privateint m\_id = -1;

publicint ID

{

set

{

m\_id = value;

}

}

privatestring m\_name = string.Empty;

publicstring Name

{

set

{

m\_name = value;

}

}

publicvoid DisplayCustomerData()

{

Console.WriteLine("ID: {0}, Name:

{1}", m\_id, m\_name);

}

}

publicclass WriteOnlyCustomerManager

{

publicstaticvoid Main()

{

Customer cust = new Customer();

cust.ID = 1;

cust.Name = "Amelio Rosales";

cust.DisplayCustomerData();

Console.ReadKey();

}

}

This time, the *get* accessor is removed from the *ID* and *Name* properties of the *Customer* class, shown in Listing 10-1. The *set* accessors have been added, assigning *value* to the backing store fields, *m\_id* and *m\_name*.

The *Main* method of the *WriteOnlyCustomerManager* class instantiates the *Customer* class with a default constructor. Then it uses the *ID* and *Name* properties of *cust* to set the *m\_id* and *m\_name* fields of *cust* to *1* and *"Amelio Rosales"*, respectively. This invokes the *set* accessor of *ID* and *Name* properties from the *cust* instance.

When you have a lot of properties in a class or struct, there can also be a lot of code associated with those properties. In the next section, you'll see how to write properties with less code.

**Creating Auto-Implemented Properties**

The patterns you see here, where a property encapsulates a property with *get* and *set* accessors, without any other logic is common. It is more code than we should have to write for such a common scenario. That's why C# 3.0 introduced a new syntax for a property, called an *auto-implemented property*, which allows you to create properties without *get* and *set* accessor implementations. Listing 10-5 shows how to add auto-implemented properties to a class.

**Listing 10-5. Auto-Implemented Properties**

using System;

publicclass Customer

{

publicint ID { get; set; }

publicstring Name { get; set; }

}

publicclass AutoImplementedCustomerManager

{

staticvoid Main()

{

Customer cust = new Customer();

cust.ID = 1;

cust.Name = "Amelio Rosales";

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.ID,

cust.Name);

Console.ReadKey();

}

}

Notice how the *get* and *set* accessors in Listing 10-5 do not have implementations. In an auto-implemented property, the C# compiler creates the backing store field behind the scenes, giving the same logic that exists with traditional properties, but saving you from having to use all of the syntax of the traditional property. As you can see in the *Main* method, the usage of an auto-implemented property is exactly the same as traditional properties, which you learned about in previous sections.

**Summary**

You now know what properties are for and how they're used. Traditional techniques of encapsulation have relied on separate methods. Properties allow you to access objects state with field-like syntax. Properties can be made read-only or write-only. You also learned how to write properties with less code by using auto-implemented properties.

## Interfaces

This lesson teaches C# *Interfaces*. Our objectives are as follows:

* Understand the Purpose of *Interfaces.*
* Define an *Interface.*
* Use an *Interface.*
* Implement Interface *Inheritance*.

An *interface* looks like a class, but has no implementation. The only thing it contains are declarations of *events*, *indexers*, *methods* and/or *properties*. The reason *interfaces* only provide declarations is because they are inherited by *classes* and *structs*, which must provide an implementation for each interface member declared.

So, what are *interfaces* good for if they don't implement functionality? They're great for putting together plug-n-play like architectures where components can be interchanged at will. Since all interchangeable components implement the same *interface*, they can be used without any extra programming. The *interface* forces each component to expose specific public members that will be used in a certain way.

Because *interfaces* must be implemented by derived *classes* and *structs*, they define a contract. For instance, if class *foo* implements the *IDisposable* interface, it is making a statement that it guarantees it has the *Dispose()* method, which is the only member of the *IDisposable* interface. Any code that wishes to use class *foo* may check to see if class *foo*implements *IDisposable*. When the answer is *true*, then the code knows that it can call *foo.Dispose()*. Listing 13-1 shows how to define an interface:

##### Listing 13-1. Defining an Interface: MyInterface.cs

interface IMyInterface  
{  
    void MethodToImplement();  
}

Listing 13-1 defines an *interface* named *IMyInterface*. A common naming convention is to prefix all *interface* names with a capital "I". This *interface* has a single method named *MethodToImplement()*. This could have been any type of method declaration with different parameters and return types. I just chose to declare this method with no parameters and a *void* return type to make the example easy. Notice that this method does not have an implementation (instructions between curly braces - *{}*), but instead ends with a semi-colon, "*;*". This is because the *interface* only specifies the signature of methods that an inheriting *class* or *struct* must implement. Listing 13-2 shows how this *interface* could be used.

##### Listing 13-2. Using an Interface: InterfaceImplementer.cs

class InterfaceImplementer : IMyInterface  
{  
    staticvoid Main()  
    {  
        InterfaceImplementer iImp = new InterfaceImplementer();  
        iImp.MethodToImplement();  
    }  
  
    publicvoid MethodToImplement()  
    {  
        Console.WriteLine("MethodToImplement() called.");  
    }  
}

The *InterfaceImplementerclass* in Listing 13.2 implements the *IMyInterfaceinterface*. Indicating that a *class* inherits an *interface* is the same as inheriting a *class*. In this case, the following syntax is used:

class InterfaceImplementer : IMyInterface

Now that this *class* inherits the *IMyInterface* interface, it must implement its members. It does this by implementing the *MethodToImplement()* method. Notice that this method implementation has the exact same signature, parameters and method name, as defined in the *IMyInterface* interface. Any difference between the method signature in the interface and the method signature in the implementing *class* or *struct* will cause a compiler error. Additionally, a *class* or *struct* that inherits an interface must include all interface members; You will receive a compiler error if you don't implement all interface members.

Interfaces may also inherit other interfaces. Listing 13-3 shows how inherited interfaces are implemented.

##### Listing 13-3. Interface Inheritance: InterfaceInheritance.cs

using System;  
  
interface IParentInterface  
{  
    void ParentInterfaceMethod();  
}  
  
interface IMyInterface : IParentInterface  
{  
    void MethodToImplement();  
}  
  
class InterfaceImplementer : IMyInterface  
{  
    staticvoid Main()  
    {  
        InterfaceImplementer iImp = new InterfaceImplementer();  
        iImp.MethodToImplement();  
        iImp.ParentInterfaceMethod();  
    }  
  
    publicvoid MethodToImplement()  
    {  
        Console.WriteLine("MethodToImplement() called.");  
    }  
  
    publicvoid ParentInterfaceMethod()  
    {  
        Console.WriteLine("ParentInterfaceMethod() called.");  
    }  
}

The code in listing 13.3 contains two *interfaces*: *IMyInterface* and the *interface* it inherits, *IParentInterface*. When one *interface* inherits another, any implementing *class* or *struct* must implement every *interface* member in the entire inheritance chain. Since the *InterfaceImplementerclass* in Listing 13-3 inherits from *IMyInterface*, it also inherits *IParentInterface*. Therefore, the *InterfaceImplementer* class must implement the *MethodToImplement()* method specified in the *IMyInterface* interface and the *ParentInterfaceMethod()* method specified in the *IParentInterface* interface.

#### Summary

You now understand what *interfaces* are. You can implement an *interface* and use it in a class. *Interfaces* may also be inherited by other *interface*. Any *class* or *struct* that inherits an *interface* must also implement all members in the entire *interface* inheritance chain.

**Introduction to Delegates and Events**

This lesson introduces *delegates* and *events*. Our objectives are as follows:

* Understand What a *Delegate* Is
* Understand What an *Event* Is
* Implement *Delegates*
* Fire *Events*

**Delegates**

During previous lessons, you learned how to implement reference types using language constructs such as classes and interfaces. These reference types allowed you to create instances of objects and use them in special ways to accomplish your software development goals. Classes allow you to create objects that contained members with attributes or behavior. Interfaces allow you to declare a set of attributes and behavior that all objects implementing them would publicly expose. Today, I'm going to introduce a new reference type called a *delegate*.

A *delegate* is a C# language element that allows you to reference a method. If you were a C or C++ programmer, this would sound familiar because a *delegate* is basically a function pointer. However, developers who have used other languages are probably wondering, "Why do I need a reference to a method?". The answer boils down to giving you maximum flexibility to implement any functionality you want at runtime.

Think about how you use methods right now. You write an algorithm that does its thing by manipulating the values of variables and calling methods directly by name. What if you wanted an algorithm that was very flexible, reusable, and allowed you to implement different functionality as the need arises? Furthermore, let's say that this was an algorithm that supported some type of data structure that you wanted to have sorted, but you also want to enable this data structure to hold different types. If you don't know what the types are, how could you decide an appropriate comparison routine? Perhaps you could implement an *if/then/else* or *switch* statement to handle well-known types, but this would still be limiting and require overhead to determine the type. Another alternative would be for all the types to implement an interface that declared a common method your algorithm would call, which is actually a nice solution. However, since this lesson is about *delegates*, we'll apply a *delegate* solution, which is quite elegant.

You could solve this problem by passing a *delegate* to your algorithm and letting the contained method, which the *delegate* refers to, perform the comparison operation. Such an operation is performed in Listing 14-1.

**Listing 14-1. Declaring and Implementing a Delegate: SimpleDelegate.cs**

using System;  
  
// this is the delegate declaration  
publicdelegateint Comparer(object obj1, object obj2);  
  
publicclass Name  
{  
    publicstring FirstName = null;  
    publicstring LastName = null;  
  
    public Name(string first, string last)  
    {  
        FirstName = first;  
        LastName = last;  
    }  
  
    // this is the delegate method handler  
    publicstaticint CompareFirstNames(object name1, object name2)  
    {  
        string n1 = ((Name)name1).FirstName;  
        string n2 = ((Name)name2).FirstName;  
  
        if (String.Compare(n1, n2) > 0)  
        {  
            return 1;  
        }  
        elseif (String.Compare(n1, n2) < 0)  
        {  
            return -1;  
        }  
        else  
        {  
            return 0;  
        }  
    }  
  
    publicoverridestring ToString()  
    {  
        return FirstName + " " + LastName;  
    }  
}  
  
class SimpleDelegate  
{  
    Name[] names = new Name[5];  
  
    public SimpleDelegate()  
    {  
        names[0] = new Name("Joe", "Mayo");  
        names[1] = new Name("John", "Hancock");  
        names[2] = new Name("Jane", "Doe");  
        names[3] = new Name("John", "Doe");  
        names[4] = new Name("Jack", "Smith");  
    }  
  
    staticvoid Main(string[] args)  
    {  
        SimpleDelegate sd = new SimpleDelegate();  
  
        // this is the delegate instantiation  
        Comparer cmp = new Comparer(Name.CompareFirstNames);  
  
        Console.WriteLine("\nBefore Sort: \n");  
  
        sd.PrintNames();  
  
        // observe the delegate argument  
        sd.Sort(cmp);  
  
        Console.WriteLine("\nAfter Sort: \n");  
  
        sd.PrintNames();  
    }  
  
    // observe  the delegate parameter  
    publicvoid Sort(Comparer compare)  
    {  
        object temp;  
  
        for (int i=0; i < names.Length; i++)  
        {  
            for (int j=i; j < names.Length; j++)  
            {  
                // using delegate "compare" just like  
                // a normal method  
                if ( compare(names[i], names[j]) > 0 )  
                {  
                    temp = names[i];  
                    names[i] = names[j];  
                    names[j] = (Name)temp;  
                }  
            }  
        }  
    }  
  
    publicvoid PrintNames()  
    {  
        Console.WriteLine("Names: \n");  
  
        foreach (Name name in names)  
        {  
            Console.WriteLine(name.ToString());  
        }  
    }  
}

The first thing the program in Listing 14-1 does is declare a *delegate*. *Delegate* declarations look somewhat like methods, except they have the *delegate* modifier, are terminated with a semi-colon (*;*), and have no implementation. Below, is the *delegate* declaration from Listing 14-1.

publicdelegateint Comparer(object obj1, object obj2);

This *delegate* declaration defines the signature of a delegate handler method that this *delegate* can refer to. The delegate handler method, for the *Comparerdelegate*, can have any name, but must have a first parameter of type *object*, a second parameter of type *object*, and return an *int* type. The following method from Listing 14-1 shows a delegate handler method that conforms to the signature of the *Comparerdelegate*.

    publicstaticint CompareFirstNames(object name1, object name2)  
    {  
        ...  
    }

**Note:** The *CompareFirstNames* method calls *String.Compare* to compare the *FirstName* properties of the two *Name* instances. The *String* class has many convenience methods, such as *Compare*, for working with strings. Please don't allow the implementation of this method to interfere with learning how delegates work. What you should concentrate on is that *CompareFirstNames* is a handler method that a delegate can refer to, regardless of the code inside of that method.

To use a *delegate*, you must create an instance of it. The instance is created, similar to a class instance, with a single parameter identifying the appropriate delegate handler method, as shown below.

        Comparer cmp = new Comparer(Name.CompareFirstNames);

The *delegate*, *cmp*, is then used as a parameter to the *Sort()* method, which uses it just like a normal method. Observe the way the *delegate* is passed to the *Sort()* method as a parameter in the code below.

        sd.Sort(cmp);

Using this technique, any delegate handler method may be passed to the *Sort()* method at run-time. i.e. You could define a method handler named *CompareLastNames()*, instantiate a new *Comparer delegate* instance with it, and pass the new *delegate* to the *Sort()* method.

**Events**

Traditional Console applications operate by waiting for a user to press a key or type a command and press the *Enter* key. Then they perform some pre-defined operation and either quit or return to the original prompt that they started from. This works, but is inflexible in that everything is hard-wired and follows a rigid path of execution. In stark contrast, modern GUI programs operate on an event-based model. That is, some event in the system occurs and interested modules are notified so they can react appropriately. With Windows Forms, there is not a polling mechanism taking up resources and you don't have to code a loop that sits waiting for input. It is all built into the system with events.

A C# *event* is a class member that is activated whenever the event it was designed for occurs. I like to use the term "fires" when the *event* is activated. Anyone interested in the *event* can register and be notified as soon as the *event* fires. At the time an *event* fires, registered methods will be invoked.

*Events* and *delegates* work hand-in-hand to provide a program's functionality. It starts with a class that declares an *event*. Any class, including the same class that the *event* is declared in, may register one of its methods for the *event*. This occurs through a *delegate*, which specifies the signature of the method that is registered for the *event*. The *delegate* may be one of the pre-defined .NET *delegates* or one you declare yourself. Whichever is appropriate, you assign the *delegate* to the *event*, which effectively registers the method that will be called when the *event* fires. Listing 14-2 shows a couple different ways to implement *events*.

**Listing 14-2. Declaring and Implementing Events: Eventdemo.cs**

using System;  
using System.Drawing;  
using System.Windows.Forms;  
  
// custom delegate  
publicdelegatevoid Startdelegate();  
  
class Eventdemo : Form  
{  
    // custom event  
    publicevent Startdelegate StartEvent;  
  
    public Eventdemo()  
    {  
        Button clickMe = new Button();  
  
        clickMe.Parent = this;  
        clickMe.Text = "Click Me";  
        clickMe.Location = new Point(  
            (ClientSize.Width - clickMe.Width) /2,  
            (ClientSize.Height - clickMe.Height)/2);  
  
        // an EventHandler delegate is assigned  
        // to the button's Click event  
        clickMe.Click += new EventHandler(OnClickMeClicked);  
  
        // our custom "Startdelegate" delegate is assigned  
        // to our custom "StartEvent" event.  
        StartEvent += new Startdelegate(OnStartEvent);  
  
        // fire our custom event  
        StartEvent();  
    }  
  
    // this method is called when the "clickMe" button is pressed  
    publicvoid OnClickMeClicked(object sender, EventArgs ea)  
    {  
        MessageBox.Show("You Clicked My Button!");  
    }  
  
    // this method is called when the "StartEvent" Event is fired  
    publicvoid OnStartEvent()  
    {  
        MessageBox.Show("I Just Started!");  
    }  
  
    staticvoid Main(string[] args)  
    {  
        Application.Run(new Eventdemo());  
    }  
}

**Note:** If you're using Visual Studio or another IDE, remember to add references to System.Drawing.dll and System.Windows.Forms.dll before compiling Listing 14.2 or just add the code to a Windows Forms project. Teaching the operation of Visual Studio or other IDE's is out-of-scope for this tutorial.

You may have noticed that Listing 14-2 is a Windows Forms program. Although I haven't covered Windows Forms in this tutorial, you should know enough about C# programming in general that you won't be lost. To help out, I'll give a brief explanation of some of the parts that you may not be familiar with.

The *Eventdemo* class inherits *Form*, which essentially makes it a Windows Form. This automatically gives you all the functionality of a Windows Form, including Title Bar, Minimize/Maximize/Close buttons, System Menu, and Borders. A lot of power, that inheritance thing, eh?

The way a Windows Form's application is started is by calling the *Run()* method of the *staticApplication* object with a reference to the *form* object as its parameter. This starts up all the underlying Windows plumbing, displays the GUI, and ensures that *events* are fired as appropriate.

Let's look at the custom *event* first. Below is the *event* declaration, which is a member of the *Eventdemo* class. It is declared with the *event* keyword, a *delegate* type, and an *event* name.

    publicevent Startdelegate StartEvent;

Anyone interested in an *event* can register by hooking up a *delegate* for that *event*. On the next line, we have a *delegate* of type *Startdelegate*, which the *event* was declared to accept, hooked up to the *StartEventevent*. The *+=* syntax registers a *delegate* with an *event*. To unregister with an *event*, use the *-=* with the same syntax.

        StartEvent += new Startdelegate(OnStartEvent);

Firing an *event* looks just like a method call, as shown below:

StartEvent();

This was how to implement *events* from scratch, declaring the *event* and *delegate* yourself. However, much of the *event* programming you'll do will be with pre-defined *events* and *delegates*. This leads us to the other *event* code you see in Listing 14-2, where we hook up an *EventHandlerdelegate* to a *ButtonClick* event.

        clickMe.Click += new EventHandler(OnClickMeClicked);

The *Click* event already belongs to the *Button* class and all we have to do is reference it when registering a *delegate*. Similarly, the *EventHandlerdelegate* already exists in the *System* namespace of the .NET Frameworks Class Library. All you really need to do is define your callback method (delegate handler method) that is invoked when someone presses the *clickMe* button. The *OnClickMeClicked()* method, shown below, conforms to the signature of the *EventHandlerdelegate*, which you can look up in the .NET Framework Class Library reference.

    publicvoid OnClickMeClicked(object sender, EventArgs ea)  
    {  
        MessageBox.Show("You Clicked My Button!");  
    }

Any time the *clickMe* button is pressed with a mouse, it will fire the *Clickevent*, which will invoke the *OnClickMeClicked()* method. The *Button* class takes care of firing the *Clickevent* and there's nothing more you have to do. Because it is so easy to use pre-defined *events* and *delegates*, it would be a good idea to check if some exist already that will do what you need, before creating your own.

**Summary**

This completes this lesson, which was an introduction to *delegates* and *events*. You learned how to declare and implement *delegates*, which provide dynamic run-time method invocation services. You also know how to declare *events* and use them in a couple different scenarios. One way is to declare your own *event*, *delegate*, and callback method from scratch. Another way is to use pre-existing *events* and *delegates* and only implement the callback method, which will save you time and make coding easier.

**Introduction to Exception Handling**

This lesson teaches how to handle exceptions in your C# programs. Our objectives are as follows:

* Learn what an exception is
* Implement a routine with a *try/catch* block
* Release resources in a *finally* block

**Exceptions**

Exceptions are unforeseen errors that happen in your programs. Most of the time, you can, and should, detect and handle program errors in your code. For example, validating user input, checking for null objects, and verifying the values returned from methods are what you expect, are all examples of good standard error handling that you should be doing all the time.

However, there are times when you don't know if an error will occur. For example, you can't predict when you'll receive a file I/O error, run out of system memory, or encounter a database error. These things are generally unlikely, but they could still happen and you want to be able to deal with them when they do occur. This is where exception handling comes in.

When exceptions occur, they are said to be "thrown". What is actually thrown is an object that is derived from the *System.Exception* class. In the next section, I'll be explaining how thrown exceptions are handled with *try/catch* blocks.

The *System.Exception* class provides several methods and properties for obtaining information on what went wrong. For example, the *Message* property provides summary information about what the error was, the *Stacktrace* property provides information from the stack for where the problem occurred, and the *ToString()* method is overridden to reveal a verbose description of the entire exception.

Identifying the exceptions you'll need to handle depends on the routine you're writing. For example, if the routine opened a file with the *System.IO.File.OpenRead()* method, it could throw any of the following exceptions:

* *SecurityException*
* *ArgumentException*
* *ArgumentNullException*
* *PathTooLongException*
* *DirectoryNotFoundException*
* *UnauthorizedAccessException*
* *FileNotFoundException*
* *NotSupportedException*

It's easy to find out what exceptions a method can raise by looking in the .NET Frameworks SDK Documentation. Just go to the Reference/Class Library section and look in the Namespace/Class/Method documentation for the methods you use. The exception in the list above were found by looking at the *OpenRead()* method definition of the *File* class in the *System.IO* namespace. Each exception identified has a hyperlink to its class definition that you can use to find out what that exception is about. Once you've figured out what exceptions can be generated in your code, you need to put the mechanisms in place to handle the exceptions, should they occur.

**try/catch Blocks**

When exceptions are thrown, you need to be able to handle them. This is done by implementing a *try/catch* block. Code that could throw an exception is put in the *try* block and exception handling code goes in the *catch* block. Listing 15-1 shows how to implement a *try/catch* block. Since an *OpenRead()* method could throw one of several exceptions, it is placed in the *try* block. If an exception is thrown, it will be caught in the *catch* block. The code in Listing 15-1 will print message and stack trace information out to the console if an exception is raised.

**Note:** The programs in this lesson cause exceptions on purpose. The exception that you see is generated intentionally to show you what the exception message looks like before you see it yourself in your own programs.

**Listing 15-1. Using try/catch Blocks: tryCatchDemo.cs**

using System;  
using System.IO;  
  
class tryCatchDemo  
{  
    staticvoid Main(string[] args)  
    {  
        try  
        {  
            File.OpenRead("NonExistentFile");  
        }  
        catch(Exception ex)  
        {  
            Console.WriteLine(ex.ToString());  
        }  
    }  
}

Although the code in Listing 15-1 only has a single *catch* block, all exceptions will be caught there because the type is of the base exception type "Exception". In exception handling, more specific exceptions will be caught before their more general parent exceptions. For example, the following snippet shows how to place multiple catch blocks:

        catch(FileNotFoundException fnfex)  
        {  
            Console.WriteLine(fnfex.ToString());  
        }  
        catch(Exception ex)  
        {  
            Console.WriteLine(ex.ToString());  
        }

If the file doesn't exist, a *FileNotFoundException* exception will be thrown and caught by the first *catch* block. However, if a *PathTooLongException* exception was raised, the second catch part would catch the exception. This is because there isn't a *catch* block for the *PathTooLongException* exception and the generic *Exception* type *catch* block is the only option available to catch the exception.

Exceptions that are not handled will normally bubble up the stack until a calling routine in the call chain handles them. If you forget to include *try/catch* blocks in a part of your code and there aren't any *try/catch* blocks earlier in the call chain, your program will abort with a message describing the exception. To your users this would be very cryptic and uncomfortable. It is good practice to provide exception handling in your programs.

**Finally Blocks**

An exception can leave your program in an inconsistent state by not releasing resources or doing some other type of cleanup. A *catch* block is a good place to figure out what may have gone wrong and try to recover, however it can't account for all scenarios. Sometimes you need to perform clean up actions whether or not your program succeeds. These situations are good candidates for using a *finally* block.

Listing 15-2 illustrates the usefulness of a *finally* block. As you know, a file stream must be closed when you're done with it. In this case, the file stream is the resource that needs to be cleaned up. In Listing 15-2, *outStream* is opened successfully, meaning the program now has a handle to an open file resource. When trying to open the *inStream*, a *FileNotFoundException* exception is raised, causing control to go immediately to the *catch* block.

It's possible to close the *outStream* in the catch block, but what if the algorithm executed successfully without an exception? On success, the file would never be closed. Fortunately, we've included a *finally* block in Listing 15-2, which will always be executed. That's right, regardless of whether the algorithm in the *try* block raises an exception or not, the code in the *finally* block will be executed before control leaves the method.

**Listing 15-2. Implementing a finally Block: FinallyDemo.cs**

using System;  
using System.IO;  
  
class FinallyDemo  
{  
    staticvoid Main(string[] args)  
    {  
        FileStream outStream = null;  
        FileStream inStream = null;  
  
        try  
        {  
            outStream = File.OpenWrite("DestinationFile.txt");  
            inStream = File.OpenRead("BogusInputFile.txt");  
        }  
        catch(Exception ex)  
        {  
            Console.WriteLine(ex.ToString());  
        }  
        finally  
        {  
            if (outStream != null)  
            {  
                outStream.Close();  
                Console.WriteLine("outStream closed.");  
            }  
            if (inStream != null)  
            {  
                inStream.Close();  
                Console.WriteLine("inStream closed.");  
            }  
        }  
    }  
}

A finally block is not required and you may ask what happens if you just put code after the catch block. True, under normal circumstances, if the exception is caught, all code following the catch will be executed. However, try/catch/finally is for exceptional circumstances and it is better to plan for the worst to make your program more robust. For example, if one of the catch handlers rethrew an exception or caused another exception, the code following the catch block (not in a finally block) would never be executed. Also, if you don't catch the exception at all, program flow would immediately do a stack walk looking for an exception handler that fits and the code following the catch blocks would not be executed. Since there is too much potential for code in an algorithm to not be executed, a finally block is your insurance for executing those critical actions you need.

**Summary**

This has been an introduction to handling exceptions. By now, you should have a good understanding of what an exception is. You can implement algorithms within *try/catch* blocks that handle exceptions. Additionally, you know how to clean up resources by implementing a *finally* block whose code is always executed before leaving a method.

**Enums**

This lesson explains how to use C# enums. Our objectives are as follows:

* Understand what an enum is
* Be able to create new enum types
* Learn how to use enums
* Gain familiarity with System.Enum type methods

**Enums Defined**

Enums are strongly typed constants. They are essentially unique types that allow you to assign symbolic names to integral values. In the C# tradition, they are strongly typed, meaning that an enum of one type may not be implicitly assigned to an enum of another type even though the underlying value of their members are the same. Along the same lines, integral types and enums are not implicitly interchangable. All assignments between different enum types and integral types require an explicit cast.

Enums lend themselves to more maintainable code because they are symbolic, allowing you to work with integral values, but using a meaningful name to do so. For example, what type of code would you rather work with - a set of values named North, South, East, and West or the set of integers 0, 1, 2, and 3 that mapped to the same values, respectively? Enums make working with strongly typed constants via symbolic names easy.

Enums are value types, which means they contain their own value, can't inherit or be inherited from, and assignment copies the value of one enum to another. You will see in this lesson and elsewhere that enums are used and referred to with both lower case, *enum*, and upper case, *Enum*. The relationship between the two is that the C# type, *enum*, inherits the Base Class Library (BCL) type, *Enum*. Use the C# type, *enum*, to define new enums and use the BCL type, *Enum*, to implement static enum methods.

**Creating an Enum**

The .NET Framework Class Library contains many enums and examples of how they are used. For example, every time you put an icon on a *MessageBox*, you use the *MessageBoxIcon* enum. For a list of available enums in the .NET Framework Class Library, look at the documentation for the *Enum* class and click on the Derived Classes link.

Whenever there are situations where you are using a set of related numbers in a program, consider replacing those numbers with enums. It will make a program more readable and type safe. Listing 17-1 contains an *enum* definition and code that uses that enum in a *switch* statement. Instead of using the numbers 0, 1, and 2 in the *switch* statement, the code is more meaningful through the use of the *Volume* enum.

**Listing 17-1. Creating and Using an Enum: EnumSwitch.cs**

using System;  
  
// declares the enum  
publicenum Volume  
{  
   Low,  
   Medium,  
   High  
}  
  
// demonstrates how to use the enum  
  
class EnumSwitch  
{  
   staticvoid Main()  
   {  
      // create and initialize   
      // instance of enum type  
      Volume myVolume = Volume.Medium;  
  
      // make decision based  
      // on enum value  
      switch (myVolume)  
      {  
         case Volume.Low:  
            Console.WriteLine("The volume has been turned Down.");  
            break;  
         case Volume.Medium:  
            Console.WriteLine("The volume is in the middle.");  
            break;  
         case Volume.High:  
            Console.WriteLine("The volume has been turned up.");  
            break;  
      }  
      Console.ReadLine();  
   }  
}

Listing 17-1 contains a definition for an enum. Notice that it is declared with the *enum* keyword, has a type identifier (*Volume*), and contains a comma separated list of values enclosed within curly braces.

This enum is of type *Volume* and we use it to declare the *myVolume* variable in the *Main* method. Since an enum is a value type, we can assign a value (*Volume.Medium*) to it directly, similar to the simple types such as *int* or *double*. Once the *myVolume* variable is declared and initialized, it is used in the *switch* statement.Each of the *case* statements represent a unique member of the *Volume* enum.

Any time a member of the *Volume* enum is used, it is fully qualified with the "*Volume*" identifier to guarantee type safety. For example, if there were a *Meat* enum in scope, then *Meat.Medium* would definitely have different semantics than *Volume.Medium*. With both enums in scope, it would be ambiguous to just use the *Medium* identifier without type qualification. Using the type identifier ensures such mistakes are not made.

**Using Enums**

An enum is typically specified as shown in Listing 17-1, but may be customized by changing its base type and member values. By default, the underlying type of an enum is *int*. This default may be changed by specifying a specific base when declaring the enum. You would specify a different base if the enum was used extensively and there was an opportunity for space savings by selecting a smaller type. Another reason may be if you wanted the underlying type of the enum to correspond to another type in your program and you wanted to explicitly cast between the two without loss of precision. Valid base types include *byte*, *sbyte*, *short*, *ushort*, *int*, *uint*, *long*, and *ulong*.

Another modification you can make to an enum is to set the value of any enum member. By default, the first member of an enum takes the value of zero. If this value doesn't make sense for your enum, you can change it to one or some other number. Additionally, you can change any of the members of an enum to any value that is valid for its base type. Unassigned enum members have a value that is one more than their predecessor. Listing 17-2 shows how to modify the base type and member values of an enum.

**Listing 17-2. Setting the Enum Base and Initializing Members: EnumBaseAndMembers.cs**

using System;  
  
// declares the enum  
publicenum Volume : byte  
{  
    Low = 1,  
    Medium,  
    High  
}  
  
class EnumBaseAndMembers  
{  
    staticvoid Main()  
    {  
        // create and initialize   
        // instance of enum type  
        Volume myVolume = Volume.Low;  
  
        // make decision based  
        // on enum value  
        switch (myVolume)  
        {  
            case Volume.Low:  
                Console.WriteLine("The volume has been turned Down.");  
                break;  
            case Volume.Medium:  
                Console.WriteLine("The volume is in the middle.");  
                break;  
            case Volume.High:  
                Console.WriteLine("The volume has been turned up.");  
                break;  
        }  
        Console.ReadLine();  
    }  
}

The *Volume* enum in Listing 17-2 shows how to modify the base type and members of an enum.Its base type is changed to *byte* with the *: <type>* syntax following the enum identifier, *Volume*.This ensures that the *Volume* enum may only have members with values that are valid for type *byte*.

The first member of the *Volume* enum, *Low*, has its value changed to 1. The same syntax, *<member> = <value>*, may be applied to any member of the enum. You are restricted from creating forward references, circular references, and duplicate references in enum members.

The default values of the *Volume* enum are *Low*=0, *Medium*=1, and *High*=2 because the first member of an enum defaults to 0 and the following members default to one more than their predecessor. However, the *Volume* enum in Listing 17-2 has its *Low* member set to 1, which means that *Medium*=2 and *High*=3.

**Enum tricks**

Enum types implicitly inherit the System.Enum type in the Base Class Library (BCL). This also means that you can use the members of System.Enum to operate on enum types. This section does just that, showing some useful tips and tricks to use with enums in your programs.

A common requirement with enums is to convert between the enum and a variable of its base type. For example, if you are getting input in the form of an *int* from a user or a file stream, then you can cast it to an enum and use it in a meaningful way in your program. You can also get a complete list of enum member names or enum values, which is useful if you have logic that needs to iterate through every enum member. Listing 17-3 shows how to perform conversions between enums and their base types and how to use some of the System.Enum type members.

**Listing 17-3. Enum Conversions and using the System.Enum Type: Enumtricks.cs**

using System;  
  
// declares the enum  
publicenum Volume : byte  
{  
    Low = 1,  
    Medium,  
    High  
}  
  
// shows different ways  
// to work with enums  
class Enumtricks  
{  
    staticvoid Main(string[] args)  
    {  
        // instantiate type  
        Enumtricks enumtricks = new Enumtricks();  
  
        // demonstrates explicit cast  
        // of int to Volume  
        enumtricks.GetEnumFromUser();  
  
        // iterate through Volume enum by name  
        enumtricks.ListEnumMembersByName();  
  
        // iterate through Volume enum by value  
        enumtricks.ListEnumMembersByValue();  
  
        Console.ReadLine();  
    }  
  
    // demonstrates explicit cast  
    // of int to Volume  
    publicvoid GetEnumFromUser()  
    {  
        Console.WriteLine("\n----------------");  
        Console.WriteLine("Volume Settings:");  
        Console.WriteLine("----------------\n");  
  
        Console.Write(@"  
1 - Low  
2 - Medium  
3 - High  
  
Please select one (1, 2, or 3): ");  
  
        // get value user provided  
        string volString = Console.ReadLine();  
        int volInt = Int32.Parse(volString);  
  
        // perform explicit cast from  
        // int to Volume enum type  
        Volume myVolume = (Volume)volInt;  
  
        Console.WriteLine();  
  
        // make decision based  
        // on enum value  
        switch (myVolume)  
        {  
            case Volume.Low:  
                Console.WriteLine("The volume has been turned Down.");  
                break;  
            case Volume.Medium:  
                Console.WriteLine("The volume is in the middle.");  
                break;  
            case Volume.High:  
                Console.WriteLine("The volume has been turned up.");  
                break;  
        }  
  
        Console.WriteLine();  
    }  
  
    // iterate through Volume enum by name  
    publicvoid ListEnumMembersByName()  
    {  
        Console.WriteLine("\n---------------------------- ");  
        Console.WriteLine("Volume Enum Members by Name:");  
        Console.WriteLine("----------------------------\n");  
  
        // get a list of member names from Volume enum,  
        // figure out the numeric value, and display  
        foreach (string volume in Enum.GetNames(typeof(Volume)))  
        {  
            Console.WriteLine("Volume Member: {0}\n Value: {1}",   
                volume, (byte)Enum.Parse(typeof(Volume), volume));  
        }  
    }  
  
    // iterate through Volume enum by value  
    publicvoid ListEnumMembersByValue()  
    {  
        Console.WriteLine("\n----------------------------- ");  
        Console.WriteLine("Volume Enum Members by Value:");  
        Console.WriteLine("-----------------------------\n");  
  
        // get all values (numeric values) from the Volume  
        // enum type, figure out member name, and display  
        foreach (byte val in Enum.GetValues(typeof(Volume)))  
        {  
            Console.WriteLine("Volume Value: {0}\n Member: {1}",   
                val, Enum.GetName(typeof(Volume), val));  
        }  
    }  
}

The code in Listing 17-3 includes three method calls to *GetEnumFromUser*, *ListEnumMembersByName*, and *ListEnumMembersByValue*. Each of these methods demonstrate a different aspect of using System.Enum to work with enums.

The *GetEnumFromUser* method shows how to obtain *int* input and translate it to an appropriate enum type. Converting an *int* to an enum makes the code more readable and type safe. The following is an excerpt from Listing 17-3 that shows the pertinent part of the code that performs the conversion:

        // get value user provided  
        string volString = Console.ReadLine();  
        int volInt = Int32.Parse(volString);  
  
        // perform explicit cast from  
        // int to Volume enum type  
        Volume myVolume = (Volume)volInt;

After the program displays a menu, it prompts the user for a selection in the form of a number (1, 2, or 3). When the user makes a selection and presses the Enter key, the code reads the value with *Console.ReadLine*, which returns the value as a *string* type. Since you can only cast an *int* to a *Volume* enum type, the user's input must be converted from a *string* to an *int* with the *Int32.Parse* method. Converting the *in*t to a *Volume* enum type is simply a matter of applying a cast operation during assignment.

To get all the members of an enum at the same time, you can use the *GetNames* method of the *System.Enum* type, which returns a *string* array of the names of all an enum's members. An excerpt from the *ListEnumMembersByName* method in Listing 17.3 that shows this appears below:

        // get a list of member names from Volume enum,  
        // figure out the numeric value, and display  
        foreach (string volume in Enum.GetNames(typeof(Volume)))  
        {  
            Console.WriteLine("Volume Member: {0}\n Value: {1}",   
                volume, (byte)Enum.Parse(typeof(Volume), volume));  
        }

Because *GetNames* returns an array of *strings*, it is easy to use in a loop statement such as *foreach*. Something you may be curious about in the code above is the second parameter to the *WriteLine* method's format string. Given the enum type and a string representation of the member name, you can use the *Enum.Parse* method to get the underlying value of that member. Because the *Volume* enum's base type is byte, the return value from *Enum.Parse*must be cast to a *byte* before assignment, forcing the numeric representation of the enum value to appear. If we would have omitted the *byte* cast, the output would be the Volume enum member, which would then be converted to a string representation of the member name, which is not what the code intended to show.

Instead of getting names of all the members of an enum, you may have a reason to get all the values of the enum at one time. The code below, from the *ListEnumMembersByValue* method in Listing 17.3, shows how to accomplish this:

        // get all values (numeric values) from the Volume  
        // enum type, figure out member name, and display  
        foreach (byte val in Enum.GetValues(typeof(Volume)))  
        {  
            Console.WriteLine("Volume Value: {0}\n Member: {1}",   
                val, Enum.GetName(typeof(Volume), val));  
        }

Given the type of the enum, the *GetValues* method of System.Enum will return an array of the given enum's base type, which in this case is *byte*. While iterating through this list, each member is printed to the console showing its value and name. The name is obtained by using the *GetName* method of System.Enum, which accepts an enum type and value for which to get the corresponding name of.

**Summary**

Enums are lists of strongly typed constants with members that are symbolic names, corresponding to an underlying integral type. Enum base types can be changed and member values can be specified. The System.Enum .NET Framework Class Library type is the base class of enum types and contains methods that allow you to work with enums in different ways, such as working with a list of names or values, converting from value to name, and converting from name to value. For more information on the System.Enum type, see the .NET Framework SDK documentation.

**Overloading Operators**

This lesson shows you how to overload C# operators. Our objectives are as follows:

* Understand what operator overloading is
* Determine when it is appropriate to overload an operator
* Learn how to overload an operator
* Familiarize yourself with rules for operator overloading

**About Operator Overloading**

In [Lesson 2](http://www.csharp-station.com/Tutorial/CSharp/Lesson02), you learned what operators were available in C#, which included *+* (plus), *-* (minus), *^* (exclusive or), and others. Operators are defined for the built-in types, but that's not all. You can add operators to your own types, allowing them to be used much like the operators with the built-in C# types.

To understand the need for operator overloading, imagine that you need to perform matrix math operations in your program. You could instantiate a couple 2-dimensional arrays and do what you need. However, add the requirement for the matrix behavior to be reusable. Because you need to do the same thing in other programs and want to take advantage of the fact that you have already written the code, you will want to create a new type.

So, you create a Matrix type, which could be a class or a struct. Now consider how this Matrix type would be used. You would want to initialize two or more Matrix instances with data and then do a mathematical operation with them, such as add or get a dot product. To accomplish the mathematical operation, you could implement an *Add()*, *DotProduct()*, and other methods to get the job done. Using the classes would look something like this:

Matrix result = mat1.Add(mat2); // instance

or

Matrix result = Matrix.Add(mat1, mat2); // static

or even worse

Matrix result = mat1.DotProduct(mat2).DotProduct(mat3); // and so on...

The problem with using methods like this is that it is cumbersome, verbose, and unnatural for the problem you are trying to solve. It would be much easier to have a *+* operator for the add operation and a *\** operator for the dot product operation. The following shows how the syntax appears using operators:

Matrix result = mat1 + mat2;

or

Matrix result = mat1 \* mat2;

or even better

Matrix result = mat1 \* mat2 \* mat3 \* mat4;

This is much more elegant and easier to work with. For a single operation, one could argue that the amount of work to implement one syntax over the other is not that great. However, when chaining multiple mathematical operations, the syntax is much simpler. Additionally, if the primary users of your type are mathematicians and scientists, operators are more intuitive and natural.

**When Not to Use Operator Overloading**

A lot of the discussion, so far, has emphasized the need to write code and implement types in the simplest and most natural way possible. A very important concept to remember is that although operators are simple, they are not always natural. In the example above it made sense to use operators with the *Matrix* type. This is similar to the reason why operators make sense with the built-in types such as *int* and *float*. However, it is easy to abuse operators and create convoluted implementations that are hard for anyone, including the original author, to understand.

For an example of a bad implementation, consider a *Car* class that needs an implementation allowing you to park the car in a garage. It would be a mistake to think that the following implementation was smart:

Car mySedan = new Car();

Garage parkingGarage = new Garage();

mySedan = mySedan + parkingGarage; // park car in the garage

This is bad code. If you ever have the temptation to do something like this - don't. No one will truly understand what it means and they will not think it is clever. Furthermore, it hurts the maintainability of the application because it is so hard to understand what the code does. Although the comment is there, it doesn't help much and if it wasn't there, it would be even more difficult to grasp the concept of adding a *Car* and a *Garage*.

The idea is this: Use operators where they lend understanding and simplicity to a type. Otherwise, do not use them.

**Implementing an Overloaded Operator**

The syntax required to implement an overloaded operator is much the same as a static method with a couple exceptions. You must use the *operator* keyword and specify the operator symbol being overloaded. Here's a skeleton example of how the dot product operator could be implemented:

public static Matrix operator \*(Matrix mat1, Matrix mat2)

{

// dot product implementation

}

Notice that the method is static. Use the keyword *operator* after specifying the return type, *Matrix* in this case. Following the *operator* keyword, the actual operator symbol is specified and then there is a set of parameters to be operated on. See Listing 18-1 for a full example of how to implement and use an overloaded operator.

**Listing 18-1. Implementing an Overloaded Operator: Matrix.cs**

using System;

class Matrix

{

publicconstint DimSize = 3;

privatedouble[,] m\_matrix = newdouble[DimSize, DimSize];

// allow callers to initialize

publicdoublethis[int x, int y]

{

get { return m\_matrix[x, y]; }

set { m\_matrix[x, y] = value; }

}

// let user add matrices

publicstatic Matrix operator +(Matrix mat1, Matrix mat2)

{

Matrix newMatrix = new Matrix();

for (int x=0; x < DimSize; x++)

for (int y=0; y < DimSize; y++)

newMatrix[x, y] = mat1[x, y] + mat2[x, y];

return newMatrix;

}

}

class MatrixTest

{

// used in the InitMatrix method.

publicstatic Random m\_rand = new Random();

// test Matrix

staticvoid Main()

{

Matrix mat1 = new Matrix();

Matrix mat2 = new Matrix();

// init matrices with random values

InitMatrix(mat1);

InitMatrix(mat2);

// print out matrices

Console.WriteLine("Matrix 1: ");

PrintMatrix(mat1);

Console.WriteLine("Matrix 2: ");

PrintMatrix(mat2);

// perform operation and print out

results

Matrix mat3 = mat1 + mat2;

Console.WriteLine();

Console.WriteLine("Matrix 1 + Matrix

2 = ");

PrintMatrix(mat3);

Console.ReadLine();

}

// initialize matrix with random values

publicstaticvoid InitMatrix(Matrix mat)

{

for (int x=0; x < Matrix.DimSize; x++)

for (int y=0; y < Matrix.DimSize; y++)

mat[x, y] = m\_rand.NextDouble();

}

// print matrix to console

publicstaticvoid PrintMatrix(Matrix mat)

{

Console.WriteLine();

for (int x=0; x < Matrix.DimSize; x++)

{

Console.Write("[ ");

for (int y=0; y < Matrix.DimSize; y++)

{

// format the output

Console.Write("{0,8:#.000000}", mat[x, y]);

if ((y+1 % 2) < 3)

Console.Write(", ");

}

Console.WriteLine(" ]");

}

Console.WriteLine();

}

}

Similar to the skeleton example of the dot product operator, the Matrix class in Listing 18-1 contains an operator overload for the + operator. For your convenience, I've extracted the pertinent overload implementation in the code below:

// let user add matrices

publicstatic Matrix operator +(Matrix mat1, Matrix mat2)

{

Matrix newMatrix = new Matrix();

for (int x=0; x < DimSize; x++)

for (int y=0; y < DimSize; y++)

newMatrix[x, y] = mat1[x, y] + mat2[x, y];

return newMatrix;

}

The operator is *static*, which is the only way it can and should be declared because an operator belongs to the type and not a particular instance. There are just a few rules you have to follow when implementing operator overloads. What designates this as an operator is the use of the keyword *operator*, followed by the *+* symbol. The parameter types are both of the enclosing type, *Matrix*. The implementation of the operator overload creates a new instance of the return type and performs a matrix add.

**Operator Rules**

C# enforces certain rules when you overload operators. One rule is that you must implement the operator overload in the type that will use it. This is sensible because it makes the type self-contained.

Another rule is that you must implement matching operators. For example, if you overload ==, you must also implement !=. The same goes for <= and >=.

When you implement an operator, its compound operator works also. For example, since the *+* operator for the *Matrix* type was implemented, you can also use the *+=* operator on *Matrix* types.

**Summary**

Operator overloading allows you to implement types that behave like the built-in types when using operators. Be sure to use operators in a way that is natural and understandable for the type. Syntax for implementing operators is much like a static method, but includes the *operator* keyword and the operator symbol in place of an identifier. Additionally, there are rules, such as maintaining symmetry,for using operators, which encourage construction of robust types.

**Encapsulation**

Earlier in this tutorial, you learned about two of the important principles of object-oriented programming, [Inheritance](http://www.csharp-station.com/Tutorial/CSharp/Lesson08) and [Polymorphism](http://www.csharp-station.com/Tutorial/CSharp/Lesson09). Now that you've seen much of the syntax of C#, I'll show you how C# supports the another of the object-oriented principles - Encapsulation. This lesson will discuss Encapsulation with the following objectives:

* Understand the object-oriented principle of Encapsulation.
* Learn the available modifiers for type members.
* Protect object state through properties.
* Control access to methods.
* Learn how to modify types for assembly encapsulation

**What is Encapsulation and How Does It Benefit Me?**

In object-oriented programming, you create objects that have state and behavior. An object's state is the data or information it contains. For example, if you have a *BankAccount* object, its state could be *Amount* and *CustomerName*. Behavior in an object is often represented by methods. For example, the BankAccount object's behavior could be *Credit*, *Debit*, and *GetAmount*. This sounds like a nice definition of an object, and it is, but you must also consider how this object will be used.

When designing an object, you must think about how others could use it. In a best-case scenario any program using the object would be well designed and the code would never change. However, the reality is that programs do change often and in a team environment many people touch the same code at one time or another. Therefore, it is beneficial to consider what could go wrong as well as the pristine image of how the object \*should\* be used.

In the case of the *BankAccount* object, examine the situation where code outside of your object could access a *decimalAmount* field or a *stringCustomerName* field. At the point of time that the code is written, everything would work well. However, later in the development cycle, you realize that the *BankAccount* object should keep track of an *intCustomerID* rather than *string CustomerName* because you don't want to duplicate relationships between information (or some other valid reason to alter the definition of internal state). Such changes cause a rippling effect in your code because it was built to use the *BankAccount* class, as originally designed (with *CustomerName* being a string), and you must now change code that accesses that state throughout your entire application.

The object-oriented principle of Encapsulation helps avoid such problems, allowing you to hide internal state and abstract access to it though type members such as methods, properties, and indexers. Encapsulation helps you reduce coupling between objects and increases the maintainability of your code.

**Type Member Access Modifiers**

An access modifier allows you to specify the visibility of code outside a type or assembly. Access modifiers can be applied to either types or type members. A later section on Type Access Modifiers discusses modifiers that can be applied to types. This section discusses those modifiers that apply to type members and how they affect visibility.

Generally, you should hide the internal state of your object from direct access from outside code. Then implement other members, such as methods and properties, that wrap that state. This allows the internal implementation of the state to change at will, while the members wrapping the state can still return a representation of the state that doesn't change. This means that outside code will access your object via members that wrap state and be guaranteed that the type of information extracted is consistent. Additionally, because external code doesn't have access to the internal state of your object, they can't alter that state in an inconsistent manner that could break the way your object works.

The first step in encapsulating object state is to determine what type of access that outside code should have to the members of your type. This is performed with access modifiers. The type of access granted varies from no external access at all to full public access and a few variations in between the extremes. Table 19-1 lists all of the type member access modifiers and explains their meaning.

**Table 19-1. Type member access modifiers control what code has access to a specified type member.**

|  |  |
| --- | --- |
| **Access Modifier** | **Description (who can access)** |
| private | Only members within the same type. (default for type members) |
| protected | Only derived types or members of the same type. |
| internal | Only code within the same assembly. Can also be code external to object as long as it is in the same assembly. (default for types) |
| protected internal | Either code from derived type or code in the same assembly. Combination of protected OR internal. |
| public | Any code. No inheritance, external type, or external assembly restrictions. |

As you've learned from previous lessons of the C# Tutorial, types contain several types of members, including constructors, properties, indexers, methods, and others.  Rather than show you an exhaustive list of all of the permutations of access modifiers you can use with these members, I'll take a more practical approach and describe a sub-set of access modifiers used on properties and methods.

**Opening Type Members to *public* Access**

You've seen the *public* access modifier used in earlier parts of the C# Tutorial. Any time the *public* access modifier is used on a type member, calling code will be able to access the type member. If you make your type member public, you are giving everyone permission to use it. Listing 19-1 shows an example of using the *public* access modifier on a method.

**Listing 19-1. Declaring a Method with a public Access Modifier: BankAccountPublic.cs**

using System;  
  
classBankAccountPublic  
{  
    publicdecimal GetAmount()  
    {  
        return1000.00m;  
    }  
}

The *GetAmount()* method in Listing 19-1 is public meaning that it can be called by code that is external to this class. Now, you can write the following code, elsewhere in your program, to use this method:

 BankAccountPublic bankAcctPub = newBankAccountPublic();  
  
// call a public method  
decimal amount = bankAcctPub.GetAmount();

All you need to do, as shown above, is create an instance of the class that contains the method and then call the method through that instance. Because it is *public*, you won't have a problem. Remember that the default access for a type member is *private*, which we'll talk about next. This means that if you forget the *public* modifier, and didn't use any modifier at all, you would receive a compiler error.

**Hiding Type Members with *private* Access**

A *private* type member is one that can only be accessed by members within the same type. For example, if the *BankAccount* class has a private member, only other members of the *BankAccount* class can access or call that member.

Although the default access for type members is *private*, I prefer to be explicit about my intentions when declaring type members and include the access modifier, rather than rely on defaults. I think it makes the code easier to read and makes it clear to other developers what my true intention is. Listing 19-2 shows how to use the *private* access modifier and offers an example of why you would want to use it.

**Listing 19-2. Declaring a private Field: BankAccountPrivate.cs**

using System;  
  
classBankAccountPrivate  
{  
    privatestring m\_name;  
  
    publicstring CustomerName  
    {  
        get { return m\_name; }  
        set { m\_name = value; }  
    }  
}

It's common to encapsulate the state of your type with properties. In fact, I always wrap my type state in a property. In Listing 19-2, you can see how the name of the customer is held in the *m\_name* field, but it is wrapped (encapsulated) with the *CustomerName* property. Because *m\_name* is declared as *private*, code outside the *BankAccountPrivate* class can't access it directly. They must use the *publicCustomerName* property instead.

Now you can change the implementation of *m\_name* in any way you want. For example, what if you wanted it to be an ID of type *int* and the *CustomerName* property would do a search to find the name or what if you wanted to have first and last name values that the *CustomerName* property could concatenate. There are all kinds of things happening to your code in maintenance that will causes implementation to change. The point is that *private* members allow the implementation to change without constraining the implementation or causing rippling effects throughout your code base that would have occurred if that external code had access to the members of your type.

The *private* and *public* access modifiers are at the two extremes of access, either denying all external access or allowing all external access, respectively. The other access modifiers are like different shades of gray between these two extremes, including the *protected* modifier, discussed next.

**Access for Derived Types with the *protected* Access Modifier**

In some ways, the *protected* access modifier acts like both the *private* and *public* access modifiers. Like *private*, it only allows access to members within the same type, except that it acts like *public* only to derived types. Said another way, *protected* type members can only be accessed by either members within the same type or members of derived types.

Returning to the *BankAccount* example, what if you needed to call code to close an account? Furthermore, what if there were different types of accounts? Each of these different account types would have their own logic for closing, but the basic process would be the same for all account types. If this sounds to you like the description of Polymorphism, you would be on the right track. Back in [Lesson 9](http://www.csharp-station.com/Tutorial/CSharp/Lesson09), we discussed polymorphism and how it allows us to treat different classes the same way. You may want to refer to [Lesson 9](http://www.csharp-station.com/Tutorial/CSharp/Lesson09) for a refresher before looking at the next example.

In the case of closing an account, there are several things that need to be done like calculating interest that is due, applying penalties for early withdrawal, and doing the work to remove the account from the database. Individually, you don't want any code to call methods of the *BankAccount* class unless all of the methods are called and each method is called in the right order. For example, what if some code called the method to delete the account from the database and didn't calculate interest or apply penalties? Someone would lose money. Also, if the calling code were to delete the account first then the other methods would run into errors because the account information isn't available. Therefore, you need to control this situation and Listing 19-3 shows how you can do it.

**Listing 19-3. Declaring protected Methods: BankAccountProtected.cs**

using System;  
  
classBankAccountProtected  
{  
    publicvoid CloseAccount()  
    {  
        ApplyPenalties();  
        CalculateFinalInterest();  
        DeleteAccountFromDB();  
    }  
  
    protectedvirtualvoid ApplyPenalties()  
    {  
        // deduct from account  
    }  
  
    protectedvirtualvoid CalculateFinalInterest()  
    {  
        // add to account  
    }  
  
    protectedvirtualvoid DeleteAccountFromDB()  
    {  
        // send notification to data entry personnel  
    }  
}

The most important parts of Listing 19-3 are that the *CloseAccount* method is public and the other methods are protected. Any calling code can instantiate *BankAccountProtected*, but it can only call the *CloseAccount* method. This gives you protection from someone invoking the behavior of your object in inappropriate ways. Your business logic is sound.

At the end of this section, you'll see an example of how to call the code in Listing 19-3. For now, it is essential that you see how the other pieces fit together first.

If you only wanted the *BankAccountProtected* class to operate on its own members, you could have made the *protected* methods *private* instead. However, this code supports a framework where you can have different account types such as Savings, Checking, and more. You will be able to add new account types in the future because the *BankAccountProtected* class is designed to support them with *protected* virtual methods. Listings 19-4 and 19-5 show you the *SavingsAccount* and *CheckingAccount* classes that derive from the *BankAccountProtected* class.

**Listing 19-4. Derived SavingsAccount Class Using protected Members of its Base Class: SavingsAccount.cs**

using System;  
  
classSavingsAccount : BankAccountProtected  
{  
    protectedoverridevoid ApplyPenalties()  
    {  
        Console.WriteLine("Savings Account Applying Penalties");  
    }  
  
    protectedoverridevoid CalculateFinalInterest()  
    {  
        Console.WriteLine("Savings Account Calculating Final Interest");  
    }  
  
    protectedoverridevoid DeleteAccountFromDB()  
    {  
        base.DeleteAccountFromDB();  
        Console.WriteLine("Savings Account Deleting Account from DB");  
    }  
}

Notice how *SavingsAccount* derives from *BankAccountProtected*. *SavingsAccount* can access any of the *protected* members of the *BankAccountProtected* class which is its base class. It demonstrates this fact via the call to *base.DeleteAccountFromDB* in it's *DeleteAccountFromDB* method. If the inheritance part of Listing 19-4 is a little confusing, you can visit [Lesson 8: Class Inheritance](http://www.csharp-station.com/Tutorial/CSharp/Lesson08) for a refresher and better understanding. Each method of *SavingsAccount* has the protected access modifier also, which simply means that classes derived from *SavingsAccount* can access those *SavingsAccount* members with the *protected* access modifier. The same situation exists with the *CheckingAccount* class, shown in Listing 19-5.

**Listing 19-5. Derived CheckingAccount Class Using protected Members of its Base Class: CheckingAccount.cs**

using System;  
  
classCheckingAccount : BankAccountProtected  
{  
    protectedoverridevoid ApplyPenalties()  
    {  
        Console.WriteLine("Checking Account Applying Penalties");  
    }  
  
    protectedoverridevoid CalculateFinalInterest()  
    {  
        Console.WriteLine("Checking Account Calculating Final Interest");  
    }  
  
    protectedoverridevoid DeleteAccountFromDB()  
    {  
        base.DeleteAccountFromDB();  
        Console.WriteLine("Checking Account Deleting Account from DB");  
    }  
}

The *CheckingAccount* class in Listing 19-5 is implemented similar to *SavingsAccount* from Listing 19-4. If you were writing this, the difference would be that the methods of each class would have unique implementations. For example, the business rules associated with the final interest calculation would differ, depending on whether the account type was checking or savings.

Notice the call to the base class method in the *DeleteAccountFromDB* method in *CheckingAccount*. Just like *SavingsAccount*, *CheckingAccount* has access to *BankAccountProtected*'s protected method because it is a derived class. This is a common pattern in polymorphism because derived classes often have a responsibility to call virtual base class methods to ensure critical functionality has the opportunity to execute. You would consult the method documentation to see if this was necessary. Without a *protected* access modifier, your only option would have been to make the base class method *public*; which, as explained earlier, is dangerous.

To use the code from Listings 19-3, 19-4, and 19-5, you can implement the following code:

BankAccountProtected[] bankAccts = newBankAccountProtected[2];  
bankAccts[0] = newSavingsAccount();  
bankAccts[1] = newCheckingAccount();  
  
foreach (BankAccountProtected acct in bankAccts)  
{  
    // call public method, which invokes protected virtual methods  
    acct.CloseAccount();  
}

Since both *SavingsAccount* and *CheckingAccount* derive from *BankAccountProtected*, you can assign them to the *bankAccts* array. They both override the *protectedvirtual* methods of *BankAccountProtected*, so it is the *SavingsAccount* and *CheckingAccount* methods that are called when *CloseAccount* in *BankAccountProtected* executes. Remember that the only reason the methods of *SavingsAccount* and *CheckingAccount* can call their *virtual* base class methods, as in the case of *DeleteAccountFromDB*, is because the *virtual* base class methods are marked with the *protected* access modifier.

**A Quick Word on *internal* and *protected internal* Access Modifiers**

In practice, most of the code you write will involve the *public*, *private*, and *protected* access modifiers. However, there are two more access modifiers that you can use in more sophisticated scenarios: *internal* and *protected internal*.

You would use *internal* whenever you created a separate class library and you don't want any code outside of the library to access the code with *internal* access. The *protected internal* is a combination of the two access modifiers it is named after, which means either *protected***or***internal*.

**Access Modifiers for Types**

So far, the discussion of access modifiers has only applied to the members of types.  However, the rules are different for the types themselves. When talking about types, I'm referring to all of the C# types, including classes, structs, interfaces, delegates, and enums. Nested types, such as a class defined within the scope of a class, are considered type members and fall under the same access rules as other type members.

Types can have only two access modifiers: *public* or *internal*. The default, if you don't specify the access modifier, is *internal*. Looking at all of the classes used in this lesson, you can see that they are *internal* because they don't have an access modifier. You can explicitly specify *internal* like this:

internalclassInternalInterestCalculator  
{  
    // members go here  
}

Perhaps the *InternalInterestCalculator*, shown above, has special business rules that you don't want other code to use. Now, it is in a class library of its own and can only be accessed by other code inside of that same class library (DLL).

**Note:** To be more specific, *internal* means that only code in the same assembly can access code marked as *internal*. However, discussing the definition of an assembly is outside the scope of this lesson, so I am simplifying the terminology.

If you declared a class inside of a class library that you wanted other code to use, you would give it a *public* access modifier. The following code shows an example of applying the *public* access modifier to a type:

publicclassBankAccountExternal  
{  
    // members go here  
}

Clearly, a bank account is something you would want to access from outside of a class library. Therefore, it only makes sense to give it a *public* access modifier as shown in the *BankAccountExternal* class above.

**Tip:** A common gottcha in Visual Studio occurs when you create a new class in a class library. The default template doesn't include an access modifier. Then, when you try to write code that uses the new class in your program (which references the class library), you get a compiler error saying that the class doesn't exist. Well, you know it exists because you just wrote it and are looking at the page. If you've already seen the clue I've given you so far, you'll key on the fact that the default template left out the access modifier on the type. This makes the class default to *internal*, which can't be seen outside of the assembly. The fix is to give the class a *public* modifier, like the *BankAccountExternal* class above.

**Summary**

Encapsulation is an object-oriented principle of hiding the internal state and behavior of an object, making your code more maintainable. In C#, you can manage encapsulation with access modifiers. For example, the *public* access modifier allows access to any code but the *private* access modifier restricts access to only members of a type. Other access modifiers restrict access in the range somewhere between *public* and *private*. While you can use any of the access modifiers on type members, the only two access modifiers you can use on types are the *public* and *internal*.

**Introduction to Generic Collections**

All the way back in Lesson 02, you learned about arrays and how they allow you to add and retrieve a collection of objects. Arrays are good for many tasks, but C# v2.0 introduced a new feature called generics. Among many benefits, one huge benefit is that generics allow us to create collections that allow us to do more than allowed by an array. This lesson will introduce you to generic collections and how they can be used. Here are the objectives for this lesson:

* Understand how generic collections can benefit you
* Learn how to create and use a generic List
* Write code that implements a generic Dictionary

**What Can Generics Do For Me?**

Throughout this tutorial, you've learned about types, whether built-in (*int*, *float*, *char*) or custom (*Shape*, *Customer*, *Account*). In .NET v1.0 there were collections, such as the *ArrayList* for working with groups of objects. An *ArrayList* is much like an array, except it could automatically grow and offered many convenience methods that arrays don't have. The problem with *ArrayList* and all the other .NET v1.0 collections is that they operate on type *object*. Since all objects derive from the *object* type, you can assign anything to an *ArrayList*. The problem with this is that you incur performance overhead converting value type objects to and from the *object* type and a single *ArrayList* could accidentally hold different types, which would cause hard to find errors at runtime because you wrote code to work with one type. Generic collections fix these problems.

A generic collection is strongly typed (type safe), meaning that you can only put one type of object into it. This eliminates type mismatches at runtime. Another benefit of type safety is that performance is better with value type objects because they don't incur overhead of being converted to and from type *object*. With generic collections, you have the best of all worlds because they are strongly typed, like arrays, and you have the additional functionality, like *ArrayList* and other non-generic collections, without the problems.

The next section will show you how to use a generic *List* collection.

**Creating Generic *List<T>* Collections**

The pattern for using a generic *List* collection is similar to arrays. You declare the *List*, populate its members, then access the members. Here's a code example of how to use a *List*:

List<int> myInts = new List<int>();

myInts.Add(1);

myInts.Add(2);

myInts.Add(3);

for (int i = 0; i < myInts.Count; i++)

{

Console.WriteLine("MyInts: {0}", myInts[i]);

}

The first thing you should notice is the generic collection *List<int>*, which is referred to as List of int. If you looked in the documentation for this class, you would find that it is defined as *List<T>*, where *T* could be any type. For example, if you wanted the list to work on *string* or *Customer* objects, you could define them as *List<string>* or *List<Customer>* and they would hold only *string* or *Customer* objects. In the example above, *myInts* holds only type *int*.

Using the *Add* method, you can add as many *int* objects to the collection as you want. This is different from arrays, which have a fixed size. The *List<T>* class has many more methods you can use, such as *Contains*, *Remove*, and more.

There are two parts of the *for* loop that you need to know about. First, the condition uses the *Count* property of *myInts*. This is another difference between collections and arrays in that an array uses a *Length* property for the same thing. Next, the way to read from a specific position in the *List<T>* collection, *myInts[i]*, is the exact same syntax you use with arrays.

The next time you start to use a single-dimension array, consider using a *List<T>* instead. That said, be sure to let your solution fit the problem and use the best tool for the job. i.e. it's common to work with *byte[]* in many places in the .NET Framework.

**Working with *Dictionary<TKey, TValue>* Collections**

Another very useful generic collection is the *Dictionary*, which works with key/value pairs. There is a non-generic collection, called a *Hashtable* that does the same thing, except that it operates on type *object*. However, as explained earlier in this lesson, you want to avoid the non-generic collections and use thier generic counterparts instead. The scenario I'll use for this example is that you have a list of *Customers* that you need to work with. It would be natural to keep track of these *Customers* via their *CustomerID*. The *Dictionary* example will work with instances of the following *Customer* class:

publicclass Customer

{

public Customer(int id, string name)

{

ID = id;

Name = name;

}

privateint m\_id;

publicint ID

{

get { return m\_id; }

set { m\_id = value; }

}

privatestring m\_name;

publicstring Name

{

get { return m\_name; }

set { m\_name = value; }

}

}

The *Customer* class above has a constructor to make it easier to initialize. It also exposes its state via public properties. It isn't very sophisticated at this point, but that's okay because its only purpose is to help you learn how to use a *Dictionary* collection.  The following example populates a *Dictionary* collection with *Customer* objects and then shows you how to extract entries from the *Dictionary*:

Dictionary<int, Customer> customers = new Dictionary<int, Customer>();

Customer cust1 = new Customer(1, "Cust 1");

Customer cust2 = new Customer(2, "Cust 2");

Customer cust3 = new Customer(3, "Cust 3");

customers.Add(cust1.ID, cust1);

customers.Add(cust2.ID, cust2);

customers.Add(cust3.ID, cust3);

foreach (KeyValuePair<int, Customer> custKeyVal in customers)

{

Console.WriteLine(

"Customer ID: {0}, Name: {1}",

custKeyVal.Key,

custKeyVal.Value.Name);

}

The *customers* variable is declared as a *Dictionary<int, Customer>*.  Considering that the formal declaration of *Dictionary* is *Dictionary<TKey, TValue>*, the meaning of *customers* is that it is a *Dictionary* where the key is type *int* and the value is type *Customer*. Therefore, any time you add an entry to the *Dictionary*, you must provide the key because it is also the key that you will use to extract a specified *Customer* from the *Dictionary*.

I created three *Customer* objects, giving each an *ID* and a *Name*. I'll use the *ID* as the key and the entire *Customer* object as the value. You can see this in the calls to *Add*, where *custX.ID* is added as the key (first parameter) and the *custX* instance is added as the value (second parameter).

Extracting information from a *Dictionary* is a little bit different. Iterating through the *customersDictionary* with a *foreach* loop, the type returned is *KeyValuePair<TKey, TValue>*, where *TKey* is type *int* and *TValue* is type *Customer* because those are the types that the *customersDictionary* is defined with.

Since *custKeyVal* is type *KeyValuePair<int, Customer>* it has *Key* and *Value* properties for you to read from. In our example, *custKeyVal.Key* will hold the *ID* for the *Customer* instance and *custKeyVal.Value* will hold the whole *Customer* instance. The parameters in the *Console.WriteLine* statement demonstrate this by printing out the *ID*, obtained through the *Key* property, and the *Name*, obtained through the *Name* property of the *Customer* instance that is returned by the *Value* property.

The *Dictionary* type is handy for those situations where you need to keep track of objects via some unique identifier. For your convenience, here's Listing 20-1, shows how both the *List* and *Dictionary* collections work.

**Listing 20-1. Introduction to Using Generic Collections with an Example of the List<T> and Dictionary<TKey, TValue> Generic Collections**

using System;

using System.Collections.Generic;

publicclass Customer

{

public Customer(int id, string name)

{

ID = id;

Name = name;

}

privateint m\_id;

publicint ID

{

get { return m\_id; }

set { m\_id = value; }

}

privatestring m\_name;

publicstring Name

{

get { return m\_name; }

set { m\_name = value; }

}

}

class Program

{

staticvoid Main(string[] args)

{

List<int> myInts = new List<int>();

myInts.Add(1);

myInts.Add(2);

myInts.Add(3);

for (int i = 0; i < myInts.Count; i++)

{

Console.WriteLine("MyInts: {0}", myInts[i]);

}

Dictionary<int, Customer> customers = new Dictionary<int, Customer>();

Customer cust1 = new Customer(1, "Cust 1");

Customer cust2 = new Customer(2, "Cust 2");

Customer cust3 = new Customer(3, "Cust 3");

customers.Add(cust1.ID, cust1);

customers.Add(cust2.ID, cust2);

customers.Add(cust3.ID, cust3);

foreach (KeyValuePair<int, Customer> custKeyVal in customers)

{

Console.WriteLine(

"Customer ID: {0}, Name: {1}",

custKeyVal.Key,

custKeyVal.Value.Name);

}

Console.ReadKey();

}

}

Whenever coding with the generic collections, add a *using System.Collections.Generic* declaration to your file, just as in Listing 20-1.

**Summary**

Generic collections give you the best of all worlds with the strong typing of arrays and flexibility of non-generic collections. There are many more generic collections to choose from also, such as *Stack*, *Queue*, and *SortedDictionary*. Look in the *System.Collections.Generic* namespace for other generic collections.

**Topics on C# Type**

Throughout this tutorial, you've seen many different types, including those that are part of C# and custom designed types. If you've taken the samples and worked on them yourself, extending and writing your own programs, you are likely to have experienced errors associated with type. For example, you can't assign a *double* to an *int* without using a cast operator to perform the conversion. Another feature of C# concerns the semantic differences between reference and value types. Such problems should make you wonder why this is so and that's what this lesson is for. Here are the objectives for this lesson:

* Understand the need for type safety
* See how to convert one type to another
* Learn about reference types
* Learn about value types
* Comprehend the semantic differences between reference and value types

**Why Type Safety?**

In untyped languages, such as scripting languages, you can assign one variable to another and the compiler/interpreter will use an intelligent algorithm to figure out how the assignment should be done. If the assignment is between two variables of the same type, all is good. However, if the assignment is between different types, you could have serious problems.

For example, if you assigned an *int* value to a *float* variable it would convert okay because the fractional part of the new *float* would just be zero. However, if you went the other way and assigned a *float* value to an *int* variable, that would most likely be a problem. You would lose all of the precision of the original *float* value. Consider the damage that could be caused if the *float* value represented a chemical ingredient, an engineering measurement, or a financial value. Finding such an error would be difficult and particularly expensive, especially if the error didn't show up until your application was in production (already being used by customers).

**Using the Cast Operator for Conversions**

In [Lesson 02: Operators, Types, and Variables](http://www.csharp-station.com/Tutorial/CSharp/Lesson02), you learned about C# types and operators. It explained the size and precision of the various types and there is a list of available operators. The cast operator, *(x)*, is listed first as a primary operator in Table 2-4. When you must convert a type that doesn't fit, it must be done via what is called an explicit conversion, which uses the cast operator. Listing 22-1 has an example of an implicit conversion, which doesn't require the cast operator, and an explicit conversion.

**Listing 22-1. Cast Operators**

using System;

class Program

{

staticvoid Main()

{

float lengthFloat = 7.35f;

// lose precision - explicit conversion

int lengthInt = (int)lengthFloat;

// no problem - implicit conversion

double lengthDouble = lengthInt;

Console.WriteLine("lengthInt = " + lengthInt);

Console.WriteLine("lengthDouble = " + lengthDouble);

Console.ReadKey();

}

}

Here's the output:

lengthInt = 7

lengthDouble = 7

Since a *float*, *lengthFloat*, has a fractional part but an *int*, *lengthInt*, doesn't; the types aren't compatible. Because of type safety, C# won't allow you to assign *lengthFloat* directly to *lengthInt*, which would be dangerous. For your protection, you must use a cast operator, *(int)*, to force the explicit conversion of *lengthFloat* to *lengthInt*. In the output, you can see that *lengthInt* is *7*, showing that it lost the fractional part of the *7.35f* value from *lengthFloat*.

The assignment from *lengthInt* to *lengthDouble* is safe because a double is 64-bit and an int is 32-bit, meaning that you won't lose information. Therefore, the conversion is implicit, meaning that you can perform the assignment without the cast operator.

**Understanding Reference Types**

Reference type variables are named appropriately (reference) because the variable holds a reference to an object. In C and C++, you have something similar that is called a pointer, which points to an object. While you can modify a pointer, you can't modify the value of a reference - it simply points at the object in memory.

An important fact you need to understand is that when you are assigning one reference type variable to another, only the reference is copied, not the object. The variable holds the reference and that is what is being copied. Listing 22-2 shows how this works.

**Listing 22-2. Reference Type Assignment**

using System;

class Employee

{

privatestring m\_name;

publicstring Name

{

get { return m\_name; }

set { m\_name = value; }

}

}

class Program

{

staticvoid Main()

{

Employee joe = new Employee();

joe.Name = "Joe";

Employee bob = new Employee();

bob.Name = "Bob";

Console.WriteLine("Original Employee Values:");

Console.WriteLine("joe = " + joe.Name);

Console.WriteLine("bob = " + bob.Name);

// assign joe reference to bob variable

bob = joe;

Console.WriteLine();

Console.WriteLine("Values After Reference Assignment:");

Console.WriteLine("joe = " + joe.Name);

Console.WriteLine("bob = " + bob.Name);

joe.Name = "Bobbi Jo";

Console.WriteLine();

Console.WriteLine("Values After Changing One Instance:");

Console.WriteLine("joe = " + joe.Name);

Console.WriteLine("bob = " + bob.Name);

Console.ReadKey();

}

}

Here's the output:

Original Employee Values:

joe = Joe

bob = Bob

Values After Reference Assignment:

joe = Joe

bob = Joe

Values After Changing One Instance:

joe = Bobbi Jo

bob = Bobbi Jo

In Listing 22-2, I created two *Employee* instances, *joe* and *bob*. You can see in the output that the *Name* properties of both *Employee* instances each show their assigned values from when the objects were first created. After assigning *joe* to *bob*, the value of the *Name* properties of both instances are the same. This is what you might expect to see.

What might surprise you is the values that occur after assigning a value to the *Employee* instance variable named *joe*. If you look at the code closely, you'll notice that it doesn't change *bob* - only *Joe*. However, the results from the output show that the *Name* property in *bob* is the same as the *Name* property in *joe*. This demonstrates that after assigning *joe* to *bob*, both variables held references to the *joe* object. Only the reference was copied - not the object. This is why you see the results of printing *Name* in both *joe* and *bob* are the same because the change was on the object that they both refer to.

The following types are reference types:

* arrays
* class'
* delegates
* interfaces

**Note:** The primitive type, *string*, is also a reference type.

**Understanding Value Types**

Value type variables, as their name (value) suggests, hold the object value. A value type variable holds its own copy of an object and when you perform assignment from one value type variable to another, both the left-hand-side and right-hand-side of the assignment hold two separate copies of that value. Listing 22-3 shows how value type assignment works.

**Listing 22-3. Value Type Assignment**

using System;

struct Height

{

privateint m\_inches;

publicint Inches

{

get { return m\_inches; }

set { m\_inches = value; }

}

}

class Program

{

staticvoid Main()

{

Height joe = new Height();

joe.Inches = 71;

Height bob = new Height();

bob.Inches = 59;

Console.WriteLine("Original Height Values:");

Console.WriteLine("joe = " + joe.Inches);

Console.WriteLine("bob = " + bob.Inches);

// assign joe value to bob variable

bob = joe;

Console.WriteLine();

Console.WriteLine("Values After Value Assignment:");

Console.WriteLine("joe = " + joe.Inches);

Console.WriteLine("bob = " + bob.Inches);

joe.Inches = 65;

Console.WriteLine();

Console.WriteLine("Values After Changing One Instance:");

Console.WriteLine("joe = " + joe.Inches);

Console.WriteLine("bob = " + bob.Inches);

Console.ReadKey();

}

}

Here's the output:

Original Height Values:

joe = 71

bob = 59

Values After Value Assignment:

joe = 71

bob = 71

Values After Changing One Instance:

joe = 65

bob = 71

In Listing 22-3, you can see that the *Inches* property of *bob* and *joe* are initially set to different values. After assigning *joe* to *bob*, a value copy occurs, where both of the variables have the same value, but are two separate copies. To demonstrate value assignment results, notice what happens after setting *joe* to *65*; The output shows that *bob* did not change, which demonstrates that value types hold distinct copies of their objects.

The following types are value types:

* enum
* struct

**Note:** All of the primitive types (int, char, double, etc.), except for string, are value types.

**Reference Type and Value Type Differences**

From the previous paragraphs, you might already see that there is a difference reference type and value type assignment. Reference types copy a reference to an object and value types copy the object. If you don't know this, then the effects can be surprising in your code when performing tasks such as making assignments and passing arguments to methods.

**Summary**

This lesson provided a few tips on working with types in C#. You should now have a better understanding of type safety and how it can help you avoid problems. This lesson showed you how to use a cast operator to perform conversions and explained the difference between explicit and implicit conversions. You also know that the type system is divided between reference types and value types. To demonstrate the differences between reference types and value types, this lesson provided examples that showed how both reference types and value types behave during assignment.

**Working with Nullable Types**

Working with value types and data can sometimes be challenging because a value type doesn't normally hold a null value. This lesson shows you how to overcome this limitation with C# nullable types. Here's what you'll learn.

* Understand the problem that nullable types solve
* See how to declare a nullable type
* Learn how to use nullable types

**Understanding the Problem with Value Types and Null Values**

As explained in [Lesson 12: Structs](http://www.csharp-station.com/Tutorial/CSharp/Lesson12), the default value of a struct (value type) is some form of 0. This is another difference between reference types and value types, in addition to what was described in [Lesson 22: Topics on C# Type](http://www.csharp-station.com/Tutorial/CSharp/Lesson22). The default value of a reference type is *null*. If you're just writing C# code and managing your own data source, such as a file that holds data for your application, the default values for structs works fine.

In reality, most applications work with databases, which have their own type systems. The implications of working with database type systems is that you don't have a one-to-one mapping between C# and database types. One glaring difference is that database types can be set to *null*. A database has no knowledge of reference and value types, which are C# language (.NET Platform) concepts. This means that C# value type equivalents in the database, such as *int*, *decimal*, and *DateTime*, can be set to *null*.

Since a type in the database can be *null*, but your C# value type can't be *null*, you have to find some way to provide a translation in your C# code to account for *null* values. Effectively, the scheme you use will often be inconsistent from one program to another; something you often don't have a choice about. For example, what if you wanted to handle a *nullDateTime* from SQL Server as the minimum *DateTime* value in C#. After that project, your next task would be to read data from a legacy Foxpro database, whose minimum *DateTime* value is different from SQL Server. Because of this lack of constency and potential confusion, C# 2.0 added nullable types, which are more elegant and natural for working with *null* data.

**Declaring Nullable Types**

To declare a value type as nullable, append a question mark, *?*, to the type name. Here's how to declare a *DateTime* variable as a nullable type:

DateTime? startDate;

A *DateTime* can't normally hold a *null* value, but the declaration above enables *startDate* to hold *null*, as well as any legal *DateTime* value. The proper terminology is to refer to the type of *startDate* as a nullable *DateTime*.

You can assign a normal value to *startDate* like this:

startDate = DateTime.Now;

or you can assign *null*, like this:

startDate = null;

Here's another example that declares and initializes a nullable *int*:

int? unitsInStock = 5;

The *unitsInStock* in the example above can be assigned a value of *null* also.

**Working with Nullable Types**

When you have nullable types, you'll want to check them to see if they're *null*. Here's an example that shows how you can check for a *null* value:

bool isNull = startDate == null;

Console.WriteLine("isNull: " + isNull);

The example above shows that you only need to use the equals operator to check for *null*. You could also make the equality check as part of an *if* statement, like this:

int availableUnits;

if (unitsInStock == null)

{

availableUnits = 0;

}

else

{

availableUnits = (int)unitsInStock;

}

**Note:** Notice the cast operator in the else clause above. An explicit conversion is required when assigning from nullable to non-nullable types.

That's several lines of code for something that appears to be such a common operation. Fortunately, there's a better way to perform the same task, using the coalesce operator, *??*, shown below:

int availableUnits = unitsInStock ?? 0;

The coalesce operator works like this: if the first value (left hand side) is *null*, then C# evaluates the second expression (right hand side).

**Summary**

This lesson explained how nullable types can be useful in your C# applications - especially when working with values from a database. You learned how to declare a nullable type and how to assign values, both *null* and non-*null* to nullable types. Another skill you learned was how to use nullable types by checking to see if their values are *null*. As you saw, the coalesce operator can be useful to help work with nullable type variables when you need to assign a valid value to a non-nullable type.