Chapter 3

Arrays and functions in C#

Chapter 3
Arrays and functions in C#

By the end of this chapter, you should see that the difference between the two language concepts : arrays and functions , and you will be able to develop .NET applications.

3.1 Arrays

In C#, an array index starts at zero. That means, first item of an array will be stored at 0th position. The position of the last item on an array will total number of items - 1.

In C#, arrays can be declared as fixed length or dynamic. Fixed length array can stores a predefined number of items, while size of dynamic arrays increases as you add new items to the array. You can declare an array of fixed length or dynamic. You can even change a dynamic array to static after it is defined. For example, the following like declares a dynamic array of integers.

```
int [] intArray;
```

The following code declares an array, which can store 5 items starting from index 0 to 4.

```
int [] intArray;
intArray = new int[5];
```

The following code declares an array that can store 100 items starting from index 0 to 99.

```
int [] intArray;
intArray = new int[100];
```

Single Dimension Arrays

Arrays can be divided into four categories. These categories are

- single-dimensional arrays,
- multidimensional arrays or rectangular arrays,
- jagged arrays, and
- mixed arrays.

Single-dimensional arrays are the simplest form of arrays. These types of arrays are used to store number of items of a predefined type. All items in a single dimension array are stored in a row starting from 0 to the size of array -1.
In C# arrays are objects. That means declaring an array doesn't create an array. After declaring an array, you need to instantiate an array by using the "new" operator.

The following code declares a integer array, which can store 3 items. As you can see from the code, first I declare the array using [] bracket and after that I instantiate the array by calling new operator.

```
int [] intArray;
intArray = new int[3];
```

yahoo Array declarations in C# are pretty simple. You put array items in curly braces ({}). If an array is not initialized, its items are automatically initialized to the default initial value for the array type if the array is not initialized at the time it is declared. The following code declares and initializes an array of three items of integer type.

```
int [] intArray;
intArray = new int[3] {0, 1, 2};
```

The following code declares and initializes an array of 5 items.

```
string[] strArray = new string[5] {"Mohammed ", "merna", "mona",
"Amro", "Dalia"};
```

You can even direct assign these values without using the new operator.

```
string[] strArray = {"Mohammed ", "merna", "mona", "Amro",
"Dalia"};
```

You can initialize a dynamic length array as following

```
string[] strArray = new string[] {"Mohammed ", "merna", "mona",
"Amro", "Dalia"};
```

Multi Dimension Arrays

A multidimensional array is an array with more than one dimension. A multi dimension array is declared as following:

```
string[,] strArray;
```

After declaring an array, you can specify the size of array dimensions if you want a fixed size array or dynamic arrays. For example, the following code two examples create two multi dimension arrays with a matrix of 3x2 and 2x2. The first array can store 6 items and second array can store 4 items respectively.

```
int[,] numbers = new int[3, 2] { {1, 2}, {3, 4}, {5, 6} };
string[,] names = new string[2, 2] { {"mohammed ","El-dosuky"},
{"Magdy ","Rashad"} };
```

If you don't want to specify the size of arrays, just don't define a number when you call new operator. For example,

```
int[,] numbers = new int[,] { {1, 2}, {3, 4}, {5, 6} };
string[,] names = new string[,] {{"mohammed ","El-dosuky"},
{"Magdy ","Rashad"} };
```

You can also omit the new operator as we did in single dimension arrays. You can assign these values directly without using the new operator.

For example:

```
int[,] numbers = { {1, 2}, {3, 4}, {5, 6} };
string[,] siblings = { {"mohammed ","El-dosuky"}, {"mona ",
"moneer"} };
```

Jagged Arrays

Jagged arrays are often called array of arrays. An element of a jagged array itself is an array. For example, you can define an array of names of students of a class where a name itself can be an array of three strings - first name, middle name and last name. Another example of jagged arrays is an array of integers containing another array of integers. For example,

```
int[][] numArray = new int[][] { new int[] {1,3,5}, new int[]
{2,4,6,8,10} };
```

Again, you can specify the size when you call the new operator.

Mixed Arrays

Mixed arrays are a combination of multi-dimension arrays and jagged arrays. Multi-dimension arrays are also called as rectangular arrays.

Accessing Arrays using foreach Loop

The foreach control statement (loop) of C# is a new to C++ or other developers. This control statement is used to iterate through the elements of a collection such as an array. For example, the following code uses foreach loop to read all items of numArray.

```
int[] numArray = {1, 3, 5, 7, 9, 11, 13};
foreach (int num in numArray)
{
    System.Console.WriteLine(num.ToString());
}
```

A Simple Example

This sample code listed shows you how to use arrays. You can access an array items by using for loop but using foreach loop is easy to use and better.

```
using System;
namespace ArraysSamp
{
class Class1
{
static void Main(string[] args)
{
int[] intArray = new int[3];
intArray[0] = 3;
intArray[1] = 6;
intArray[2] = 9;
foreach (int i in intArray)
      Console.WriteLine(i.ToString() );

string[] strArray = new string[5]
{“Mohammed “, “merna”, “mona”, “Amro”, “Dalia”};
foreach( string str in strArray)
      Console.WriteLine(str);

string[,] names = new string[,]
     {
           {"mohammed ","El-dosuky"},
           {"mona ", "moneer"}
    };
foreach( string str in names)
        Console.WriteLine(str);
Console.ReadLine();
}
}
}
```

Understanding the Array Class

The **Array** class, defined in the System namespace, is the base class for arrays in C#. Array class is an abstract base class but it provides CreateInstance method to construct an array. The Array class provides methods for creating, manipulating, searching, and sorting arrays.

Table 1 describes Array class properties.

IsFixedSize	Return a value indicating if an array has a fixed size or not.
IsReadOnly	Returns a value indicating if an array is read-only or not.
IsSynchronized	Returns a value indicating if access to an array is thread-safe or not.
Length	Returns the total number of items in all the dimensions of an array.
Rank	Returns the number of dimensions of an array.
SyncRoot	Returns an object that can be used to synchronize access to the array.

Table 2 describes some of the Array class methods.

BinarySearch	This method searches a one-dimensional sorted **Array** for a value, using a binary search algorithm.
Clear	This method removes all items of an array and sets a range of items in the array to 0.
Clone	This method creates a shallow copy of the **Array**.
Copy	This method copies a section of one **Array** to another **Array** and performs type casting and boxing as required.

CopyTo	This method copies all the elements of the current one-dimensional **Array** to the specified one-dimensional **Array** starting at the specified destination **Array** index.
CreateInstance	This method initializes a new instance of the **Array** class.
GetEnumerator	This method returns an IEnumerator for the **Array**.
GetLength	This method returns the number of items in an **Array**.
GetLowerBound	This method returns the lower bound of an **Array**.
GetUpperBound	This method returns the upper bound of an **Array**.
GetValue	This method returns the value of the specified item in an Array.
IndexOf	This method returns the index of the first occurrence of a value in a one-dimensional **Array** or in a portion of the **Array**.
Initialize	This method initializes every item of the value-type **Array** by calling the default constructor of the value type.
LastIndexOf	This method returns the index of the last occurrence of a value in a one-dimensional **Array** or in a portion of the **Array**.
Reverse	This method reverses the order of the items in a one-dimensional **Array** or in a portion of the **Array**.
SetValue	This method sets the specified items in the current **Array** to the specified value.
Sort	This method sorts the items in one-dimensional **Array** objects.

The Array Class

Array class is an abstract base class but it provides CreateInstance method to construct an array.

```
Array names = Array.CreateInstance( typeof(String), 2, 4 );
```

After creating an array using the CreateInstance method, you can use SetValue method to add items to an array. I will discuss SetValue method later in this article.
The Array class provides methods for creating, manipulating, searching, and sorting arrays. Array class provides three boolean properties IsFixedSize, IsReadOnly, and IsSynchronized to see if an array has fixed size, read only or synchronized or not respectively. The Length property of Array class returns the number of items in an array and the Rank property returns number of dimensions in a multi-dimension array. this code creates two arrays with a fixed and variable lengths and sends the output to the system console.

```
int [] intArray;
// fixed array with 3 items
intArray = new int[3] {0, 1, 2};
// 2x2 varialbe length array
string[,] names = new string[,] {
  {"Mohammed ","El-dosuky"},
  {"Mohammed","Abo-Fotoh"} };

if(intArray.IsFixedSize)
{
Console.WriteLine("Array is fixed size");
Console.WriteLine("Size :" + intArray.Length.ToString());
}
if(names.IsFixedSize)
{
Console.WriteLine("Array is varialbe.");
Console.WriteLine("Size :" + names.Length.ToString());
Console.WriteLine("Rank :" + names.Rank.ToString());
}
```

Besides these properties, the Array class provides methods to add, insert, delete, copy, binary search, reverse, reverse and so on.

Searching an Item in an Array

The BinarySearch static method of Array class can be used to search for an item in a array. This method uses binary search algorithm to search for an item. The method takes at least two parameters - an array and an object (the item you are looking for). If an item found in an array, the method returns the index of the item (based on first item as 0th item), else method returns a negative value. Listing uses BinarySearch method to search two arrays.

```
int [] intArray = new int[3] {0, 1, 2};
string[] names = new string[]
        {“Mohammed “, “merna”, “mona”, “Amro”, “Dalia”};
object obj1 = "Mohammed";
object obj2 = 1;
int retVal = Array.BinarySearch(names, obj1);
if(retVal >=0)
Console.WriteLine("Item index " +retVal.ToString() );
else
Console.WriteLine("Item not found");
retVal = Array.BinarySearch(intArray, obj2);
if(retVal >=0)
Console.WriteLine("Item index " +retVal.ToString() );
else
Console.WriteLine("Item not found");
```

Sorting Items in an Array

The Sort static method of the Array class can be used to sort an array items. This method has many overloaded forms. The simplest form takes a parameter of the array, you want to sort to. Listing 3 uses Sort method to sort an array items.

```
string[] names = new string[]
            {“Mohammed “, “merna”, “mona”, “Amro”, “Dalia”};
```

```
Console.WriteLine("Original Array:");
foreach (string str in names)
{
Console.WriteLine(str);
}
Console.WriteLine("Sorted Array:");
Array.Sort(names);
foreach (string str in names)
{
Console.WriteLine(str);
}
```

Getting and Setting Values

The GetValue and SetValue methods of the Array class can be used to return a value from an index of an array and set values of an array item at a specified index respectively. The code listed creates a 2-dimension array instance using the CreateInstance method. After that I use SetValue method to add values to the array.

In the end I find number of items in both dimensions and use GetValue method to read values and display on the console.

```
Array names = Array.CreateInstance( typeof(String), 2, 4 );
names.SetValue( "Rosy", 0, 0 );
names.SetValue( "Amy", 0, 1 );
names.SetValue( "Peter", 0, 2 );
names.SetValue( "Albert", 0, 3 );
names.SetValue( "Mel", 1, 0 );
names.SetValue( "Mongee", 1, 1 );
names.SetValue( "Luma", 1, 2 );
names.SetValue( "Lara", 1, 3 );
int items1 = names.GetLength(0);
int items2 = names.GetLength(1);
for ( int i =0; i < items1; i++ )
for ( int j = 0; j < items2; j++ )
Console.WriteLine(i.ToString() +","+ j.ToString() +": "
+names.GetValue( i, j ) );
```

Other Methods of Array Class

The Reverse static method of the Array class reverses the order of items in a array. Similar to the sort method, you can just pass an array as a parameter of the Reverse method.

```
Array.Reverse(names);
```

The Clear static method of the Array class removes all items of an array and sets its length to zero. This method takes three parameters - first an array object, second starting index of the array and third is number of elements. The following code removes two elements from the array starting at index 1 (means second element of the array).

```
Array.Clear(names, 1, 2);
```

The GetLength method returns the number of items in an array. The GetLowerBound and GetUppperBound methods return the lower and upper bounds of an array respectively. All these three methods take at least a parameter, which is the index of the dimension of an array. The following code snippet uses all three methods.

```
string[] names = new string[]
        {“Mohammed “, “merna”, “mona”, “Amro”, “Dalia”};
Console.WriteLine(names.GetLength(0).ToString());
Console.WriteLine(names.GetLowerBound(0).ToString());
Console.WriteLine(names.GetUpperBound(0).ToString());
```

The Copy static method of the Array class copies a section of an array to another array. The CopyTo method copies all the elements of an array to another one-dimension array. The code listed copies contents of an integer array to an array of object types.

```
// Creates and initializes a new Array of type Int32.
Array oddArray = Array.CreateInstance(
Type.GetType("System.Int32"), 5 );
oddArray.SetValue(1, 0);
oddArray.SetValue(3, 1);
oddArray.SetValue(5, 2);
oddArray.SetValue(7, 3);
oddArray.SetValue(9, 4);
// Creates and initializes a new Array of type Object.
Array objArray = Array.CreateInstance(
Type.GetType("System.Object"), 5 );
Array.Copy(oddArray, oddArray.GetLowerBound(0), objArray,
objArray.GetLowerBound(0), 4 );
int items1 = objArray.GetUpperBound(0);
for ( int i =0; i < items1; i++ )
Console.WriteLine(objArray.GetValue(i).ToString());
```

You can even copy a part of an array to another array by passing number of items and starting item in the Copy method. The following format copies a range of items from an Array starting at the specified source index and pastes them to another **Array** starting at the specified destination index.

```
public static void Copy(Array, int, Array, int, int);
```

3.2 Functions

The basic philosophy of function is divide and conquer by which a complicated tasks are successively divided into simpler and more manageable tasks which can be easily handled. A program can be divided into smaller subprograms that can be developed and tested successfully.

A function is a complete and independent program which is used (or invoked) by the main program or other subprograms. A subprogram receives values called arguments from a calling program, does calculations and returns results to the program.

There are many advantages in using functions in a program:

1. It facilitates top down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower level functions is addressed later.

2. the length of the source program can be reduced by using functions at appropriate places. This factor is critical with microcomputers where memory space is limited.

3. It is easy to locate and isolate a faulty function for further investigation.

4. A function may be used by many other programs this means that a c programmer can build on what others have already done, instead of starting over from scratch.

5. A program can be used to avoid rewriting the same sequence of code at two or more locations in a program. This is especially useful if the code involved is long or complicated.

6. Programming teams does a large percentage of programming. If the program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program

We already know that C# support the use of library functions and use defined functions. The library functions are used to carry out a number of commonly used operations or calculations. The user-defined functions are written by the programmer to carry out various individual tasks.

Functions are used in c for the following reasons:

1. Many programs require that a specific function is repeated many times instead of writing the function code as many timers as it is required we can write it as a single function and access the same function again and again as many times as it is required.

2. We can avoid writing redundant program code of some instructions again and again.

3. Programs with using functions are compact & easy to understand.

4. Testing and correcting errors is easy because errors are localized and corrected.

5. We can understand the flow of program, and its code easily since the readability is enhanced while using the functions.

6. A single function written in a program can also be used in other programs also.

Function definition:

```
[ data type] function name (argument list)
argument declaration;
{
local variable declarations;
statements;
[return expression]
}
```

Example :

```
int mul(int a, int b)
{
int y;
y=a+b;
return y;
}
```

When the value of y which is the addition of the values of a and b. the last two statements ie, y=a+b; can be combined as
return(y)
return(a+b);

Types of functions:
A function may belong to any one of the following categories:
1. Functions with no arguments and no return values.
2. Functions with arguments and no return values.
3. Functions with arguments and return values.

Functions with no arguments and no return values:
Let us consider the following program

```
/* Program to illustrate a function with no argument and no return
values*/

public static void main()
{
staetemtn1();
starline();
statement2();
starline();
}
```

```
/*function to print a message*/
void statement1()
{
Console.WriteLine("\n Sample subprogram output");
}

void statement2()
{
Console.WriteLine("\n Sample subprogram output two");
}

void starline()
{
int a;
for (a=1;a<60;a++)
Console.WriteLine("*");
}
```

In the above example there is no data transfer between the calling function and the called function. When a function has no arguments it does not receive any data from the calling function. Similarly when it does not return value the calling function does not receive any data from the called function. A function that does not return any value cannot be used in an expression it can be used only as independent statement.

Functions with arguments but no return values:
The nature of data communication between the calling function and the arguments to the called function and the called function does not return any values to the calling function this shown in example below:

Consider the following: Function calls containing appropriate arguments. For example the function call

```
value (500,0.12,5)
```

Would send the values 500,0.12 and 5 to the function value (p, r, n) and assign values 500 to p, 0.12 to r and 5 to n. the values 500,0.12 and 5 are the actual arguments which become the values of the formal arguments inside the called function.

Both the arguments actual and formal should match in number type and order. The values of actual arguments are assigned to formal arguments on a one to one basis starting with the first argument as shown below:

```
public static void main()
{
function1(a1,a2,a3……an)
}

void function1(f1,f2,f3….fn);
{
function body;
}
```

here a1,a2,a3 are actual arguments and f1,f2,f3 are formal arguments.

The no of formal arguments and actual arguments must be matching to each other suppose if actual arguments are more than the formal arguments, the extra actual arguments are discarded. If the number of actual arguments are less than the formal arguments then the unmatched formal arguments are initialized to some garbage values. In both cases no error message will be generated.

The formal arguments may be valid variable names, the actual arguments may be variable names expressions or constants. The values used in actual arguments must be assigned values before the function call is made.

When a function call is made only a copy of the values actual arguments is passed to the called function. What occurs inside the functions will have no effect on the variables used in the actual argument list.

Let us consider the following program

```
/*Program to find the largest of two numbers using function*/
main()
{
int a,b;
Console.WriteLine("Enter the two numbers");

a = int.Parse(Console.ReadLine());
b= int.Parse(Console.ReadLine());

largest(a,b)
}

/*Function to find the largest of two numbers*/
void largest(int a, int b)
{
if(a>b)
Console.WriteLine("Largest element={0}",a);
else
Console.WriteLine("Largest element={0}",b);
}
```

in the above program we could make the calling function to read the data from the terminal and pass it on to the called function. But function foes not return any value.

Functions with arguments and return values:
The function of the type Arguments with return values will send arguments from the calling function to the called function and expects the result to be returned back from the called function back to the calling function.

To assure a high degree of portability between programs a function should generally be coded without involving any input output operations. For example different programs may require different output formats for displaying the results. Theses shortcomings can be overcome by handing over the result of a function to its calling function where the returned value can be used as required by the program. In the above type of function the following steps are carried out:

1. The function call transfers the controls along with copies of the values of the actual arguments of the particular function where the formal arguments are creates and assigned memory space and are given the values of the actual arguments.

2. The called function is executed line by line in normal fashion until the return statement is encountered. The return value is passed back to the function call is called function.

3. The calling statement is executed normally and return value is thus assigned to the calling function.

Note that the value return by any function when no format is specified is an integer.

Return value data type of function:
A C# function returns a value of type int as the default data type when no other type is specified explicitly. For example if function does all the calculations by using float values and if the return statement such as return (sum); returns only the integer part of the sum. This is since we have not specified any return type for the sum. There is the necessity in some cases it is important to receive float or character or double data type. To enable a calling function to receive a non-integer value from a called function we can do the two things:
1. The explicit type specifier corresponding to the data type required must be mentioned in the function header. The general form of the function definition is

```
Type_specifier function_name(argument list)
Argument declaration;
{
function statement;
}
```

The type specifier tells the compiler, the type of data the function is to return.

2. The called function must be declared at the start of the body in the calling function, like any other variable. This is to tell the calling function the type of data the function is actually returning. The program given below illustrates the transfer of a floating-point value between functions done in a multiple function program.

```
public static void main()
{
float x,y;
x=12.345;
y=9.82;
Console.WriteLine(add(x,y));
Console.WriteLine(sub(x,y);
}

float add(float a, float b)
{
        return(a+b);
}
double sub(double p, double q)
{
        return(p-q);
}
```

We can notice that the functions too are declared along with the variables. These declarations clarify to the compiler that the return type of the function add is float and sub is double.

Void functions:

The functions that do not return any values can be explicitly defined as void. This prevents any accidental use of these functions in expressions.

Using ref and out Parameter

When we pass a parameter as **ref** to a method, the method refers to the same variable and changes made will affect the actual variable.Even the variable passed as **out** parameter is similar to **ref**, but there are few implementation differences when you use it in **C#** .

Argument passed as **ref** must be initialized before it is passed to the method, where as in case of out its is not necessary,but after a call to the method as an out parameter the variable must be initialized.

When to use **ref** and **out** parameter. out parameter can be used when we want to return more than one value from a method.

IMPORTANT NOTE : We now know what are ref and out parameters, but these are only for C#(these are only understood by csc Compiler) when looking inside the IL Code there is no difference whether you use ref or out parameters. The implementation of ref and out parameter in IL Code is same.

When Calling a method and in the method signature after the datatype of the parameter a & sign is used, indicating the address of the variable.

Source Code:

```
using System;
class RefOut
{
public static void Main(String [] args)
{
int a = 0,b,c=0,d=0;
/*
a is normal parameter will not affect the changes after the function
call ,

b is out parameter will affect the changes after the function call but
not necessary to initialize the variable b but should be initialized in
the function ParamTest

c is ref parameter will affect the changes after the function call and
is compulsory to initialize the variable c before calling the function
ParamTest
*/

Console.WriteLine("a is normal parameter ");
Console.WriteLine("b is out parameter ");
Console.WriteLine("c is ref parametert");
Console.WriteLine("d is used to store the return value");

d=ParamTest(a,out b,ref c);
Console.WriteLine("a = {0}", a);
Console.WriteLine("b = {0}", b);
Console.WriteLine("c = {0}", c);
Console.WriteLine("d = {0}", d);
}
public static int ParamTest(int a, out int b, ref int c)
{
a=10;
b=20;
c=30;
return 40;
}
}
```

3.3 Delegates

All of us have been exposed to **event driven** programming of some sort or the other. C# adds on value to the often mentioned world of event driven programming by adding support through **events and delegates**. The emphasis of this article would be to identify what exactly happens when you add an event handler to your common UI controls. A simple simulation of what could possibly be going on behind the scenes when the AddOnClick or any similar event is added to the Button class will be explained. This will help you understand better the nature of event handling using multi cast delegates.

Delegates

A delegate in C# is similar to a function pointer in C or C++. Using a delegate allows the programmer to encapsulate a reference to a method inside a delegate object. The delegate object can then be passed to code which can call the referenced method, without having to know at compile time which method will be invoked.

Call a Function directly - No Delegate

In most cases, when we call a function, we specify the function to be called directly. If the class MyClass has a function named Process, we'd normally call it like this (SimpleSample.cs):

```
using System;

namespace ElDosuky.NoDelegate
{
   public class MyClass
   {
      public void Process()
      {
         Console.WriteLine("Process() begin");
         Console.WriteLine("Process() end");
      }
   }
```

```
  public class Test
    {
       static void Main(string[] args)
       {
          MyClass myClass = new MyClass();
          myClass.Process();
       }
    }
}
```

That works well in most situations. Sometimes, however, we don't want to call a function directly - we'd like to be able to pass it to somebody else so that they can call it. This is especially useful in an event-driven system such as a graphical user interface, when I want some code to be executed when the user clicks on a button, or when I want to log some information but can't specify how it is logged.

The very basic Delegate

An interesting and useful property of a delegate is that it does not know or care about the class of the object that it references. Any object will do; all that matters is that the method's argument types and return type match the delegate's. This makes delegates perfectly suited for "anonymous" invocation.

The signature of a single cast delegate is shown below:

delegate result-type identifier ([parameters]);

where:

result-type: The result type, which matches the return type of the function.
identifier: The delegate name.
parameters: The Parameters, that the function takes.

Examples:

public delegate void SimpleDelegate ()
This declaration defines a delegate named SimpleDelegate, which will encapsulate any method that takes
no parameters and returns no value.

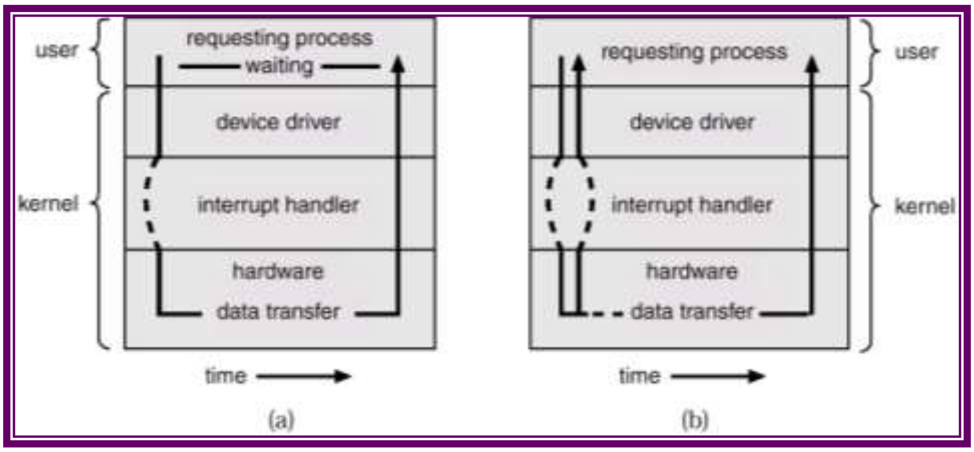
public delegate int ButtonClickHandler (object obj1, object obj2)
This declaration defines a delegate named ButtonClickHandler, which will encapsulate any method that takes
two objects as parameters and returns an int.

A delegate will allow us to specify what the function we'll be calling *looks like* without having to specify *which* function to call. The declaration for a delegate looks just like the declaration for a function, except that in this case, we're declaring the signature of functions that this delegate can reference.

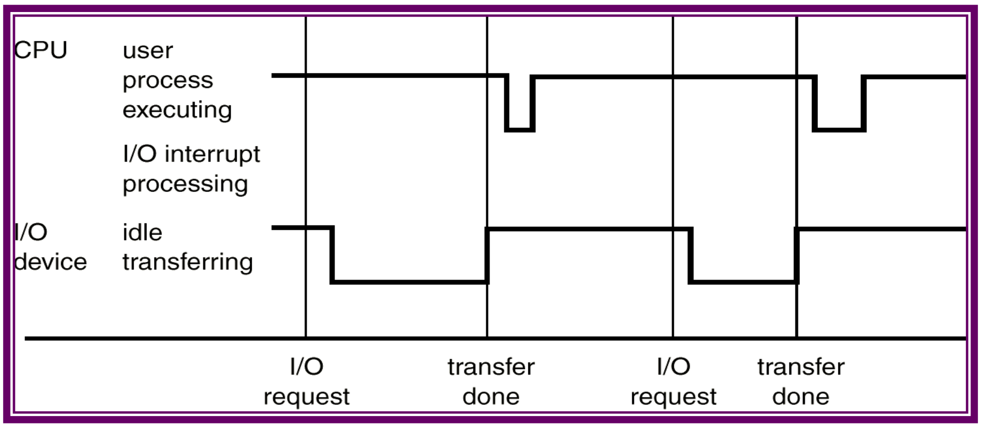
There are three steps in defining and using delegates:
Declaration , Instantiation , and Invocation

```
using System;
namespace ElDosuky.BasicDelegate
{
   // Declaration
   public delegate void SimpleDelegate();
   class TestDelegate
   {
      public static void MyFunc()
      {
         Console.WriteLine("I was called by delegate ...");
      }
      public static void Main()
      {
                   SimpleDelegate simpleDelegate =
                      new SimpleDelegate(MyFunc); //Instantiation

               simpleDelegate();  // Invocation
      }
   }
}
```

Two I/O Methods



Interrupt Time Line For a Single Process Doing Output



After I/O starts, control returns to user program only upon I/O completion. Wait instruction idles the CPU until the next interrupt Wait loop (contention for memory access). At most one I/O request is outstanding at a time, no simultaneous I/O processing.

Calling Static Functions

declares a delegate that takes a single string parameter and has no return type:

```
using System;

namespace ElDosuky.SimpleDelegate
{
   // Delegate Specification
   public class MyClass
   {
      // Declare a delegate that takes a single string parameter
      // and has no return type.
      public delegate void LogHandler(string message);

      // The use of the delegate is just like calling a function directly,
      // though we need to add a check to see if the delegate is null
      // (that is, not pointing to a function) before calling the
function.
      public void Process(LogHandler logHandler)
      {
         if (logHandler != null)
         {
            logHandler("Process() begin");
         }

         if (logHandler != null)
         {
            logHandler ("Process() end");
         }
      }
   }
```

```
 // Test Application to use the defined Delegate
   public class TestApplication
   {
      // Static Function: To which is used in the Delegate.
      // To call the Process() function,
      // we need to declare a logging function: Logger() that matches
      // the signature of the delegate.

      static void Logger(string s)
      {
         Console.WriteLine(s);
      }

      static void Main(string[] args)
      {
         MyClass myClass = new MyClass();

         // Create instance of delegate, pointing to logging function.
         // This delegate will be passed to the Process() function.

         MyClass.LogHandler myLogger =
                 new MyClass.LogHandler(Logger);
         myClass.Process(myLogger);
      }
   }
}
```

Calling Member Functions

In the simple example above, the **Logger( )** function merely writes the string out. A different function might want to log the information to a file, but to do this, the function needs to know what file to write the information

```
using System;
using System.IO;

namespace ElDosuky.SimpleDelegate
{
   // Delegate Specification
   public class MyClass
   {
      // Declare a delegate that takes a single string parameter
      // and has no return type.
      public delegate void LogHandler(string message);

      // The use of the delegate is just like calling a function directly,
      // though we need to add a check to see if the delegate is null
      // (that is, not pointing to a function) before calling function.

      public void Process(LogHandler logHandler)
      {
         if (logHandler != null)
         {
            logHandler("Process() begin");
         }

         if (logHandler != null)
         {
            logHandler ("Process() end");
         }
      }
   }

   // The FileLogger class merely encapsulates the file I/O
   public class FileLogger
   {
      FileStream fileStream;
      StreamWriter streamWriter;
```

```
 // Constructor
        public FileLogger(string filename)
        {
            fileStream = new FileStream(filename, FileMode.Create);
            streamWriter = new StreamWriter(fileStream);
        }

        // Member Function which is used in the Delegate
        public void Logger(string s)
        {
            streamWriter.WriteLine(s);
        }
        public void Close()
        {
            streamWriter.Close();
            fileStream.Close();
        }
    }
    // Main() is modified so that the delegate points to the Logger()
    // function on the fl instance of a FileLogger. When this delegate
    // is invoked from Process(), the member function is called and
    // the string is logged to the appropriate file.
    public class TestApplication
    {
        static void Main(string[] args)
        {
            FileLogger fl = new FileLogger("process.log");
            MyClass myClass = new MyClass();
            // Crate an instance of the delegate, pointing to the Logger()
            // function on the fl instance of a FileLogger.
            MyClass.LogHandler myLogger =
                        new MyClass.LogHandler(fl.Logger);
            myClass.Process(myLogger);
            fl.Close();
        }
    }
}
```

The cool part here is that we didn't have to change the Process() function; the code to all the delegate is the same regardless of whether it refers to a **static** or **member** function.

Multicasting

Being able to point to member functions is nice, but there are more tricks you can do with delegates. In C#, delegates are ***multicast***, which means that they can **point to more than one function at a time** (that is, they're based off the System.MulticastDelegate type). A multicast delegate maintains a list of functions that will all be called when the delegate is invoked. We can add back in the logging function from the first example, and call both delegates. Here's what the code looks like:

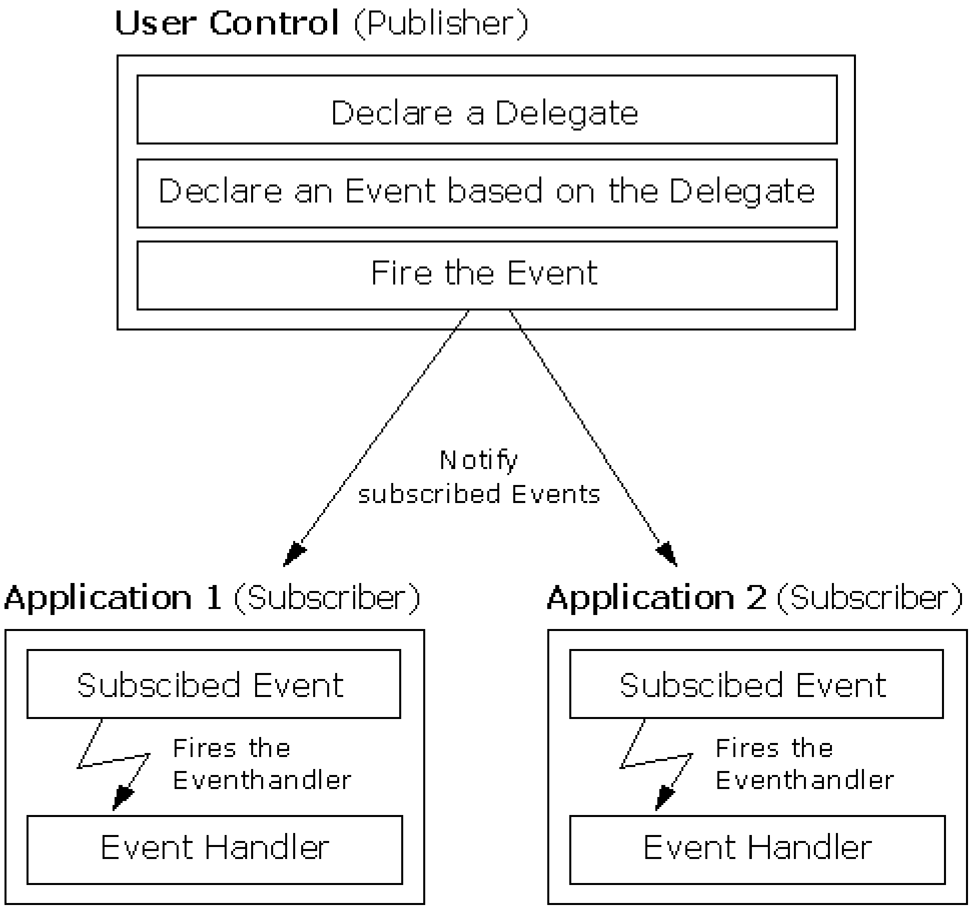
```
using System;
using System.IO;

namespace ElDosuky.SimpleDelegate
{
   // Delegate Specification
   public class MyClass
   {
      // Declare a delegate that takes a single string parameter
      // and has no return type.
      public delegate void LogHandler(string message);

      // The use of the delegate is just like calling a function directly,
      // though we need to add a check to see if the delegate is null
      // (that is, not pointing to a function) before calling function.

public void Process(LogHandler logHandler)
      {
         if (logHandler != null)
         {
            logHandler("Process() begin");
         }
```

```
        if (logHandler != null)
        {
            logHandler ("Process() end");
        }
    }
}

// The FileLogger class merely encapsulates the file I/O
public class FileLogger
{
    FileStream fileStream;
    StreamWriter streamWriter;

    // Constructor
    public FileLogger(string filename)
    {
        fileStream = new FileStream(filename, FileMode.Create);
        streamWriter = new StreamWriter(fileStream);
    }

    // Member Function which is used in the Delegate
    public void Logger(string s)
    {
        streamWriter.WriteLine(s);
    }

    public void Close()
    {
        streamWriter.Close();
        fileStream.Close();
    }
}
```

```
// Test Application which calls both Delegates
    public class TestApplication
    {
         // Static Function which is used in the Delegate
        static void Logger(string s)
        {
            Console.WriteLine(s);
        }

        static void Main(string[] args)
        {
            FileLogger fl = new FileLogger("process.log");

            MyClass myClass = new MyClass();

         // Crate an instance of the delegates, pointing to the static
       // Logger() function defined in the TestApplication class
// then to member function on the fl instance of a FileLogger.
            MyClass.LogHandler myLogger = null;
            myLogger += new MyClass.LogHandler(Logger);
            myLogger += new MyClass.LogHandler(fl.Logger);

            myClass.Process(myLogger);
            fl.Close();
        }
    }
}
```

Events
The Event model in C# finds its roots in the event programming model that is popular in asynchronous programming. The basic foundation behind this programming model is the idea of "publisher and subscribers." In this model, you have ***publishers*** who will do some logic and publish an "event." Publishers will then send out their event only to ***subscribers*** who have subscribed to receive the specific event.

In C#, any object can *publish* a set of events to which other applications can *subscribe*. When the publishing class raises an event, all the subscribed applications are notified. The following figure shows this mechanism.



The following important conventions are used with events:

Event Handlers in the .NET Framework return void and take two parameters.

The first paramter is the source of the event; that is the publishing object.

The second parameter is an object derived from EventArgs.

Events are properties of the class publishing the event.

The keyword event controls how the event property is accessed by the subscribing classes.

Simple Event

Let's modify our logging example from above to use an event rather than a delegate:

```
using System;
using System.IO;

namespace ElDosuky.SimpleEvent
{
   /* ========= Publisher of the Event ============== */
   public class MyClass
   {
      // Define a delegate named LogHandler, which encapsulate
      // any method that takes a string as parameter and returns none
      public delegate void LogHandler(string message);

      // Define an Event based on the above Delegate
      public event LogHandler Log;

      // Instead of having the Process() function take a delegate
      // as a parameter, we've declared a Log event. Call the Event,
      // using the OnX Method, where X is the name of the Event.
      public void Process()
      {
         OnLog("Process() begin");
         OnLog("Process() end");
      }

      // By Default, create an OnX Method, to call the Event
      protected void OnLog(string message)
      {
         if (Log != null)
         {
            Log(message);
         }
      }
   }
```

```
// The FileLogger class merely encapsulates the file I/O
public class FileLogger
{
   FileStream fileStream;
   StreamWriter streamWriter;

   // Constructor
   public FileLogger(string filename)
   {
      fileStream = new FileStream(filename, FileMode.Create);
      streamWriter = new StreamWriter(fileStream);
   }

   // Member Function which is used in the Delegate
   public void Logger(string s)
   {
      streamWriter.WriteLine(s);
   }

   public void Close()
   {
      streamWriter.Close();
      fileStream.Close();
   }
}

/* ========= Subscriber of the Event ============== */
// It's now easier and cleaner to merely add instances
// of the delegate to the event, instead of having to
// manage things ourselves
public class TestApplication
{
   static void Logger(string s)
   {
      Console.WriteLine(s);
   }
```

```
        static void Main(string[] args)
        {
            FileLogger fl = new FileLogger("process.log");
            MyClass myClass = new MyClass();

            // Subscribe the Functions Logger and fl.Logger
            myClass.Log += new MyClass.LogHandler(Logger);
            myClass.Log += new MyClass.LogHandler(fl.Logger);

            // The Event will now be triggered in the Process() Method
            myClass.Process();

            fl.Close();
        }
    }
}
```

The Second Change Event Example
Suppose you want to create a Clock class that uses events to notify potential subscribers whenever the local time changes value by one second. Here is the complete, documented example:

```
using System;
using System.Threading;

namespace SecondChangeEvent
{
     // Our subject -- it is this class that other classes
   // will observe. This class publishes one event:
   // SecondChange. The observers subscribe to that event.
   public class Clock
   {
      // Private Fields holding the hour, minute and second
      private int _hour;
      private int _minute;
      private int _second;
```

```
// delegate named SecondChangeHandler, which encapsulate
// any method that takes a clock object and TimeInfoEventArgs
// object as the parameter and returns no value. It's the
// delegate the subscribers must implement.
public delegate void SecondChangeHandler (
  object clock,
  TimeInfoEventArgs timeInformation
);

// The event we publish
public event SecondChangeHandler SecondChange;

// The method which fires the Event
protected void OnSecondChange(
  object clock,
  TimeInfoEventArgs timeInformation
)
{
  // Check if there are any Subscribers
  if (SecondChange != null)
  {
    // Call the Event
    SecondChange(clock,timeInformation);
  }
}

// Set the clock running, it will raise an
// event for each new second
public void Run()
{
  for(;;)
  {
    // Sleep 1 Second
    Thread.Sleep(1000);

    // Get the current time
    System.DateTime dt = System.DateTime.Now;
```

```
        // If the second has changed
        // notify the subscribers
        if (dt.Second != _second)
        {
          // Create the TimeInfoEventArgs object
          // to pass to the subscribers
          TimeInfoEventArgs timeInformation =
            new TimeInfoEventArgs(
            dt.Hour,dt.Minute,dt.Second);

          // If anyone has subscribed, notify them
          OnSecondChange (this,timeInformation);
        }
        // update the state
        _second = dt.Second;
        _minute = dt.Minute;
        _hour = dt.Hour;

      }
    }
  }
  // The class to hold the information about the event
  // in this case it will hold only information
  // available in the clock class, but could hold
  // additional state information
  public class TimeInfoEventArgs : EventArgs
  {
    public TimeInfoEventArgs(int hour, int minute, int second)
    {
      this.hour = hour;
      this.minute = minute;
      this.second = second;
    }
    public readonly int hour;
    public readonly int minute;
    public readonly int second;
  }
```

```
// An observer. DisplayClock subscribes to the
// clock's events. The job of DisplayClock is
// to display the current time
public class DisplayClock
{
  // Given a clock, subscribe to
  // its SecondChangeHandler event
  public void Subscribe(Clock theClock)
  {
    theClock.SecondChange +=
      new Clock.SecondChangeHandler(TimeHasChanged);
  }

  // The method that implements the
  // delegated functionality
  public void TimeHasChanged(
    object theClock, TimeInfoEventArgs ti)
  {
    Console.WriteLine("Current Time: {0}:{1}:{2}",
      ti.hour.ToString(),
      ti.minute.ToString(),
      ti.second.ToString());
  }
}

// A second subscriber whose job is to write to a file
public class LogClock
{
  public void Subscribe(Clock theClock)
  {
    theClock.SecondChange +=
      new Clock.SecondChangeHandler(WriteLogEntry);
  }

  // This method should write to a file
  // we write to the console to see the effect
  // this object keeps no state
```

```
    public void WriteLogEntry(
      object theClock, TimeInfoEventArgs ti)
    {
      Console.WriteLine("Logging to file: {0}:{1}:{2}",
        ti.hour.ToString(),
        ti.minute.ToString(),
        ti.second.ToString());
    }
  }

  // Test Application which implements the
  // Clock Notifier - Subscriber Sample
  public class Test
  {
    public static void Main()
    {
      // Create a new clock
      Clock theClock = new Clock();

      // Create the display and tell it to
      // subscribe to the clock just created
      DisplayClock dc = new DisplayClock();
      dc.Subscribe(theClock);

      // Create a Log object and tell it
      // to subscribe to the clock
      LogClock lc = new LogClock();
      lc.Subscribe(theClock);

      // Get the clock started
      theClock.Run();
    }
  }
}
```

3.4 Lambda Expressions and Anonymous functions

The Lambda Calculus

In the semantics we will develop, logical forms for sentences will be expressions in first-order logic that encode propositions. For instance, the sentences “shrdlu halts” and “Every student wrote a program” will be associated with the first-order logic expressions:

$$halts(shrdlu) \text{ and}$$
$$\forall s\, student(s) \Rightarrow \exists p(program(p) \& wrote(s, p)).$$

We again emphasize the differences between the FOL notation we use here and the notation for Prolog. The differences help to carefully distinguish the abstract notion of logical forms from the particular encoding of them in Prolog. FOL formulas will later be encoded in Prolog not as Prolog formulas, but as terms, because our programs are treating the formulas as *data*.

Most intermediate (i.e., nonsentential) logical forms in the grammar do not encode whole propositions, but rather, propositions with certain parts missing. For instance, a verb phrase logical form will typically be a proposition parameterized by one of the entities in the situation described by the proposition. That is, the VP logical form can be seen as a *function* from entities to propositions, what is often called a *property* of entities. In first-order logic, functions can only be specified with function symbols. We must have one such symbol for each function that will ever be used. Because arbitrary numbers of functions might be needed as intermediate logical forms for phrases, we will relax the “one symbol per function” constraint by extending the language of FOL with a special function-forming operator.

The lambda calculus allows us to specify functions by describing them in terms of combinations of other functions. For instance, consider the function from an integer, call it x, to the integer $x + 1$.

We would like to specify this function without having to give it a name. The expression x+1 seems to have all the information needed to pick out which function we want, except that it does not specify what in the expression marks the argument of the function.

This problem may not seem especially commanding in the case of the function $x + 1$, but consider the function specified by expression $x + y$, it becomes clear that we must be able to distinguish the function that takes an integer x onto the sum of that integer and y from the function that takes an integer y onto the sum of it and x.

Therefore, to pick out which variable is marking the argument of the function, we introduce a new symbol "λ" into the logical language (hence the name "lambda calculus"). To specify a function, we will use the notation

$$\lambda x.\varphi ,$$

where x is the variable marking the argument of the function and φ is the expression defining the value of the function at that argument. Thus we can specify the successor function as $\lambda x.x + 1$ and the two incrementing functions can be distinguished as $\lambda x.x + y$ and $\lambda y.x + y$. We will allow these *lambda expressions* anywhere a functor would be allowed. For instance, the following is a well-formed lambda calculus term:

$$(\lambda x.x + 1)(3) .$$

Intuitively, such a function application expression should be semantically identical to the expression 3 + 1. The formal operation called (for historical reasons) β-*reduction* codifies this intuition.

The rule of β-reduction says that any expression of the form

$$(\lambda x.\varphi)a$$

can be reduced to the (semantically equivalent) expression

$$[\varphi]\{x = a\} ,$$

The expression φ with all occurrences of x replaced with a. We ignore here and in the sequel the problem of renaming of variables, so-called α-*conversion*, to avoid capture of free variables.

A *lambda expression* is an anonymous function that can contain expressions and statements, and can be used to create delegates or expression tree types.
All lambda expressions use the lambda operator =>, which is read as "goes to". The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression or statement block. The lambda expression x => x * x is read "x goes to x times x." This expression can be assigned to a delegate type as follows:
C#

```
delegate int del(int i);
static void Main(string[] args)
{
   del myDelegate = x => x * x;
   int j = myDelegate(5); //j = 25
}
```

To create an expression tree type:

```
using System.Linq.Expressions;

namespace ConsoleApplication1
{
   class Program
   {
      static void Main(string[] args)
      {
         Expression<del> myET = x => x * x;
      }
   }
}
```

The => operator has the same precedence as assignment (=) and is right-associative.

Lambdas are used in method-based LINQ queries as arguments to standard query operator methods such as Where.

When you use method-based syntax to call the Where method in the Enumerable class (as you do in LINQ to Objects and LINQ to XML) the parameter is a delegate type System..::.Func<(Of <(T, TResult>)>). A lambda expression is the most convenient way to create that delegate. When you call the same method in, for example, the System.Linq..::.Queryable class (as you do in LINQ to SQL) then the parameter type is an System.Linq.Expressions..::.Expression<Func> where Func is any Func delegates with up to five input parameters. Again, a lambda expression is just a very concise way to construct that expression tree. The lambdas allow the **Where** calls to look similar although in fact the type of object created from the lambda is different.
In the previous example, notice that the delegate signature has one implicitly-typed input parameter of type **int**, and returns an **int**. The lambda expression can be converted to a delegate of that type because it also has one input parameter (x) and a return value that the compiler can implicitly convert to type **int**. (Type inference is discussed in more detail in the following sections.) When the delegate is invoked by using an input parameter of 5, it returns a result of 25.

Lambdas are not allowed on the left side of the is or as operator.
All restrictions that apply to anonymous methods also apply to lambda expressions.

Expression Lambdas

A lambda expression with an expression on the right side is called an *expression lambda*. Expression lambdas are used extensively in the construction of Expression Trees. An expression lambda returns the result of the expression and takes the following basic form:

```
(input parameters) => expression
```

The parentheses are optional only if the lambda has one input parameter; otherwise they are required. Two or more input parameters are separated by commas enclosed in parentheses:

```
(x, y) => x == y
```

Sometimes it is difficult or impossible for the compiler to infer the input types. When this occurs, you can specify the types explicitly as shown in the following example:

```
(int x, string s) => s.Length > x
```

Specify zero input parameters with empty parentheses:

```
() => SomeMethod()
```

Note in the previous example that the body of an expression lambda can consist of a method call. However, if you are creating expression trees that will be consumed in another domain, such as SQL Server, you should not use method calls in lambda expressions. The methods will have no meaning outside the context of the .NET common language runtime.

Statement Lambdas

A statement lambda resembles an expression lambda except that the statement(s) is enclosed in braces:

```
(input parameters) => {statement;}
```

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

```
delegate void TestDelegate(string s);
TestDelegate myDel = n => { string s = n + " " + "World";
Console.WriteLine(s); };
myDel("Hello");
```

Statement lambdas, like anonymous methods, cannot be used to create expression trees.

Lambdas with the Standard Query Operators

Many Standard query operators have an input parameter whose type is one of the Func<(Of <(T, TResult>)>) family of generic delegates. The Func<(Of <(T, TResult>)>) delegates use type parameters to define the number and type of input parameters, and the return type of the delegate. **Func** delegates are very useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the following delegate type:

```
public delegate TResult Func<TArg0, TResult>(TArg0 arg0)
```

The delegate can be instantiated as Func<int,bool> myFunc where **int** is an input parameter and **bool** is the return value. The return value is always specified in the last type parameter. **Func<int, string, bool>** defines a delegate with two input parameters, **int** and **string**, and a return type of **bool**. The following **Func** delegate, when it is invoked, will return true or false to indicate whether the input parameter is equal to 5:

```
   Func<int, bool> myFunc = x => x == 5;
   bool result = myFunc(4); // returns false of course
```

You can also supply a lambda expression when the argument type is an **Expression<Func>**, for example in the standard query operators that are defined in System.Linq.Queryable. When you specify an **Expression<Func>** argument, the lambda will be compiled to an expression tree.
A standard query operator, the Count method, is shown here:

```
   int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
   int oddNumbers = numbers.Count(n => n % 2 == 1);
```

The compiler can infer the type of the input parameter, or you can also specify it explicitly. This particular lambda expression counts those integers (n) which when divided by two have a remainder of 1.

The following method will produce a sequence that contains all the elements in the numbers array that are to the left of the “9” because that is the first number in the sequence that does not meet the condition:

```
   var firstNumbersLessThan6 = numbers.TakeWhile(n => n < 6);
```

This example shows how to specify multiple input parameters by enclosing them in parentheses. The method returns all the elements in the numbers array until a number is encountered whose value is less than its position. Do not confuse the lambda operator (=>) with the greater than or equal operator (>=).

```
   var firstSmallNumbers = numbers.TakeWhile((n, index) => n >=
index);
```

Type Inference in Lambdas
When writing lambdas, you often do not have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the underlying delegate type, and other factors as described in the C# 3.0 Language Specification. For most of the standard query operators, the first input is the type of the elements in the source sequence. So if you are querying an **IEnumerable<Customer>**, then the input variable is inferred to be a Customer object, which means you have access to its methods and properties:

```
customers.Where(c => c.City == "London");
```

The general rules for lambdas are as follows:

- The lambda must contain the same number of parameters as the delegate type.
- Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.
- The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

Note that lambda expressions in themselves do not have a type because the common type system has no intrinsic concept of "lambda expression." However, it is sometimes convenient to speak informally of the "type" of a lambda expression. In these cases the type refers to the delegate type or Expression type to which the lambda expression is converted.

Variable Scope in Lambda Expressions

Lambdas can refer to *outer variables* that are in scope in the enclosing method or type in which the lambda is defined. Variables that are captured in this manner are stored for use in the lambda expression even if variables would otherwise go out of scope and be garbage collected. An outer variable must be definitely assigned before it can be consumed in a lambda expression. The following example demonstrates these rules:

```
delegate bool D();
   delegate bool D2(int i);

   class Test
   {
      D del;
      D2 del2;
      public void TestMethod(int input)
      {
         int j = 0;
         // Initialize the delegates with lambda expressions.
         // Note access to 2 outer variables.
         // del will be invoked within this method.
```

```
        del = () => { j = 10;  return j > input; };

        // del2 will be invoked after TestMethod goes out of scope.
        del2 = (x) => {return x == j; };

        // Demonstrate value of j:
        // Output: j = 0
        // The delegate has not been invoked yet.
        Console.WriteLine("j = {0}", j);

        // Invoke the delegate.
        bool boolResult = del();

        // Output: j = 10 b = True
        Console.WriteLine("j = {0}. b = {1}", j, boolResult);
    }

    static void Main()
    {
        Test test = new Test();
        test.TestMethod(5);

        // Prove that del2 still has a copy of
        // local variable j from TestMethod.
        bool result = test.del2(10);

        // Output: True
        Console.WriteLine(result);

        Console.ReadKey();
    }
}
```

The following rules apply to variable scope in lambda expressions:

- A variable that is captured will not be garbage-collected until the delegate that references it goes out of scope.
- Variables introduced within a lambda expression are not visible in the outer method.
- A lambda expression cannot directly capture a **ref** or **out** parameter from an enclosing method.
- A return statement in a lambda expression does not cause the enclosing method to return.
- A lambda expression cannot contain a **goto** statement, **break** statement, or **continue** statement whose target is outside the body or in the body of a contained anonymous function.