Programming in .NET

Microsoft Development Center Serbia programming course

Lesson 5 – Delegates and lambda expressions

A delegate is an object that knows how to call a method. A *delegate type* defines the kind of method that *delegate instances* can call. Specifically, it defines the method's *return type* and its *parameter types*. The following code demonstrate how delegates are used:

```
        static double Max(double x, double y)
        {
            return x > y ? x : y;
        }

        static double Min(double x, double y)
        {
            return x > y ? y : x;
        }


        // declaration for a method named Function
        // with 2 double input values and double output value

        delegate double Function(double x, double y);

        public static void RunExample()
        {
            Function fun = Min;
            double x = fun(3, 4); // x = Min(3, 4);
            double y = fun(5, x); // y = Min(5, x);

            fun = Max;
            x = fun(4, 7); // x = Max(4, 7);


            fun = Math.Pow;
            x = fun(2, 7); // x = Math.Pow(2,7);

        }
```

In this example both `Min` and `Max` methods have the same signature and therefore can be used with a `Function` delegate type. In `RunExample` method we can see how the same delegate can be assigned to different methods and later executed. `Math.Pow` is a static method of `Math` class that have the same signature and therefore can be used as well. Statement

```
Function fun = Min;
```

Is shorthand for:

```
Function fun = new Function(Min);
```

Technically Min represents a method group, not a specific method, but C# compiler will pick an overload that matches the delegate signature. Executing a delegate:

```
x = fun(4, 7);
```

Is shorthand for:

```
x = fun.Invoke(4, 7);
```

Delegates can be passed as an argument as used as a callback. In the following example this trick is used to create a generic `Count` method that will count the number of instances that satisfy the specified criteria via function delegate called `Condition`.

```
        // "Conditional method" that should return true if condition is met
        delegate bool Condition(string x);

        public static void ExampleForCondition()
        {
            string[] arr = new string[] {
                           "Test",
                           "",
                           "Another test",
                           "   ",
                           "Another word",
                           ""
                       };

            int count;

            count = Count(arr, TooLong);
             // static string method provided as condition for counting
            count = Count(arr, string.IsNullOrEmpty);
            count = Count(arr, string.IsNullOrWhiteSpace);

        }

        static int Count(string[] array, Condition condition)
        {
            int count = 0;

            foreach (string d in array)
                if (condition(d))
                    count++;

            return count;
        }

        static bool TooLong(string word)
        {
            return word.Length > 5;
        }
```

In this example delegate is used as a predicate function. This means that delegate is typically taking a set of arguments and return bool which is then used as a condition for a certain action. Another example of using delegates is for action itself. This delegate type is typically without a return type. The following example shows the usage pattern for a callback delegate.

```
        // a call back type of a delegate
        delegate void Callback(double result);

        static void Process(double x, double y, Callback f)
        {
            double result = x * y;

            f(result);
        }

        public static void CallbackExample()
        {
            Callback fun = Console.WriteLine;

            Process(3.14, 4, fun);
            Process(1.5, 3.1, Console.WriteLine);

            // inline definition of a delegate
            Callback fun2 = delegate(double result)
                            {
                                Console.WriteLine("Result is {0}", result);
                            };

            Process(1.5, 3.1, fun2);
        }
```

Process method does some processing and calls a callback delegate which is supposed to do something with a result of an operation. A caller to this method specifies with delegate what should be done with result. In this example we can also see that delegates can be specified as inline methods without names, often called *anonymous delegates*. `Callback fun2` is assigned to an anonymous delegate that takes one double result and doesn't have any return value.

For an example from the beginning of this topic we can create anonymous delegates for Min function like this:

```
            Function fun;

            fun = delegate(double x, double y)
            {
                return x > y ? x : y;
            };
            double min = fun(3.14, 5);
```

So far we saw two patterns of usage for delegates, a callback and a condition delegate. For these typical usages, .NET defines a generic delegate types that can be used. This way a new delegate type doesn't have to be defined for each specific signature.

Action delegates are defined as:

```
   public delegate void Action<in T>(T obj);
   public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
   public delegate void Action<in T1, in T2, in T3>(T1 arg1, T2 arg2, T3 arg3);
   ...
```

`Action` delegates have all types as input values and void as returned value. Therefore `Callback` delegate type from our previous example is equivalent to `Action<double>` delegate type.

`Predicate` delegate is defined as:

```
public delegate bool Predicate<in T>(T obj);
```

Predicate delegate is used as a condition for a specific action. Single provided type is used as a type for input argument and return value is always `bool`.

More general than previous two, `Func` delegates are defined as:

```
    public delegate TResult Func<out TResult>();
    public delegate TResult Func<in T, out TResult>(T arg);
    public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
    ...
```

`Func` delegates define last type `T1...Tn` as input types for arguments and last types `TResult` as a result type. The following example shows how generic delegates can be used.

```
        public static void Example10()
        {
            Func<double, double> f = Math.Sin;
            double y = f(4); //y=sin(4)

            f = Math.Exp;
            y = f(4); //y=exp(4)

            f = delegate(double x) { return 3 * x + 1; };
            y = f(4); //y=13

            f = x => 3 * x + 1;
            y = f(5); //y=16

            Action<string> action = Console.WriteLine;
            action("Hello");

            Action<DateTime, string> printDate = delegate(DateTime date, string format){
                string result = date.ToString(format);
                Console.WriteLine(result);
            };

            printDate( DateTime.Now, "dd/MM/yyyy hh:mm");
            printDate( DateTime.Now, "MMM dd yyyy hh:mm:ss");


            Predicate<string> isEmpty = string.IsNullOrEmpty;
            Predicate<int> isPrime = delegate(int number)
            {

                if (number == 1) return false;
                if (number == 2) return true;

                for (int i = 2; i < number; ++i)
                {
                    if (number % i == 0) return false;
                }

                return true;

            };
```

```
            if (isEmpty("Word") && isPrime(7))
            {
            }

        }
```

These generic delegates are extremely general. The only practical scenarios not covered by these delegates are `ref/out` and pointer parameters.

All delegate instances have *multicast* capability. This means that a delegate instance can reference not just a single target method, but also a list of target methods. The + and += operators combine delegate instances. For example:

```
        delegate void Del(string s);

        static void Hello(string s)
        {
            Console.WriteLine("  Hello, {0}!", s);
        }

        static void Goodbye(string s)
        {
            Console.WriteLine("  Goodbye, {0}!", s);
        }

        public static void Example5()
        {
            Del fun = Console.WriteLine;

            fun += Hello;
            fun += Goodbye;

            fun("First"); // Console.WriteLine("First"), Hello("First"), Goodbye("First")

            fun -= Hello;

            fun("Two"); // Console.WriteLine("Two"), Goodbye("Two")

            Del f = fun + Console.Write;

            f("Three"); // Console.WriteLine("Three"), Goodbye("Three"), Console.Write("Three"),

            fun = f - Console.WriteLine;

            fun("Four"); // Goodbye("Four"), Console.Write("Four"),

            fun = delegate { Console.WriteLine("I don't use arguments!"); };

            fun("Five"); // Goodbye("Five"), Console.Write("Five"), Console.WriteLine("I don't us
e arguments!")
        }
```

If a multicast delegate has a non-void return type, the caller receives the return value from the last method to be invoked. The preceding methods are still called, but their return values are discarded. In most scenarios in which multicast delegates are used, they have void return types, so this subtlety does not arise.

A problem that can be solved with a delegate can also be solved with an interface. For example our example with Callback delegate can also be solved with ICallback interface that defines single method:

```
public interface ICallback
{
      delegate void Callback(double result);
}
```

A delegate design may be a better choice than an interface design if one or more of these conditions are true:

- The interface defines only a single method.
- Multicast capability is needed.
- The subscriber needs to implement the interface multiple times.

When you call a method, you can supply arguments that have more specific types than the parameters of that method. This is ordinary polymorphic behavior. For exactly the same reason, a delegate can have more specific parameter types than its method target. This is called *contravariance*.

```
delegate void HandleException(SystemException ex);

static void HandleIndexOutOfRangeException(object ex)
{
}

static void HandleNullReferenceException(Exception ex)
{
}

public static void ContravarianceExample()
{
    HandleException f;

    f = HandleIndexOutOfRangeException;
    f(new IndexOutOfRangeException("Index is out of range"));

    f = HandleIndexOutOfRangeException;
    f(new NullReferenceException("Object reference is not defined"));

}
```

Lambda expressions

A lambda expression is an unnamed method written in place of a delegate instance. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator `=>`, and you put the expression or statement block on the other side. For example, the lambda expression `x => x * x` specifies a parameter that's named x and returns the value of x squared.

The compiler immediately converts the lambda expression to either:

- A delegate instance.

- An expression tree, of type `Expression<TDelegate>`, representing the code inside the lambda expression in a traversable object model. This allows the lambda expression to be interpreted later at runtime.

The following example uses lambda expressions as delegates. It is in a way shorter syntax than anonymous delegates, and much easier to read, however lambda expressions are still strongly typed.

```
        delegate double Function(double x, double y);

        delegate bool Condition(string x);

        static int Count(string[] array, Condition condition)
        {
            int count = 0;

            foreach (string d in array)
                if (condition(d))
                    count++;

            return count;
        }


        public static void LambdaExample()
        {

            Function fun;

            // Lambda expression with two double input parameters that returns double
            // Types for x and y are determined based on a casting rules (they must be double)
            fun = (x, y) => x > y ? x : y;

            double min = fun(3.14, 5);

            string[] arr = new string[]
                  { "Test", "", "Another test", "   ", "Another word", "" };

            // Another lambda used to specify a string predicate
            int count = Count(arr, word => word.Length > 10);

        }
```

As already mentioned lambdas can appear in a form of expression and statement block.

Expression lambdas are defined with an expression on the right side. 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()
```

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);

public static void LambdaStatementExample()
{
      TestDelegate myDelegate = n => { string s = n + " " + "World"; Console.WriteLine(s); };
      myDelegate("Hello");
}
```

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

Recap

Lambda expressions and delegates (especially anonymous delegate) are a powerful tool for making a code more flexible and reusable. All of that C# makes really easy with clean and lightweight syntax.