Multicasting

This lesson spans over multicast delegates, which is an integral part of delegates in C#.

WE'LL COVER THE FOLLOWING

Example

Fun part in using delegates is that you can point to multiple functions using them, at the same time. This means that by calling a single delegate, you can actually invoke as many functions as you want.

Let's look at the example:

```
using System;
delegate void Procedure();

class DelegateDemo
{
    public static void Method1()
    {
        Console.WriteLine("Method 1");
    }

    public static void Method2()
    {
        Console.WriteLine("Method 2");
    }

    public void Method3()
    {
        Console.WriteLine("Method 3");
    }

    static void Main()
    {
        Procedure someProcs = null;
```

```
        Procedure someProcs = null;

        someProcs += new Procedure(DelegateDemo.Method1);
        someProcs += new Procedure(Method2);  // Example with omitted class name

        DelegateDemo demo = new DelegateDemo();

        someProcs += new Procedure(demo.Method3);
        someProcs();
    }
}
```

Explanation

The statement `someProcs += newProcedure(demo.Method3)` adds a *non-static method* to the delegate instance. For a *non-static method*, the method name is preceded by an **object reference**.

When the delegate instance is called, `Method3()` is called on the object that was supplied when the method was added to the delegate instance.

Finally, the statement `someProcs()` (line **31**) calls the delegate instance. All the methods that were added to the delegate instance are now called in the order that they were added.

Methods that have been added to a delegate instance can be removed with the `-=` **operator**:

```
someProcs -= new Procedure(DelegateDemo.Method1);
```

In C# 2.0, adding or removing a method reference to a delegate instance can be shortened as follows:

```
someProcs += DelegateDemo.Method1;
someProcs -= DelegateDemo.Method1;
```

Another Example

Let us look at another interesting example:

```
using System;
using System.Reflection;
```

```
using System.Reflection.Emit;

namespace DelegatesExample {

    class MainClass {
       private delegate void MyDelegate(int a);

       private static void PrintInt(int a)
       {
         Console.WriteLine(a);
       }

       private static void PrintType<T>(T a)
       {
         Console.WriteLine(a.GetType());
       }

       public static void Main (string[] args)
       {
         MyDelegate d1 = PrintInt;
         MyDelegate d2 = PrintType;
         // Output:
         // 1
         d1(1);
         // Output:
         // System.Int32
         d2(1);
         MyDelegate d3 = d1 + d2;
         // Output:
         // 1
         // System.Int32
         d3(1);
         MyDelegate d4 = d3 - d2;
         // Output:
         // 1
         d4(1);
         // Output:
         // True
         Console.WriteLine(d1 == d4);
       }
   }
}
```

Explanation

In this example, `d3` is a combination of `d1` and `d2` delegates, so when called the program outputs both `1` and `System.Int32` strings.

> Invoking a delegate instance that presently contains no method references results in a `NullReferenceException`.

Note that, if a delegate declaration specifies a return type and multiple

Note that, if a delegate declaration specifies a return type and multiple methods are added to a delegate instance, an invocation of the delegate instance returns the return value of the last method referenced. The return values of the other methods cannot be retrieved (unless explicitly stored somewhere in addition to being returned).

Interesting? Let us study how events work in C# next!