Delegates, Lamdba Expressions and Anonymous Methods

Declare, Instantiate, and Use a Delegate

In C# 1.0 and later, delegates can be declared as shown in the following example.

```
// Declare a delegate.

delegate void Del(string str);

// Declare a method with the same signature as the delegate.

static void Notify(string name)

{    Console.WriteLine("Notification received for: {0}", name);

}

// Create an instance of the delegate.

Del del1 = new Del(Notify);
```

C# 2.0 provides a simpler way to write the previous declaration, as shown in the following example.

```
// C# 2.0 provides a simpler way to declare an instance of Del.

Del del2 = Notify;
```

In C# 2.0 and later, it is also possible to use an anonymous method to declare and initialize a delegate, as shown in the following example.

```
// Instantiate Del by using an anonymous method.

Del del3 = delegate(string name)

   { Console.WriteLine("Notification received for: {0}", name); };
```

In C# 3.0 and later, delegates can also be declared and instantiated by using a lambda expression, as shown in the following example.

```
// Instantiate Del by using a lambda expression.

Del del4 = name =>  { Console.WriteLine("Notification received for: {0}", name); };
```

How to: Combine Delegates (Multicast Delegates)

This example demonstrates how to create multicast delegates. A useful property of delegate objects is that multiple objects can be assigned to one delegate instance by using the + operator. The multicast delegate contains a list of the assigned delegates. When the multicast delegate is called, it invokes the delegates in the list, in order. Only delegates of the same type can be combined.

The - operator can be used to remove a component delegate from a multicast delegate.

```
using System;

// Define a custom delegate that has a string parameter and returns void.

delegate void CustomDel(string s);

class TestClass

{   // Define two methods that have the same signature as CustomDel.

   static void Hello(string s)

   {       System.Console.WriteLine("  Hello, {0}!", s);

   }

   static void Goodbye(string s)

   {       System.Console.WriteLine("  Goodbye, {0}!", s);

   }

   static void Main()
   {       // Declare instances of the custom delegate.

       CustomDel  hiDel, byeDel, multiDel, multiMinusHiDel;

       // In this example, you can omit the custom delegate if you

       // want to and use Action<string> instead.

       //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

       // Create the delegate object hiDel that references the

       // method Hello.

       hiDel = Hello;

       // Create the delegate object byeDel that references the

       // method Goodbye.

       byeDel = Goodbye;

       // The two delegates, hiDel and byeDel, are combined to

       // form multiDel.

       multiDel = hiDel + byeDel;

       // Remove hiDel from the multicast delegate, leaving byeDel,

       // which calls only the method Goodbye.

       multiMinusHiDel = multiDel - hiDel;
```

```
        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}

/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

Action<T1, T2> Delegate

```
public delegate void Action<in T1, in T2>(
        T1 arg1,
        T2 arg2
)
```

Parameters

arg1

Type: T1

The first parameter of the method that this delegate encapsulates.

arg2

Type: T2

The second parameter of the method that this delegate encapsulates.

Type Parameters

in T1

The type of the first parameter of the method that this delegate encapsulates.

in T2

The type of the second parameter of the method that this delegate encapsulates.

You can use the Action<T1, T2> delegate to pass a method as a parameter without explicitly declaring a custom delegate. The encapsulated method must correspond to the method signature that is defined by this delegate. This means that the encapsulated method must have two parameters that are both passed to it by value, and it must not return a value

```
using System;

using System.IO;

delegate void ConcatStrings(string string1, string string2);

public class TestDelegate

{   public static void Main()

  {   string message1 = "The first line of a message.";

    string message2 = "The second line of a message.";

    ConcatStrings concat;

    if (Environment.GetCommandLineArgs().Length > 1)

      concat = WriteToFile;

    else

      concat = WriteToConsole;

    concat(message1, message2);

  }

  private static void WriteToConsole(string string1, string string2)

  {     Console.WriteLine("{0}\n{1}", string1, string2);

  }
  private static void WriteToFile(string string1, string string2)

  {

    StreamWriter writer = null;

    try

    {

      writer = new StreamWriter(Environment.GetCommandLineArgs()[1], false);

      writer.WriteLine("{0}\n{1}", string1, string2);

    }

    catch

    {

      Console.WriteLine("File write operation failed...");

    }

    finally

    {

      if (writer != null) writer.Close();

    }

  }

}
```

The following example simplifies this code by instantiating the Action<T1, T2> delegate instead of explicitly defining a new delegate and assigning a named method to it.

```
using System;
using System.IO;

public class TestAction2
{
   public static void Main()
   {
      string message1 = "The first line of a message.";
      string message2 = "The second line of a message.";
      Action<string, string> concat;

      if (Environment.GetCommandLineArgs().Length > 1)
         concat = WriteToFile;
      else
         concat = WriteToConsole;

      concat(message1, message2);
   }
   private static void WriteToConsole(string string1, string string2)
   {
      Console.WriteLine("{0}\n{1}", string1, string2);
   }

   private static void WriteToFile(string string1, string string2)
   {
      StreamWriter writer = null;
      try
      {
         writer = new StreamWriter(Environment.GetCommandLineArgs()[1],
false);
         writer.WriteLine("{0}\n{1}", string1, string2);
      }
      catch
      {
         Console.WriteLine("File write operation failed...");
      }
      finally
      {
         if (writer != null) writer.Close();
      }
   }
}
```

Func<T1, T2, TResult> Delegate

Encapsulates a method that has two parameters and returns a value of the type specified by the *TResult* parameter.

You can use this delegate to represent a method that can be passed as a parameter without explicitly declaring a custom delegate. The encapsulated method must correspond to the method signature that is defined by this delegate. This means that the encapsulated method must have two parameters, each of which is passed to it by value, and that it must return a value.

For example, the following code explicitly declares a delegate named ExtractMethod and assigns a reference to the ExtractWords method to its delegate instance.

```
using System;

delegate string[] ExtractMethod(string stringToManipulate, int maximum);

public class DelegateExample{

  public static void Main()

  {     // Instantiate delegate to reference ExtractWords method

    ExtractMethod extractMeth = ExtractWords;

    string title = "The Scarlet Letter";

    // Use delegate instance to call ExtractWords method and display result
    foreach (string word in extractMeth(title, 5))
       Console.WriteLine(word);
  }
private static string[] ExtractWords(string phrase, int limit)
  {     char[] delimiters = new char[] {' '};
    if (limit > 0)
       return phrase.Split(delimiters, limit);

  else
       return phrase.Split(delimiters);
  }
}
```

The following example simplifies this code by instantiating a Func<T1, T2, TResult> delegate instead of explicitly defining a new delegate and assigning a named method to it.

```
using System;

public class GenericFunc
{
   public static void Main()
   {
      // Instantiate delegate to reference ExtractWords method
      Func<string, int, string[]> extractMethod = ExtractWords;
      string title = "The Scarlet Letter";
      // Use delegate instance to call ExtractWords method and display result
      foreach (string word in extractMethod(title, 5))
         Console.WriteLine(word);
   }

   private static string[] ExtractWords(string phrase, int limit)
   {
      char[] delimiters = new char[] {' '};
      if (limit > 0)
         return phrase.Split(delimiters, limit);
      else
         return phrase.Split(delimiters);
   }
}
```

Anonymous Methods

In versions of C# before 2.0, the only way to declare a delegate was to use named methods. C# 2.0 introduced anonymous methods and in C# 3.0 and later, lambda expressions supersede anonymous methods as the preferred way to write inline code. However, the information about anonymous methods in this topic also applies to lambda expressions.

```
// Create a delegate.

delegate void Del(int x);

// Instantiate the delegate using an anonymous method.

Del d = delegate(int k) { /* ... */ };
```

The following example demonstrates two ways of instantiating a delegate:

- Associating the delegate with an anonymous method.
- Associating the delegate with a named method (DoWork).

In each case, a message is displayed when the delegate is invoked.

```
// Declare a delegate.

delegate void Printer(string s);

class TestClass

{   static void Main()

  {     // Instantiate the delegate type using an anonymous method.

      Printer p = delegate(string j)

      {         System.Console.WriteLine(j);

      };

      // Results from the anonymous delegate call.

      p("The delegate using the anonymous method is called.");

      // The delegate instantiation using a named method "DoWork".

      p = new Printer(TestClass.DoWork);

      // Results from the old style delegate call.

      p("The delegate using the named method is called.");

  }

  // The method associated with the named delegate.
```

```
    static void DoWork(string k)

    {

        System.Console.WriteLine(k);

    }

}

/* Output:

    The delegate using the anonymous method is called.

    The delegate using the named method is called.

*/
```

Lambda Expressions

A lambda expression is an anonymous function that you can use to create delegates or expression tree types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. Lambda expressions are particularly helpful for writing LINQ query expressions

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. You can assign this expression to a delegate type, as the following example shows:

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

Expression Lambdas

A lambda expression with an expression on the right side of the => operator is called an *expression lambda*. Expression lambdas are used extensively in the construction of Expression Trees (C# and Visual Basic). 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 are evaluated outside of the .NET Framework, such as in 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.