Agenda

- What are Delegates?
- Delegates Implementation
- Multicast Delegate
- Delegates Vs Interfaces
- What are Events?
- Events Implementation
- Covariance and Contravariance

What is a Delegate?

- 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.
- A delegate can reference a method only if the signature of the method exactly matches the signature specified by the delegate type.
- Delegates are similar to function pointers in other languages like C++, however, delegates are type-safe.

Delegates Implementation Defining and Using Delegates

- There are three steps in defining and using delegates:
  - Declaration - A delegate declaration defines a class that is derived from the class System.Delegate.
  - Instantiation - A delegate instance encapsulates an invocation list, which is a list of one or more methods, each of which is referred to as a callable entity.
  - Invocation - Invoking a delegate instance with an appropriate set of arguments causes each of the delegate instance's callable entities to be invoked with the given set of arguments.
    - Note: Delegates run under the caller's security permissions, not the declarer's permissions.

Delegates Implementation – Example 1

```
using System;

namespace TestConsoleApps
{
    /// <summary>
    /// Summary description for SimpleDelegate.
    /// </summary>

    public class SimpleDelegate
    {
        public delegate int AddMulDelegate(int a, int b);
        public int AddNumber(int a, int b)
        {
                return (a + b);
        }

        public int MulNumber(int a, int b)
        {
                return (a * b);
        }

        static void Main()
        {
                SimpleDelegate simpDel = new SimpleDelegate();
                AddMulDelegate addDelegate = new AddMulDelegate(simpDel.AddNumber);
                AddMulDelegate mulDelegate = new AddMulDelegate(simpDel.MulNumber);
                int addAns = addDelegate(10, 12);
                int mulAns = mulDelegate(10, 10);
                Console.WriteLine("Result by calling the AddNumber method using a delegate: {0}", addAns);
                Console.WriteLine("Result by calling the MulNumber method using a delegate: {0}", mulAns);
                Console.Read();
            }
        }
}
```

Delegate Declaration

The call to the delegate is translated into a call to the associated method. The respective method is invoked with the parameters passed to the delegate.

Delegate is instantiated and a method is associated with it. Call to the delegate would now be translated into a call to that method.

Delegates Implementation – Example 2

```
using System;
namespace TestCSharpApps
{
        public class DelegateClass
        {
                // 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 TestSimpleDelegate
{
    // 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)
    {
            DelegateClass myDelegateClass = new DelegateClass();

            // Crate an instance of the delegate, pointing to the logging
function.
            // This delegate will then be passed to the Process() function.
            DelegateClass.LogHandler myLogger = new
DelegateClass.LogHandler(Logger);
            myDelegateClass.Process(myLogger);
            Console.Read();
        }
  }
}
```

Delegate Declaration **Delegate Invocation** **Delegate Instantiation**

Delegates Implementation – Example 2 (cont.)

- Output

```
Process() begin
Process() end
```

Multicast Delegate

- Delegate object that we have created holds a reference to one method. However, it is possible for a delegate object to hold references to invoke multiple methods. Such delegate objects are called multicast delegates or combinable delegates.
- Can encapsulate more than one method into a single delegate by using += operator.
- The following two conditions need to be met for multicast delegates:
  - None of the parameters of the delegate is an output parameter.
  - The return type of the delegate is void.

Multicast Delegate

- Example:

```
class MulticastExample
{
    public delegate int AddMul(int I,int j);
    public int Add(int i,int j)
    {
            return i+j;
    }
    public int Mul(int i,int j)
    {
            return i*j;
    }
}
```

```
MulticastExample me=new MulticastExample();
MulticastExample.AddMul Arith=null;
Arith+=new MulticastExample.AddMul(me.Add);
Arith+=new MulticastExample.AddMul(me.Mul);
Int x=Arith(10,10);
Console.WriteLine("Result for Add={0}",x);
Int y=Arith(2,10);
Console.WriteLine("Result for Mul={0}",y);
```

Delegates vs. Interfaces

- Delegates are useful when:
  - A single method is being called.
  - A class may want to have multiple implementations of the method specification.
  - It is desirable to allow using a static method to implement the specification.
  - An event-like design pattern is desired.
  - The caller has no need to know or obtain the object in which the method is defined.
  - The provider of the implementation wants to "hand out" the implementation of the specification to only a few select components.
  - Callback functions are required.

Delegates vs. Interfaces (cont.)

- Interfaces are useful when:
  - The specification defines a set of related methods that will be called.
  - A class typically implements the specification only once.
  - The caller of the interface wants to cast to or from the interface type to obtain other interfaces or classes.

Events

- An event is a way for a class to provide notifications to clients of that class when some interesting thing happens to an object.
- Events provide a generally useful way for objects to signal state changes that may be useful to clients of that object. Events are an important building block for creating classes that can be reused in a large number of different programs.
- Events are declared using delegates. An event is a way for a class to allow clients to give delegates to methods that should be called when the event occurs. When the event occurs, the delegate(s) given to it by its clients are invoked.

Event Implementation (1 of 4)

- Steps to handle an event:
  - Step 1: Declare a delegate.
  - Step 2: Create a class derived from System.EventArgs to encapsulate the event information.
  - Step 3: Create the class that will raise the event. This class will have:
    a) An event which will have the registered clients.
    b) A method to notify registered clients.
  - Step 4: Create a client class (This class will have a method that matches the signature of the delegate and will receive the event).
  - Step 5: Register this method with the event as being the authorized listener for the event.

Event Implementation (2 of 4)

```
using System;
using System.ComponentModel;

namespace TestEventApps
{
  // First step -- declare the delegate
  public delegate void EvenEventHandler(object sender, EvenEventArgs e);

  //Second step -- create a class derived from EventArgs
  public class EvenEventArgs : EventArgs
  {
    // Declare private variables to reflect the information about the event
    private readonly bool evensteven;
    private readonly int evenOne, evenTwo;

    //Constructor
    public EvenEventArgs(bool evensteven, int firstEvent, int secondEvent)
    {
      this.evensteven = evensteven;
      this.evenOne = firstEvent;
      this.evenTwo = secondEvent;
    }

    //The properties to enable the client to access the event info
    public bool Evensteven
    {
                    get
                    {
                      return evensteven;
                    }
    }
```

```
    public int EvenOne
    {
      get
      {
        return evenOne;
      }
    }

    public int EvenTwo
    {
      get
      {
        return evenTwo;
      }
    }
  }
```

Step 1: Declare a delegate.

Step 2: Create class to encapsulate event information.

Event Implementation (3 of 4)

```
//Third step -- create the class that will raise the event
  public class EvenDetector
  {
    //Declare some variables to make life simpler
    private bool gotEven, done;

    //and our own random number generator
    private Random r1;

    //as well as the number we shall check for 'evenness'
    private int randomNum;

    //Also the two even numbers
    private int evenOne, evenTwo;

    //Constructor
    public EvenDetector()
    {
      r1 = new Random();
    }

   public void NumberCruncher()
    {
        //Loop until two successive even numbers have been generated
       while (!done)
       {
          randomNum = (int)(100 * r1.NextDouble());
```

Step 3a: Declare the event.

Step 3b: Method to notify the clients.

```
if (randomNum % 2 == 0)
{
     if (gotEven)
     {
       done = true;
       evenTwo = randomNum;
     }
     else
     {
       gotEven = true;
       evenOne = randomNum;
     }
}
else
{
     gotEven = false;
}
}
     //Success -- create an object for the client(s)
     EvenEventArgs e = new EvenEventArgs(done, evenOne, evenTwo);

     //and call the method to send it to the client(s)
     OnEven(e);
  }
  //Step 3a -- the event is declared
  public event EvenEventHandler Even;

  //Step 3b -- the method that notifies client(s)
  protected virtual void OnEven(EvenEventArgs e)
  {
    // If the list of clients is NOT empty
    if (Even != null)
    {
          //despatch the event to each client
          Even(this, e);
    }
  }
}
```

Invoke the event.

Event Implementation (4 of 4)

```
//Fourth step -- and now the client class
  public class EvenListener
  {
    //This is the method with the same signature as the delegate.
    //It will receive the event
    public void EvenAnnouncer(object sender, EvenEventArgs e)
    {
      //This will always be true and hence is redundant.
 //Just illustrates the use of EventArgs
 if (e.Evensteven)
 {
   Console.WriteLine("THE EVEN TWINS ARE HERE! -- {0} and {1}", e.EvenOne, e.EvenTwo);
 }
    }
  }

//Fifth step -- hook them up
  public class EvenTester
  {
    public static void Main()
    {
      //Instantiate EvenDetector
     EvenDetector ed = new EvenDetector();

    //Instantiate EvenListener
    EvenListener el = new EvenListener();

      //Register the listener method
    ed.Even += new EvenEventHandler(el.EvenAnnouncer);

      ed.NumberCruncher();
    }
  }
}
```

Step 4: Create the client class.

Step 5: Register the method with the event.

Asynchronous Delegate Calls (1 of 4)

- Generally delegate operations are executed in the order in which they are called i.e., they are synchronous. But **BeginInvoke()** and **EndInvoke()** methods of a delegate help to process delegates asynchronously.
- **BeginInvoke()** returns **IAsyncResult** which is used to monitor asynchronous calls. IAsyncResult contains the following members which help in monitoring the asynchronous calls:
  - **AsyncState** – returns an object containing information of asynchronous operations.
  - **AsyncWaitHandle** – waits for an asynchronous operation to complete.
  - **IsCompleted** – checks whether the asynchronous call has completed or not.

Asynchronous Delegate Calls (2 of 4)

- Methods to determine whether an asynchronous operation is completed or not:
  - Polling using IsCompleted.
  - Waiting for the operation to complete using AsyncWaitHandle.
  - Handling callbacks using AsyncState.
- When EndInvoke() method is called, it blocks the main thread of execution and waits for the specific asynchronous operation to complete. It is also used to get the values returned by the asynchronous operations.

Asynchronous Delegate Calls (3 of 4)

```
using System;
namespace TestAsyncDelegateCalls
{
   class TeamManager
   {
      public delegate void DevelopmentHandler();
      public void DeliverChangeRequest()
      {
         Console.WriteLine("Estimation delivered - by manager");

         DevelopmentHandler developer = new DevelopmentHandler(ImplementChanges);
         IAsyncResult result = developer.BeginInvoke(null, null);

         Console.WriteLine("Updated task in MS Project server - by manager");
         Console.WriteLine("Updated requirements document - by manager");

         developer.EndInvoke(result);

         Console.WriteLine("Change Request delivered - by manager");

      }

      private void ImplementChanges()
      {
         Console.WriteLine("Impact analysis done - by developer");
         Console.WriteLine("Updated design documents - by developer");
         Console.WriteLine("Implemented code changes - by developer");
      }
   }
}
```

Declare delegate **DevelopmentHandler** which will be called asynchronously.

```
using System;
namespace TestAsyncDelegateCalls
{
   class MainClass
   {
      static void Main()
      {
         TeamManager manager = new TeamManager();
         manager.DeliverChangeRequest();

         Console.ReadLine();
      }
   }
}
```

Create **developer** of type **DevelopmentHandler** and associate **ImplementChanges()** method to it.

Call delegate **developer** asynchronously by using its **BeginInvoke()** method which has a return type of **IAsyncResult.**

Wait for the asynchronous operation to get over by calling **EndInvoke()** method of the delegate **developer**.

ImplementChanges() method processes the delegate **developer** and is executed asynchronously.

Asynchronous Delegate Calls (4 of 4)

- Output 1:

```
Estimation delivered - by manager
Updated task in MS Project server - by manager
Updated requirements document - by manager
Impact analysis done - by developer
Updated design documents - by developer
Implemented code changes - by developer
Change Request delivered - by manager
```

- Output 2:

```
Estimation delivered - by manager
Impact analysis done - by developer
Updated task in MS Project server - by manager
Updated requirements document - by manager
Updated design documents - by developer
Implemented code changes - by developer
Change Request delivered - by manager
```

- Two different outputs show the asynchronous processing of delegates.

Covariance and Contravariance

- To have more flexibility while matching delegate methods with delegate signatures.
- Flexibility can be the ability of a delegate to cope up with different classes or events of the program.
- **Covariance:**
  - The method passed to a delegate must have the same return type and signature as the delegate. However Covariance allows you to use a method that has a derived return type as delegate.
  - Consider the following example:
    - Delegate object GetTextData(TextBox control); //defines delegate
    - The above delegate can reference to the method below, which has a different return type:

```
String GetTextControl(TextBox control)
{
        return control.Text;
}
```

Covariance and Contravariance (cont.)

- **Contravariance:**
  - Allows you to use method with derived parameters as a delegate.
  - Enables a delegate to point to a method whose argument is the base type of argument appearing in the delegate signature.
  - Allows you to create more general delegate methods that can be used with larger number of classes:

Example:

```
Object GetControl(Control ctl)
{
 return ctl.Tag;
}
```