PROGRAMMING IN C#

Unit-3
Delegates and Events

Dr.M.Paul Arokiadass Jerald

SYLLABUS – UNIT 3

DELEGATES AND EVENTS

Delegates – Declaring a Delegate – Defining Delegate Methods – Creating and Invoking Delegate Objects – Multicasting with Delegates

Events – Event Sources – Event Handlers – Events and Delegates.

3.1 Delegates

- A delegate is an **object** that can **refer to a method**.
- When a delegate is created, an object that can hold a reference to a method is created.
- Their flexibility allows to define the exact signature of the callback, and that information becomes part of the delegate type itself.
- Delegates are type-safe, object-oriented and secure

Delegates

- Delegates implement the callback mechanism.
  - Provide a typesafe way to define a callback.
  - Implement the ability to call several methods in sequence.
  - Support calling of both static and instance methods.

- Delegates are declared and used in one class, created in another:
  - Publisher class, pubClass, declares a callback type:

    ```
    public delegate returnType CallbackName([callback arg],…);
    ```

  - Subscriber class, subClass, creates a delegate, handing it a pointer to the event handler function:

    ```
    pubClass.CallbackName cb = new  pubClass.CallbackName(subClass.fun1);
    cb += myClass.CallbackName(subClass.fun2);
    ```

  - Used in pubClass:

    ```
    if(cb != null)
        cb(callback args);  // calls subClass.fun1, fun2
    ```

3.2 CHARACTERISTICS OF DELEGATES

- Delegates are derived from the System.MulticastDelegate class.
- They have a signature and a return type. A function that is added to delegates must be compatible with this signature.
- Delegates can point to either static or instance methods.
- Once a delegate object has been created, it may dynamically invoke the methods it points to at runtime.
- Delegates can call methods synchronously and asynchronously.
- Fields in a delegate : a reference to an object, and the second holds a method pointer.
- When a delegate is invoked, the instance method is called on the contained reference.

Delegate Class

- When you declare a delegate:

```
    public delegate rtn MyEventHandler(arg1, arg2);
```

- The compiler generates a nested class:

```
public class MyEventHandler : System.MulticastDelegate {

  public MyEventHandler(object target, Int32 methodPtr);      // ctor
  public virtual rtn invoke(arg1, arg2);                      // what’s called

// these methods support asynchronous callbacks

  public virtual IAsyncResult BeginInvoke(
    arg1, arg2, AsyncCallback callback, object Obj
  );
  public virtual void EndInvoke(IAsyncResult result);

}
```

System.Delegate Class

```
public abstract class Delegate :
    object,
    ICloneable,
    System.Runtime.Serialization.ISerializable
{
    // Fields

    // Constructors

    // Properties
    public MethodInfo Method { get; }
    public object Target { get; }

    // Methods
    public virtual object Clone();
    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Combine(Delegate[] delegates);
    public static Delegate CreateDelegate(Type type, System.Reflection.MethodInfo method);
    public static Delegate CreateDelegate(Type type, object target, string method);
    public static Delegate CreateDelegate(Type type, Type target, string method);
    public static Delegate CreateDelegate(Type type, object target, string method, bool ignoreCase);
    public object DynamicInvoke(object[] args);
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public virtual Delegate[] GetInvocationList();
    public virtual void GetObjectData(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context
    );
    public Type GetType();
    public static Delegate Remove(Delegate source, Delegate value);
    public virtual string ToString();
}
```

Combine adds a new callback method

Remove deletes a callback method

3.3 Types of Delegates

- **Types of Delegates**
  1. Single Cast Delegates.
  2. Multi Cast Delegates.

3.3.1 SINGLE CAST DELEGATE

- Single cast delegate refer to single method at a time.
- In this the delegate is assigned to a single method at a time.
- They are derived from System.Delegate class.
- **Example :**
- https://www.dotnetheaven.com/article/singlecast-delegate-in-csharp#:~:text=Singlecast%20delegate%20refer%20to%20single,Delegate%20class

SINGLE CAST DELEGATE- Example

```
using System;
public delegate int Addsub(int a);
namespace DelegateAppl
{
   class DelegateExample
   {
      static int num = 5;
      public static int AddNum(int b)
      {
         num += b;
         return num;
      }

      public static int subNum(int c)
      {
         num -= c;
         return num;
      }
      public static int getNum()
      {
         return num;
      }
```

```
  static void Main(string[] args)
  {
     //create delegate instances
     Addsub D1 = new Addsub(AddNum);
     Addsub D2 = new Addsub(subNum);
     D1(25);
     Console.WriteLine("Value of Num: {0}",
getNum());
     D2(5);
     Console.WriteLine("Value of Num: {0}",
getNum());
     Console.ReadKey();
      }
   }
}
```

3.3.2 Multicast Delegates

- Multicast delegate can be used to invoke the multiple methods.
- The delegate instance can do multicasting (adding new method on existing delegate instance) using the + operator and – operator can be used to remove a method from a delegate instance.
- All methods will invoke in sequence as they are assigned.
- Multicast delegates, also known as combinable delegates.
- They must follow following conditions.
  - The return type must be void
  - None of the parameters of the delegate type can be declared as output parameter.

System.MulticastDelegate Class

-
```
public abstract class MulticastDelegate : Delegate, ICloneable,
System.Runtime.Serialization.ISerializable
{
    // Fields

    // Constructors

    // Properties
    public MethodInfo Method { get; }
    public object Target { get; }

    // Methods
    public virtual object Clone();
    public object DynamicInvoke(object[] args);
    public virtual bool Equals(object obj);
    public virtual int GetHashCode();
    public virtual Delegate[] GetInvocationList();
    public virtual void GetObjectData(
        System.Runtime.Serialization.SerializationInfo info,
        System.Runtime.Serialization.StreamingContext context
    );
    public Type GetType();
    public virtual string ToString();
}
```

DynamicInvoke is used by derived class’s invoke function

3.4 Creating a Delegate

- Creating and using delegates involves four steps. They include:
  1. Delegate Declaration
  2. Delegate method definition
  3. Delegate instantiation
  4. Delegate invocation.

Syntax for Delegates

- ***Declare Delegate (in publisher):***
  - Public delegate [return type] [delegate name] ([list of parameters]);
  - Example : **public delegate int** operation(**int** x, **int** y);
- ***Declare Event (in publisher):***
  - [modifier] event [delegate name] [event identifier];
- ***Register Callback Function (in subscriber):***
  - [delegate name] [delegate object] = new [delegate name]([callback1 name]);
  - [delegate object] += new [delegate name]([callback2 name]);
- ***Invoke Callback Function:***
  - If([delegate object != null) [delegate object]([list of arguments]);
  - in [list of arguments] types must match those in [list of parameters] in delegate declaration

SAMPLE DELEGATE PROGRAM

```
using System;
namespace Delegates
{   // Delegate Definition
    public delegate int operation(int x, int y);
    class Program
    {
        // Method that is passes as an Argument
        // It has same signature as Delegates
        static int Addition(int a, int b)
        {
            return a + b;
        }
        static void Main(string[] args)
        {   // Delegate instantiation
            operation obj = new operation(Program.Addition);

            // output
            Console.WriteLine("Addition is={0}",obj(23,27));
            Console.ReadLine();
        }
    }
```

Declaring a delegate

Defining a delegate

instantiating a delegate

invokingg a delegate

3.5 Delegate Invocation

- When a class method invokes a callback:

```
C.MyEventHandler myEventHandler = new C.MyEventHandler(func)
                :
if(myEventHandler != null)
    myEventHandler(arg1, arg2);
```

- The compiled code is doing this:

```
if(myEventHandler != null)
    myEventHandler.invoke(arg1, arg2);
```

3.6 EVENTS

- An event is a delegate type class member that is used by the object or class to provide a notification to other objects that an event has occurred.
- Events are declared using the simple event declaration format as follows:-

modifier event type event-name;

- The modifier may be a new , static , override , abstract and sealed.
- For eg:- public event EventHandler Click;
- EventHandler is a delegate and Click is an event.

Types of Events

- Types of Events :
    - **Physical event**
    - **logical event**

    - **A physical event** is the initiation or completion of some action that the program needs to know about, e.g., a mouse button down, keypress action, or some specific state in your program.
    - **A logical event** is a language construct, explained below.
- Events are members of CLR classes.

Functionalities of Events

- Events functionalities:
  - The ability for other objects to register interest in the event
  - The ability to unregister for the event
  - That the object defining the event will maintain a list of registered objects and notify them when the physical event occurs.

- Events are CLR constructs, that is, they are available to all languages that support the CLR:
  - C#, managed C++, Visual Basic, Jscript, …

Callbacks

- A callback is a pointer to a function that is called when some event occurs.
- Usually, a class defines methods for other objects to call.
- A callback is different. A callback is a request for another class to implement a function with a specific signature that this class will invoke.
  - A callback is a method pointer that the defining class declares, giving the method's signature
  - Some other class is responsible for implementing the function to be called and registering the name of that function by passing back a pointer to the defining class.
  - The defining class uses the function pointer to invoke the implementor's function when some event occurs.
- Callbacks are a general programming technique that have been used ever since event-based programming began.
- The CLR supports callbacks with an event keyword and a delegate type.

Publish and Subscribe

- A callback is declared and invoked by the publisher of an event.
  - Declaration sets the signature that must be used for the callback.
  - A callback is invoked by the publisher every time an event occurs.

- A callback is defined by a subscriber class.
  - The subscriber defines a method with the same signature declared by the publisher in its callback declaration.
    - This function handles the event
  - The subscriber then registers its event handler function by passing back a pointer to the function to the publisher.

Conventions

- Convention:
  - By convention the Delegate type accepts two parameters:
    - ***object sender***
      A reference to invoker of the callback
    - ***EventArgs e***
      An instance of a class derived from EventArgs that wraps data needed by the Application’s callback function.
  - And returns void
- The CLR predefines a “standard” delegate:
  - delegate void System.EventHandler(object sender, EventArgs e);

EventArgs Class

-
  ```
  public class EventArgs : object
  {

      // Fields
      public static readonly EventArgs Empty;

      // Constructors
      public EventArgs();

      // Methods
      public virtual bool Equals(object obj);
      public virtual int GetHashCode();
      public Type GetType();
      public virtual string ToString();
  }
  ```

Event Class

- When the compiler detects delegates:

  ```
  public delegate rtn MyEventHandler(arg1, arg2);
  public event MyEventHandler myEv;
  ```

- Compiler generates statements in the class for the events:
  - `Private MyEventHandler myEv = null;    // private delegate field`

  -
    ```
    [MethodImplAttribute(MethodImplOptions.Synchronized)]
    public void add_myEv(myEventHandler handler) {
        myEv = (myEventHander)Delegate.Combine(myEv, handler);
    }
    ```

  -
    ```
    [MethodImplAttribute(MethodImplOptions.Synchronized)]
    public void remove_myEv(myEventHandler handler) {
        myEv = (myEventHander)Delegate.Remove(myEv, handler);
    }
    ```

Default System.EventHandler Class

-
  ```
  public sealed class EventHandler :
      MulticastDelegate,
      ICloneable,
      System.Runtime.Serialization.ISerializable
  {
      // Constructors
      public EventHandler(object object, IntPtr method);

      // Properties
      public MethodInfo Method { get; }
      public object Target { get; }

      // Methods
      public virtual IAsyncResult BeginInvoke(
          object sender, EventArgs e, AsyncCallback callback, object object
      );
      public virtual object Clone();
      public object DynamicInvoke(object[] args);
      public virtual void EndInvoke(IAsyncResult result);
      public virtual bool Equals(object obj);
      public virtual int GetHashCode();
      public virtual Delegate[] GetInvocationList();
      public virtual void GetObjectData(
          System.Runtime.Serialization.SerializationInfo info,
          System.Runtime.Serialization.StreamingContext context
      );
      public Type GetType();
      public virtual void Invoke(object sender, EventArgs e);
      public virtual string ToString();
  }
  ```

Publisher’s Responsibilities

1. Define a nested type derived from System.EventArgs to package arguments needed by the event handler functions.
   - If you don’t need any, skip this step and just use an EventArgs object.
2. Define a delegate type specifying the prototype for the event handler.
3. Declare an event in the publisher class using the delegate you just defined.
4. Define a protected virtual method responsible for using the delegate to notify subscribers. The publisher calls this method when the event occurs, passing to it the EventArgs instance.
5. Define the processing that results in events. When an event occurs, call the notification function defined above.

Subscriber's Responsibilities

- Provide a constructor that accepts a reference to a Publisher instance, say pub.
  - In the constructor you construct a new instance of Publisher's delegate:

    ```
    pub.theEvent += new Publisher.theEventHandler(subHandler);
    ```

- Define a message handler that accepts the parameters specified by the delegate and returns the type specifed by the delegate.
  - Usually the arguments are object sender and the publisher's EventArgs object.

    ```
    Private void subHandler(object sender, Publisher.PubEventArgs e)
    {
        // handle message
    }
    ```

Application’s Responsibilities

- Construct a Publisher object:

  ```
  Publisher pub;
  ```

- Construct a Subscriber object:

  ```
  Subscriber(pub);
  ```

- Call pub’s method(s) to perform the application’s activities.

3.7 Difference between delegates and events

- **Delegate** is a function pointer. It holds the reference of one or more methods at runtime.
- **Delegate** is independent and not dependent on **events**.
- An **event** is dependent on a **delegate** and cannot be created without **delegates**.

DELEGATE

Another Example

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace DelegateApp
{   /// A class to define a person
   public class Person
    {  public string Name { get; set; }
        public int Age { get; set; }
    }

    class Program
    {  //delegate declaration
        public delegate bool FilterDelegate(Person p);
       static void Main(string[] args)
        {   //Create 4 Person objects
      Person p1 = new Person() { Name = "John", Age = 41 };
      Person p2 = new Person() { Name = “Gopi", Age = 69 };
      Person p3 = new Person() { Name = “Ram", Age = 12 };
      Person p4 = new Person() { Name = “Jenni", Age = 25 };
            //Create a list of Person objects and fill it
      List<Person> people = new List<Person>() { p1, p2, p3,
       p4 };
     //Invoke DisplayPeople using appropriate delegate
            DisplayPeople("Children:", people, IsChild);
            DisplayPeople("Adults:", people, IsAdult);
            DisplayPeople("Seniors:", people, IsSenior);
            Console.Read();
        }
```

```
/// A method to filter out the people you need
/// <param name="people">A list of people</param>
/// <param name="filter">A filter</param>
```

```
 static void DisplayPeople(string title,
List<Person> people, FilterDelegate filter)
 {
     Console.WriteLine(title);
     foreach (Person p in people)
     {
         if (filter(p))
         {
Console.WriteLine("{0}, {1} years old", p.Name, p.Age);
         }
     }
     Console.Write("\n\n");
 }
//==========FILTERS==================
 static bool IsChild(Person p)
 {           return p.Age < 18;
 }
 static bool IsAdult(Person p)
 {           return p.Age >= 18;
 }
 static bool IsSenior(Person p)
 {           return p.Age >= 65;
 }
}
}
```

C# EVENTS - Example

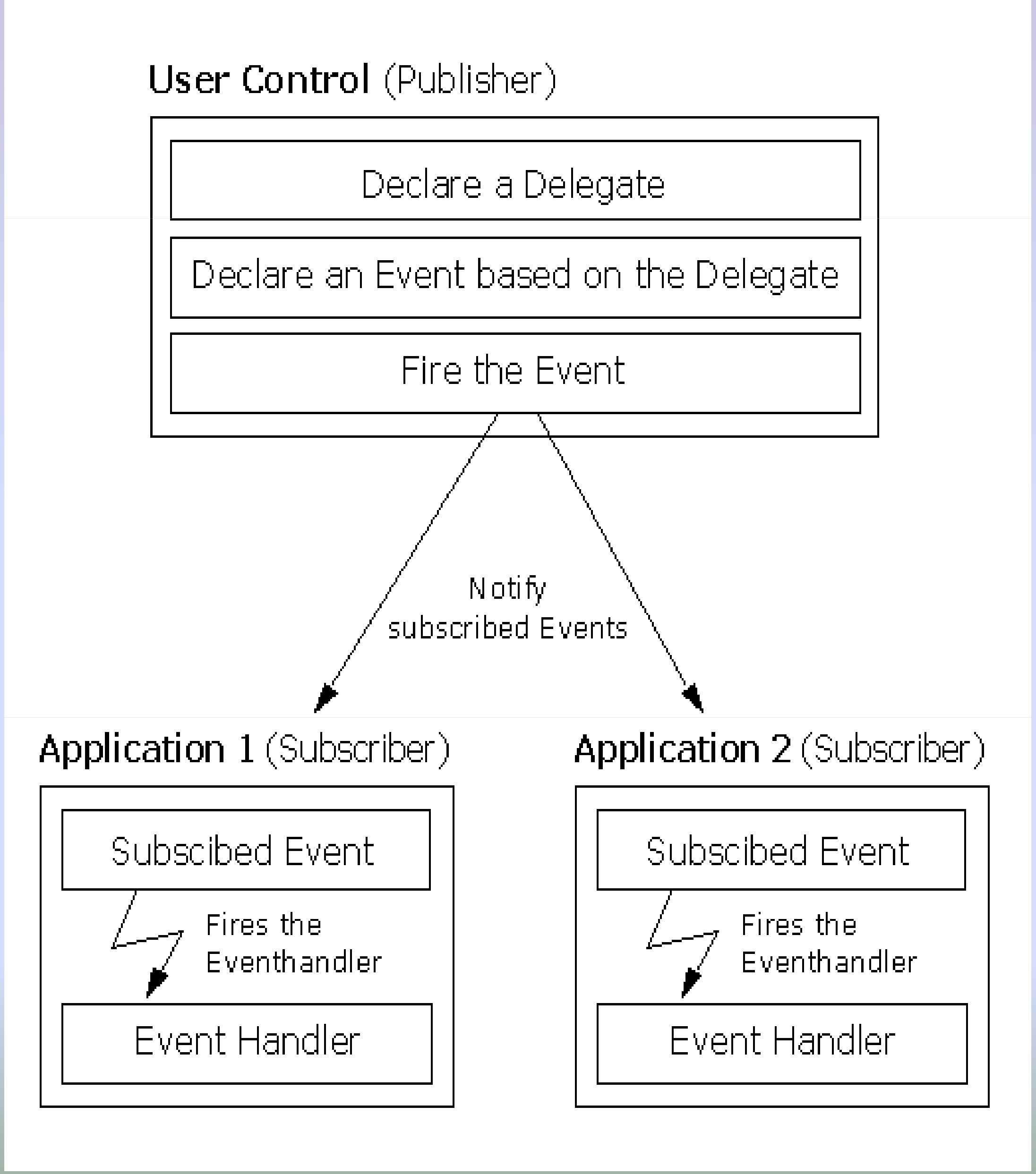
```
using System;

namespace SampleApp {
   public delegate string MyDel(string str);

class EventProgram {
      event MyDel MyEvent;

   public EventProgram()
     {
        this.MyEvent += new
     MyDel(this.WelcomeUser);
      }
      public string WelcomeUser(string
     username)
      {
        return "Welcome " + username;
       }
static void Main(string[] args)
{
         EventProgram obj1 = new
     EventProgram();
         string result = obj1.MyEvent("Tutorials
     Point");
         Console.WriteLine(result);
       }
    }
}
```

3.8 Publish and Subscribe event



•**The publisher and the subscribers are decoupled by the delegate**.

•This is highly desirable as it makes for more flexible and robust code.

3.9 Conventions 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.

3.10 Event handler

- In C#, event handler takes two parameters as input and return the void.
- The first parameter of the Event is also known as the source, which will publish the object.
- The publisher will decide when to raise the Event, and the subscriber will determine what response to give.
- Event can contain many subscribers.
- Generally, we used the Event for the single user action like clicking on the button.
- If the Event includes the multiple subscribers, then event handler is synchronously invoked.

References

- Applied Microsoft .Net Programming, Jeffrey Richter, Microsoft Press, 2002
- http://msdn.microsoft.com
- http://www.csharphelp.com/
- http://www.csharp-station.com/
- http://www.csharpindex.com/
- http://www.c-sharpcorner.com/
- https://www.w3schools.com/cs/
- https://www.akadia.com/services/dotnet_delegates_and_events .html
- Balagurusamy “Programming in C#”