**Lecture 9 – Delegates And Events**

Two side interaction, GUI events concepts

A [delegate](http://msdn.microsoft.com/en-us/library/900fyy8e.aspx) is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are nothing more than methods that are invoked through delegates. You create a custom method, and a class such as a windows control can call your method when a certain event occurs. The following example shows a delegate declaration:

C#

public delegate int PerformCalculation(int x, int y);

Any method from any accessible class or struct that matches the delegate type can be assigned to the delegate. The method can be either static or an instance method. This makes it possible to programmatically change method calls, and also plug new code into existing classes.

This ability to refer to a method as a parameter makes delegates ideal for defining callback methods. For example, a reference to a method that compares two objects could be passed as an argument to a sort algorithm. Because the comparison code is in a separate procedure, the sort algorithm can be written in a more general way.

[Delegates Overview](javascript:void(0))

Delegates have the following properties:

* Delegates are like C++ function pointers but are type safe.
* Delegates allow methods to be passed as parameters.
* Delegates can be used to define callback methods.
* Delegates can be chained together; for example, multiple methods can be called on a single event.
* Methods do not have to match the delegate type exactly. For more information, see [Using Variance in Delegates (C# and Visual Basic)](http://msdn.microsoft.com/en-us/library/ms173174.aspx).
* C# version 2.0 introduced the concept of [Anonymous Methods](http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx), which allow code blocks to be passed as parameters in place of a separately defined method. C# 3.0 introduced lambda expressions as a more concise way of writing inline code blocks. Both anonymous methods and lambda expressions (in certain contexts) are compiled to delegate types. Together, these features are now known as anonymous functions.

A [delegate](http://msdn.microsoft.com/en-us/library/900fyy8e.aspx) is a type that safely encapsulates a method, similar to a function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type safe, and secure. The type of a delegate is defined by the name of the delegate. The following example declares a delegate named Del that can encapsulate a method that takes a [string](http://msdn.microsoft.com/en-us/library/362314fe.aspx) as an argument and returns [void](http://msdn.microsoft.com/en-us/library/yah0tteb.aspx):

C#

public delegate void Del(string message);

A delegate object is normally constructed by providing the name of the method the delegate will wrap, or with an [anonymous Method](http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx). Once a delegate is instantiated, a method call made to the delegate will be passed by the delegate to that method. The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method is returned to the caller by the delegate. This is known as invoking the delegate. An instantiated delegate can be invoked as if it were the wrapped method itself. For example:

C#

// Create a method for a delegate.

public static void DelegateMethod(string message)

{

System.Console.WriteLine(message);

}

C#

// Instantiate the delegate.

Del handler = DelegateMethod;

// Call the delegate.

handler("Hello World");

Delegate types are derived from the [Delegate](http://msdn.microsoft.com/en-us/library/system.delegate.aspx) class in the .NET Framework. Delegate types are [sealed](http://msdn.microsoft.com/en-us/library/88c54tsw.aspx)—they cannot be derived from— and it is not possible to derive custom classes from [Delegate](http://msdn.microsoft.com/en-us/library/system.delegate.aspx). Because the instantiated delegate is an object, it can be passed as a parameter, or assigned to a property. This allows a method to accept a delegate as a parameter, and call the delegate at some later time. This is known as an asynchronous callback, and is a common method of notifying a caller when a long process has completed. When a delegate is used in this fashion, the code using the delegate does not need any knowledge of the implementation of the method being used. The functionality is similar to the encapsulation interfaces provide.

Another common use of callbacks is defining a custom comparison method and passing that delegate to a sort method. It allows the caller's code to become part of the sort algorithm. The following example method uses the Del type as a parameter:

C#

public void MethodWithCallback(int param1, int param2, Del callback)

{

callback("The number is: " + (param1 + param2).ToString());

}

You can then pass the delegate created above to that method:

C#

MethodWithCallback(1, 2, handler);

and receive the following output to the console:

The number is: 3

Using the delegate as an abstraction, MethodWithCallback does not need to call the console directly—it does not have to be designed with a console in mind. What MethodWithCallback does is simply prepare a string and pass the string to another method. This is especially powerful since a delegated method can use any number of parameters.

When a delegate is constructed to wrap an instance method, the delegate references both the instance and the method. A delegate has no knowledge of the instance type aside from the method it wraps, so a delegate can refer to any type of object as long as there is a method on that object that matches the delegate signature. When a delegate is constructed to wrap a static method, it only references the method. Consider the following declarations:

C#

public class MethodClass

{

public void Method1(string message) { }

public void Method2(string message) { }

}

Along with the static DelegateMethod shown previously, we now have three methods that can be wrapped by a Del instance.

A delegate can call more than one method when invoked. This is referred to as multicasting. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+='). For example:

C#

MethodClass obj = new MethodClass();

Del d1 = obj.Method1;

Del d2 = obj.Method2;

Del d3 = DelegateMethod;

//Both types of assignment are valid.

Del allMethodsDelegate = d1 + d2;

allMethodsDelegate += d3;

At this point allMethodsDelegate contains three methods in its invocation list—Method1, Method2, and DelegateMethod. The original three delegates, d1, d2, and d3, remain unchanged. When allMethodsDelegate is invoked, all three methods are called in order. If the delegate uses reference parameters, the reference is passed sequentially to each of the three methods in turn, and any changes by one method are visible to the next method. When any of the methods throws an exception that is not caught within the method, that exception is passed to the caller of the delegate and no subsequent methods in the invocation list are called. If the delegate has a return value and/or out parameters, it returns the return value and parameters of the last method invoked. To remove a method from the invocation list, use the decrement or decrement assignment operator ('-' or '-='). For example:

C#

//remove Method1

allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2

Del oneMethodDelegate = allMethodsDelegate - d2;

Because delegate types are derived from **System.Delegate**, the methods and properties defined by that class can be called on the delegate. For example, to find the number of methods in a delegate's invocation list, you may write:

C#

int invocationCount = d1.GetInvocationList().GetLength(0);

Delegates with more than one method in their invocation list derive from [MulticastDelegate](http://msdn.microsoft.com/en-us/library/system.multicastdelegate.aspx), which is a subclass of **System.Delegate**. The above code works in either case because both classes support **GetInvocationList**.

Multicast delegates are used extensively in event handling. Event source objects send event notifications to recipient objects that have registered to receive that event. To register for an event, the recipient creates a method designed to handle the event, then creates a delegate for that method and passes the delegate to the event source. The source calls the delegate when the event occurs. The delegate then calls the event handling method on the recipient, delivering the event data. The delegate type for a given event is defined by the event source. For more, see [Events (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/awbftdfh.aspx).

Comparing delegates of two different types assigned at compile-time will result in a compilation error. If the delegate instances are statically of the type **System.Delegate**, then the comparison is allowed, but will return false at run time. For example:

C#

delegate void Delegate1();

delegate void Delegate2();

static void method(Delegate1 d, Delegate2 e, System.Delegate f)

{

// Compile-time error.

//Console.WriteLine(d == e);

// OK at compile-time. False if the run-time type of f

// is not the same as that of d.

System.Console.WriteLine(d == f);

}

**Anonymous methods**

In versions of C# before 2.0, the only way to declare a [delegate](http://msdn.microsoft.com/en-us/library/900fyy8e.aspx) was to use [named methods](http://msdn.microsoft.com/en-us/library/98dc08ac.aspx). 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. There is one case in which an anonymous method provides functionality not found in lambda expressions. Anonymous methods enable you to omit the parameter list. This means that an anonymous method can be converted to delegates with a variety of signatures. This is not possible with lambda expressions. For more information specifically about lambda expressions, see [Lambda Expressions (C# Programming Guide)](http://msdn.microsoft.com/en-us/library/bb397687.aspx).

Creating anonymous methods is essentially a way to pass a code block as a delegate parameter. Here are two examples:

C#

// Create a handler for a click event.

button1.Click += delegate(System.Object o, System.EventArgs e)

{ System.Windows.Forms.MessageBox.Show("Click!"); };

C#

// Create a delegate.

delegate void Del(int x);

// Instantiate the delegate using an anonymous method.

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

By using anonymous methods, you reduce the coding overhead in instantiating delegates because you do not have to create a separate method.

For example, specifying a code block instead of a delegate can be useful in a situation when having to create a method might seem an unnecessary overhead. A good example would be when you start a new thread. This class creates a thread and also contains the code that the thread executes without creating an additional method for the delegate.

C#

void StartThread()

{

System.Threading.Thread t1 = new System.Threading.Thread

(delegate()

{

System.Console.Write("Hello, ");

System.Console.WriteLine("World!");

});

t1.Start();

}

[Remarks](javascript:void(0))

The scope of the parameters of an anonymous method is the *anonymous-method-block*.

It is an error to have a jump statement, such as [goto](http://msdn.microsoft.com/en-us/library/13940fs2.aspx), [break](http://msdn.microsoft.com/en-us/library/adbctzc4.aspx), or [continue](http://msdn.microsoft.com/en-us/library/923ahwt1.aspx), inside the anonymous method block if the target is outside the block. It is also an error to have a jump statement, such as **goto**, **break**, or**continue**, outside the anonymous method block if the target is inside the block.

The local variables and parameters whose scope contains an anonymous method declaration are called *outer* variables of the anonymous method. For example, in the following code segment, n is an outer variable:

C#

int n = 0;

Del d = delegate() { System.Console.WriteLine("Copy #:{0}", ++n); };

A reference to the outer variable n is said to be *captured* when the delegate is created. Unlike local variables, the lifetime of a captured variable extends until the delegates that reference the anonymous methods are eligible for garbage collection.

An anonymous method cannot access the [ref](http://msdn.microsoft.com/en-us/library/14akc2c7.aspx) or [out](http://msdn.microsoft.com/en-us/library/t3c3bfhx.aspx) parameters of an outer scope.

No unsafe code can be accessed within the *anonymous-method-block*.

Anonymous methods are not allowed on the left side of the [is](http://msdn.microsoft.com/en-us/library/scekt9xw.aspx) operator.

[Example](javascript:void(0))

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.

C#

// 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.

\*/

**Event**

An event is a member that enables an object or class to provide notifications. Clients can attach executable code for events by supplying event handlers.

Events are declared using *event-declaration*s:

*event-declaration:*

*attributesoptevent-modifiersopt*event   *type   variable-declarators*;  
*attributesoptevent-modifiersopt*event   *type   member-name*{   *event-accessor-declarations*}

*event-modifiers:*

*event-modifier*  
*event-modifiers   event-modifier*

*event-modifier:*

new  
public  
protected  
internal  
private  
static  
virtual  
sealed  
override  
abstract  
extern

*event-accessor-declarations:*

*add-accessor-declaration   remove-accessor-declaration*  
*remove-accessor-declaration   add-accessor-declaration*

*add-accessor-declaration:*

*attributesopt*add   *block*

*remove-accessor-declaration:*

*attributesopt*remove   *block*

An *event-declaration* may include a set of *attributes* and a valid combination of the four access modifiers, the new, static, virtual, override, sealed, abstract, and extern modifiers.

Event declarations are subject to the same rules as method declarations with regard to valid combinations of modifiers.

The *type* of an event declaration must be a *delegate-type*, and that *delegate-type* must be at least as accessible as the event itself.

An event declaration may include *event-accessor-declaration*s. However, if it does not, for non-extern, non-abstract events, the compiler supplies them automatically; for extern events, the accessors are provided externally.

An event declaration that omits *event-accessor-declaration*s defines one or more events — one for each of the *variable-declarator*s. The attributes and modifiers apply to all of the members declared by such an *event-declaration*.

It is a compile-time error for an *event-declaration* to include both the abstract modifier and brace-delimited *event-accessor-declaration*s.

When an event declaration includes an extern modifier, the event is said to be an external event. Because an external event declaration provides no actual implementation, it is an error for it to include both the externmodifier and *event-accessor-declaration*s.

An event can be used as the left-hand operand of the += and -= operators. These operators are used, respectively, to attach event handlers to or to remove event handlers from an event, and the access modifiers of the event control the contexts in which such operations are permitted.

Since += and -= are the only operations that are permitted on an event outside the type that declares the event, external code can add and remove handlers for an event, but cannot in any other way obtain or modify the underlying list of event handlers.

In an operation of the form x += y or x -= y, when x is an event and the reference takes place outside the type that contains the declaration of x, the result of the operation has type void (as opposed to having the type of x, with the value of x after the assignment). This rule prohibits external code from indirectly examining the underlying delegate of an event.

The following example shows how event handlers are attached to instances of the Button class:

public delegate void EventHandler(object sender, EventArgs e);

public class Button: Control

{

public event EventHandler Click;

}

public class LoginDialog: Form

{

Button OkButton;

Button CancelButton;

public LoginDialog() {

OkButton = new Button(...);

OkButton.Click += new EventHandler(OkButtonClick);

CancelButton = new Button(...);

CancelButton.Click += new EventHandler(CancelButtonClick);

}

void OkButtonClick(object sender, EventArgs e) {

// Handle OkButton.Click event

}

void CancelButtonClick(object sender, EventArgs e) {

// Handle CancelButton.Click event

}

}

Here, the LoginDialog instance constructor creates two Button instances and attaches event handlers to the Click events.

The keword Event Specifies an event.

[attributes] [modifiers] event type declarator;

[attributes] [modifiers] event type member-name {accessor-declarations};

where:

*attributes*(optional)

Optional declarative information. For more information on attributes and attribute classes, see [C# Attributes](http://msdn.microsoft.com/en-us/library/aa287992(v=vs.71).aspx).

*modifiers*(optional)

Optional modifiers that include:

* [abstract](http://msdn.microsoft.com/en-us/library/sf985hc5(v=vs.71).aspx)
* [new](http://msdn.microsoft.com/en-us/library/51y09td4(v=vs.71).aspx)
* [override](http://msdn.microsoft.com/en-us/library/ebca9ah3(v=vs.71).aspx)
* [static](http://msdn.microsoft.com/en-us/library/98f28cdx(v=vs.71).aspx)
* [virtual](http://msdn.microsoft.com/en-us/library/9fkccyh4(v=vs.71).aspx)
* [extern](http://msdn.microsoft.com/en-us/library/e59b22c5(v=vs.71).aspx)
* one of the four [access modifiers](http://msdn.microsoft.com/en-us/library/wxh6fsc7(v=vs.71).aspx)

*type*

The [delegate](http://msdn.microsoft.com/en-us/library/900fyy8e(v=vs.71).aspx) to which you want to associate this event.

*declarator*

The name of the event.

*member-name*

The name of the event.

*accessor-declarations*(optional)

Declaration of the accessors, which are used to add and remove event handlers in client code. The accessor functions are add and remove. It is an error to define one but not the other.

#### **Remarks**

The **event** keyword lets you specify a delegate that will be called upon the occurrence of some "event" in your code. The delegate can have one or more associated methods that will be called when your code indicates that the event has occurred. An event in one program can be made available to other programs that target the .NET Framework common language runtime.

The following steps must be taken in order to create and use C# events:

1. Create or identify a delegate. If you are defining your own event, you must also ensure that there is a delegate to use with the event keyword. If the event is predefined, in the .NET Framework for example, then consumers of the event need only know the name of the delegate.
2. Create a class that contains:
   1. An event created from the delegate.
   2. (optional) A method that verifies that an instance of the delegate declared with the **event** keyword exists. Otherwise, this logic must be placed in the code that fires the event.
   3. Methods that call the event. These methods can be overrides of some base class functionality.

This class defines the event.

1. Define one or more classes that connect methods to the event. Each of these classes will include:
   1. Associate one or more methods, using the += and -= operators, with the event in the base class.
   2. The definition of the method(s) that will be associated with the event.
2. Use the event:
   1. Create an object of the class that contains the event declaration.
   2. Create an object of the class that contains the event definition, using the constructor that you defined.

#### **Examples**

A basic example of defining and using events can be found in the [Events Tutorial](http://msdn.microsoft.com/en-us/library/aa645739(v=vs.71).aspx). The following examples in this topic also demonstrate events.

#### **Example Comment**

#### **Example 1 Declaring an event in an interface and implementing it in a class.**

#### **Example 2 Using a hash table to store event instances.**

#### **Example 3 Implementing, via event properties, two interfaces that have an event with the same name.**

#### **Example 1**

This example shows that it is possible to declare an event in an interface and implement it in a class:

// event\_keyword.cs

using System;

public delegate void MyDelegate(); // delegate declaration

public interface I

{

event MyDelegate MyEvent;

void FireAway();

}

public class MyClass: I

{

public event MyDelegate MyEvent;

public void FireAway()

{

if (MyEvent != null)

MyEvent();

}

}

public class MainClass

{

static private void f()

{

Console.WriteLine("This is called when the event fires.");

}

static public void Main ()

{

I i = new MyClass();

i.MyEvent += new MyDelegate(f);

i.FireAway();

}

}

#### **Example 2**

One use for *accessor-declarations* is to expose a large number of events without allocating a field for each event, but instead using a hash table to store the event instances. This is only useful if you have a very large number of events, but you expect most of the events will not be implemented.

// event\_keyword2.cs

using System;

using System.Collections;

public delegate void MyDelegate1(int i);

public delegate void MyDelegate2(string s);

public delegate void MyDelegate3(int i, object o);

public delegate void MyDelegate4();

public class PropertyEventsSample

{

private Hashtable eventTable = new Hashtable();

public event MyDelegate1 Event1

{

add

{

eventTable["Event1"] = (MyDelegate1)eventTable["Event1"] + value;

}

remove

{

eventTable["Event1"] = (MyDelegate1)eventTable["Event1"] - value;

}

}

public event MyDelegate1 Event2

{

add

{

eventTable["Event2"] = (MyDelegate1)eventTable["Event2"] + value;

}

remove

{

eventTable["Event2"] = (MyDelegate1)eventTable["Event2"] - value;

}

}

public event MyDelegate2 Event3

{

add

{

eventTable["Event3"] = (MyDelegate2)eventTable["Event3"] + value;

}

remove

{

eventTable["Event3"] = (MyDelegate2)eventTable["Event3"] - value;

}

}

public event MyDelegate3 Event4

{

add

{

eventTable["Event4"] = (MyDelegate3)eventTable["Event4"] + value;

}

remove

{

eventTable["Event4"] = (MyDelegate3)eventTable["Event4"] - value;

}

}

public event MyDelegate3 Event5

{

add

{

eventTable["Event5"] = (MyDelegate3)eventTable["Event5"] + value;

}

remove

{

eventTable["Event5"] = (MyDelegate3)eventTable["Event5"] - value;

}

}

public event MyDelegate4 Event6

{

add

{

eventTable["Event6"] = (MyDelegate4)eventTable["Event6"] + value;

}

remove

{

eventTable["Event6"] = (MyDelegate4)eventTable["Event6"] - value;

}

}

}

public class MyClass

{

public static void Main()

{

}

}

#### **Example 3**

Another use for event properties covers the situation where you are implementing two interfaces, each with an event of the same name. In such a case, you must use an explicit implementation event property:

// event\_keyword3.cs

using System;

public delegate void MyDelegate1();

public interface I1

{

event MyDelegate1 MyEvent;

}

public delegate int MyDelegate2(string s);

public interface I2

{

event MyDelegate2 MyEvent;

}

public class ExplicitEventsSample: I1, I2

{

public event MyDelegate1 MyEvent; // normal implementation of I1.MyEvent.

event MyDelegate2 I2.MyEvent // explicit implementation of I2.MyEvent

{

add

{

MyEvent2Storage += value;

}

remove

{

MyEvent2Storage -= value;

}

}

private MyDelegate2 MyEvent2Storage; // underlying storage for I2.MyEvent.

private void FireEvents()

{

if (MyEvent != null)

MyEvent();

if (MyEvent2Storage != null)

MyEvent2Storage("hello");

}

}

public class MyClass

{

public static void Main()

{

}

}

Events and Delegates

An event is a message sent by an object to signal the occurrence of an action. The action could be caused by user interaction, such as a mouse click, or it could be triggered by some other program logic. The object that raises (triggers) the event is called the event sender. The object that captures the event and responds to it is called the event receiver.

In event communication, the event sender class does not know which object or method will receive (handle) the events it raises. What is needed is an intermediary (or pointer-like mechanism) between the source and the receiver. The .NET Framework defines a special type ([Delegate](http://msdn.microsoft.com/en-us/library/system.delegate(v=vs.71).aspx)) that provides the functionality of a function pointer.

A delegate is a class that can hold a reference to a method. Unlike other classes, a delegate class has a signature, and it can hold references only to methods that match its signature. A delegate is thus equivalent to a type-safe function pointer or a callback. While delegates have other uses, the discussion here focuses on the event handling functionality of delegates. The following example shows an event delegate declaration.

C#

// AlarmEventHandler is the delegate for the Alarm event.

// AlarmEventArgs is the class that holds event data for the alarm event.

// It derives from the base class for event data, EventArgs.

public delegate void AlarmEventHandler(object sender, AlarmEventArgs e);

The syntax is similar to that of a method declaration; however, the **delegate** keyword informs the compiler that AlarmEventHandler is a delegate type.

By convention, event delegates in the .NET Framework have two parameters, the source that raised the event and the data for the event.

**Note**   A delegate declaration is sufficient to define a delegate class. The declaration supplies the signature of the delegate, and the common language runtime provides the implementation.

An instance of the AlarmEventHandler delegate can bind to any method that matches its signature, such as the AlarmRang method of the WakeMeUp class shown in the following example.

C#

public class WakeMeUp

{

// AlarmRang has the same signature as AlarmEventHandler.

public void AlarmRang(object sender, AlarmEventArgs e){...};

...

}

**To connect (wire) AlarmRang to an Alarm event:**

1. Create an instance of the AlarmEventHandler delegate that takes a reference to the AlarmRang method of the WakeMeUp instance in its constructor, as shown in the following example.

C#

// Create an instance of WakeMeUp.

//

WakeMeUp w = new WakeMeUp();

// Instantiate the event delegate.

// The C# compiler provides a constructor for event handlers that takes

// one parameter, the reference to the method that performs the

// event handling logic. The two-parameter constructor for EventHandler

// provided in the class library is intended for developers of

// compilers and other tools.

//

AlarmEventHandler alhandler = new AlarmEventHandler(w.AlarmRang);

1. Register the alhandler delegate with the Alarm event. For details and a complete sample, see [Event Sample](http://msdn.microsoft.com/en-us/library/9aackb16(v=vs.71).aspx).

Custom event delegates are needed only when an event generates event data. Many events, including some user-interface events such as mouse clicks, do not generate event data. In such situations, the event delegate provided in the class library for the no-data event, [System.EventHandler](http://msdn.microsoft.com/en-us/library/system.eventhandler(v=vs.71).aspx), is adequate. Its declaration follows.

C#

// The base class for event data, EventArgs, does not have

// any data and hence can be used as the event data type for events

// that do not generate data.

//

delegate void EventHandler(object sender, EventArgs e);

Event delegates are multicast, which means that they can hold references to more than one event handling method. For details, see [Delegate](http://msdn.microsoft.com/en-us/library/system.delegate(v=vs.71).aspx). Delegates allow for flexibility and fine-grain control in event handling. A delegate acts as an event dispatcher for the class that raises the event by maintaining a list of registered event handlers for the event.

Questions

1. What is delegate? How can we use delegates?
2. The delegates are declared?
3. What is multicast deligate ?
4. How events use delegates?
5. Can event be broadcasting ?