# Module 1: The Role of Events, Delegates and Event Handlers

## Introduction:

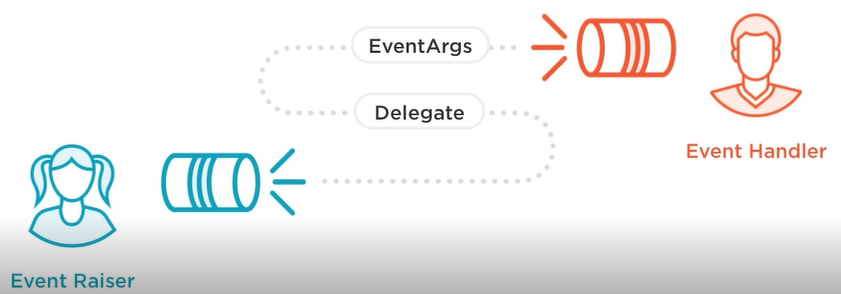
* We have here the girl is talking in, and her dad is listening, so the girl acts as “**Event Raiser**”, and the dad is the “**Event Hander**”.
* There is a glue or pipeline between “**event raiser**” and “**event handler**” this is called “**Delegate**”.
* Another important piece is the “**Event Args**”, because when the child talks in we need this “data” to get into the “**Event Handler**”.

Figure Introduction

## The Role of Events

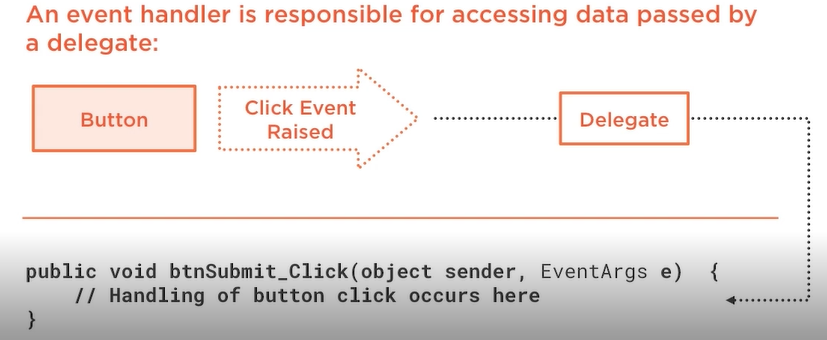
* “**Event**” is just a notification, exactly here as the girl provides notification in this case through speech to the person on the right.
* “**Events**” are notifications, it is a message that goes out to one or more” **subscribers**”, in previous example was here dad, but in .net the subscribers are “**objects**”.
* “**Events**” provides a way to trigger notifications from end users or from objects, you can think it as user that loudly announces to a group of people that something is going to happen.
* The simplest example of “**Events**” is a button.
* It is possible for event to have multiple “**Object**” listening if they are attached to the event, then they can be get notified.
* “**Event**” signal the occurrence of an action/notification.
* **“Objects**” that raises events do not need to explicitly know the object that will handle the event (receive the data).
* Most “**Events**” pass “**EventArgs**” (event data) that get routed from point A to point B.

## Role of Delegates

* The “**Delegates**” is the pipeline between an “**Event**” and “**Event handler**”, That allows “**Event**” and “**EventArgs**” to go to “**Event handler**”.
* A “**Delegate**” is a specialized class often called a “**Function Pointer**”.
* Based on a “**MultiCastDelegate**” base class, it is a class that tracks everyone listening, so when the “**Events**” notification goes off, then it should be sent to all listeners.
* So, before we raise the “**event**” we have the “**EventArgs**” (our data that we want move from point A to point B), and then we send “**Event Args**” through pipeline (**Delegates**) into the “**Event** **Handler**”.
* We call “**Delegates**” a “**function** **pointer**” because the “**event** **handler**” will be a function, so we need to point the data through the pipeline into “**Event handler**”.

Figure Delegates

## Role of Event Handlers

* “**Event** **Handler**” is a responsible for receiving and processing data from a “**delegate**”.
* It receives and process “**EventArgs**” Data.
* It normally receives 2 parameters: 1- “**Sender**” who send to you, and it will be an object, 2- “**EventArgs**”, that is responsible for encapsulating “**event** **data**”
* When a button is click, event is raised, then notification, then send the “**Sender**” and “**EventArgs**” through delegates to the “**event** **handler**”.

# Module 2: Creating Delegates, Events and EventArgs

## Introduction

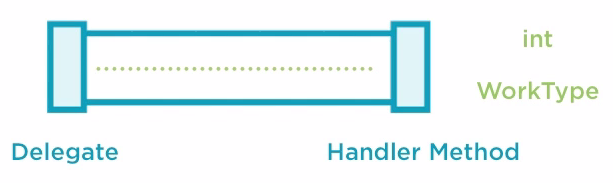
This module concentrates on:

1. The process of creating a delegate.
2. Defining an event.
3. Raising events and the proper way to do that
4. Creating an “**EventArgs**” class

## Creating Delegates

* What delegates are? And how to create custom delegate?
* Custom delegates are defined using the “delegate” keyword

public **delegate** void WorkPerformedHandler**(**int hours**,** workType workType**);**

* When we write “delegate” behind the scene, when the compiler see the “delegate” keyword, it automatically generate a class that inherits from another .net delegate classes.
* You can think of the above created delegate as a one way pipeline, the name of the delegate is “**WorkPerformedHandler**”, and it takes “int” and “workType” (it could be enumeration).
* The delegate keyword acts as a shell for a defined method, so the delegate acts as blueprint for the method (event handler) that the data will be get dumped into.
* The delegate is a pipeline, and what we want is to dump data from point A to point B (handler method).
* In below fig the pipeline accepts only 2 parameters (“int” and “workType”)
* The delegate signature must be mimicked by a handler method, so in previous case the “**handler**” should accept (“int” and “workType”)

**public** void Manager**-**WorkPerformed**(**int workHours**,** WorkType wType**)**

**{}**

* What matters here are the parameters types, but the parameter name can be changed.

### What happen behind the scene?

* .Net has a couple of abstract base classes, and one of these core classes that provide some basic functionality is called “**Delegate**”.
* “**Delegate**” class has 2 properties [**Method**, **Target**], [**GetInvocationList**()]
* “**Method**” -> the pipeline has to dump data into somewhere, and this defines the name of the method that the data should go.
* “**Target**” -> if you have object instance that this method lives in, then the target will be the actual object that has that method.
* “**MulticastDelegate**”, every delegate you create once compiled, it will inherit from “**MulticastDelegate**”, so it is a way to hold multiple delegates, in other words we might have one message that I need to send it out, but I want to send it across multiple pipelines, so you can imagine “**MulticastDelegate**” as have multi pipelines that dump in different methods.
* “Custom **Delegate**”, can inherits from “**MulticastDelegate**”, but you cannot do this inherits directly, but the way you do is to use the “**delegate**” keyword (it is compiler trick), as this is very specific base classes that the compiler blocks from inheriting from them unless you use the “**delegate**” keyword, and so once the compiler see the “delegate” signature, it will automatically generate a class that inherits from “MulticastDelegate”.
* “**MulticastDelegate**”:
  + Image you have a number of pipelines that dumps into different places in a list, so pipeline one goes to method 1, pipeline 2 goes to method 2,…, this is called “**InvocationList**”
  + Can reference more than one delegate function
  + Tracks delegate references using invocation list
  + It acts as array of multiple pipelines (**delegates**)
  + Delegates in the list are invoked sequentially

### Creating a Delegate Instance

* Once we created our “**custom** **delegate**” we need to create the “**handler**” somewhere that the data will go.

**Delegate:** **public** **delegate** void WorkPerformedHandler**(**int hours**,** workType workType**);**

**Delegate** **Instance:** WorkPerdormedHandler del1 **=** **new** WorkPerdormedHandler**(**WorkPerformed1**)**

**Handler:** **public** void WorkPerformed1**(**int workHours**,** WorkType wType**)**

**{**Console.WriteLine**(**"WorkPerformed1 called"**)}**

* Now we need to hook up the “**delegate**” and the “**event handler**” using “**Delegate instance**”
* So once the compiler see the delegate signature, it will create class that inherits from “**MulticastDelegate**”
* So we can use this “delegate” as a class, we can create instance of this class and pass to its constructor the “method handler name” in the “**delegate** **instance**”, when this delegate is invoked the handler should be called.

### Invoking a Delegate

* To invoked a method you should call it as a method

**Delegate Instance:** WorkPerdormedHandler del1 **=** **new** WorkPerdormedHandler**(**WorkPerformed1**)**

**Invoke Delegate exactly like a method:** del1**(**5**,** WorkType**.**Golf**)**

**Handler:** **public** void WorkPerformed1**(**int workHours**,** WorkType wType**)**

**{**Console.WriteLine**(**"WorkPerformed1 called"**)}**

* Now we have only 1 pipeline (and 1 place to go) in the “**invocationList**”, so now we need to add multiple
* In below example we have 2 delegates and they points to a different functions here.
* We need when **del1** is instantiated and invoked, we need it also to invoke the second delegate
* What we are doing here that we adding **del2** into the “**invocationList**” that has only 1 item (**del1**).
* And when we invoke **del1**, these parameters will go to both these 2 handlers
* This provides a simple way to wire up punch of notifications, so with only 1 call I can notify in this case 2 handler methods

Delegate Instance**:** WorkPerdormedHandler del1 **=** **new** WorkPerdormedHandler**(**WorkPerformed1**)**

Delegate Instance**:** WorkPerdormedHandler del2 **=** **new** WorkPerdormedHandler**(**WorkPerformed1**)**

del1 **+=** del2**;**

del1**(**5**,** WorkType**.**Golf**)**

## 6-Defining an Event

* We will talk about the process of associating delegate with an event, so as an event is raised we can move data up to the listener (event handler).
* Event can be defined in a class using the “event” keyword.

**public** **event** WorkPerformedHandler WorkPerformed**;**

**where** WorkPerformedHandler is our delegate

**where** WorkPerformed is event name

* Event is really friendly wrapper around delegate, although you can use “delegate” on its own as shown previously, but we are going to use “event” because they are easy, and it is standard way of providing notifications.
* So in this case, listeners can go in and attached to “WorkPerformed” event, behind the scene they are adding themselves to the **invocationList** of the “WorkPerformedHandler”
* In addition to simple way of using “event” keyword, the “delegate” in the event name, if you want more control over how the “listener” added/remove to the invocationList so we can use Add/Remove accessor.
* Events can be defined using add/remove accessors

**public** **event** WorkPerformedHandler workperformed

**{**

**[**MethodImpl**(**MethodImplOptions**.**Synchronized**)]**

add

**{**

\_WorkPerformedHandler **=** **(**WorkPerformedHandler**)**Delegate**.**Combine**(**\_WorkPerformedHandler**,** value**);**

**//previous line acts as += in delegates**

**}**

**[**MethodImpl**(**MethodImplOptions**.**Synchronized**)]**

remove

**{**

\_WorkPerformedHandler **=** **(**WorkPerformedHandler**)**Delegate**.**Remove**(**\_WorkPerformedHandler**,** value**);**

**//previous line acts as -= in delegates**

**}**

**}**

* Why we are doing the previous? There may be occasions, where you have some logic on when the listeners can be attached and when it cannot be attached, if you need more controls in add/removing from invocation list, then you can use add/remove accessors

## 8-Raising Events

* Once the event is defined you need a way to raise the event.
* Events are raised by calling the event like a method, because what is behind event is a delegate, and we invoke a delegate exactly like calling a method.
* But before invoke event, you have to check if it is not null (is there anything in the invocationList) to prevent exception from being fired.

**if(**WorkPerformed **!=** **null)**

**{**

WorkPerformed**(**8**,** WorkType**.**GenerateReports**);**

**}**

* Another option is to access the event’s delegate and invoke it directly, by casting the event as delegate

WorkPerformedHandler del **=** WorkPerformed **as** WorkPerformHandler**;**

**if(**del **!=** **null)**

**{**

del**(**8**,** WorkType**.**GenerateReports**);**

**}**

### Exposing and Raising Events

**public** **delegate** void WorkPerformHandler**(**int hours**,** WorkType workType**);**

**public** class Worker

**{**

// Event definition is hooked up to the delegate

**public** **event** WorkPerformHandler WorkPerformed**; 🡨------ “Event Definition”**

**public** **virtual** void DoWork**(**int hours**,** WorkType workType**)**

**{**

// Do work here and notify consumer that work has been performed

// this method handle the raising of the event.

OnWorkPerformed**(**hours**,** workType**);**

**}**

**protected** **virtual** void OnWorkPerformed **(**int hours**,** WorkType workType**)**

**{**

WorkPerformHandler del = WorkPerformed as WorkPerformHandler;

**if(**del **!=** **null)** // Listeners are attached

**{**

**// now any listener in the invocationList will be notified**

**// by the bellowed raised event**

del**(**hours**,** workType**); 🡨-----Raise Event**

**}**

**}**

**}**

* The best practice is to take the name of the event and make a method called “OnEventName”