LU4 - Event Handling

▼ Overview

In this section we will learn about events and how to use events in unity to “trigger” certain parts of our code.

We have a range of events available to us. These include and are not limited to:

- Delegates
- Unity Event
- Action<in>
- Func<in, out>

With events we can create code that is modular and remove dependencies in our code by abstracting them to events.
We do not care who are the subscribers to our events, we just know that we invoke them and whoever responds, does so.

Events use the **Publisher-Subscriber** model we also use events in the **Observer Pattern**

The class who defines and raises the events is called the **publisher** class.

Some other class that receives a notification is called the **subscriber**

Publishers raises an event when some action occurred. The Subscribers who are interested in getting notified when an action occurred, should register with an event and handle it.

When using the **event** keyword with your “delegates”, “Action”, “Func” you will no longer be able to invoke or raise a “delegates”, “Action”, “Func” from an external class. So no class outside of the publisher class raise your event.

▼ Delegates

Delegates simply are variables that holds a signature of method instead of data.

```
public delegate void DoSomething();
public delegate void DoSomethingWithParameters(string parameterA);
public delegate string DoReturn();
public delegate string DoReturnWithParameters(string parameterA);
```

To use a delegate, create an “instance” of the delegate signature.

```
// Create Instance
public DoSomething Anything;

// Random Method matching Delegate Signature
public void PrintHelloWorld() => Debug.Log("Hello, World!");
```

After creating an “instance” of the delegate, you can assign or subscribe methods that match its signature to it.

```
// Assign or Subsribe a method to your delegate that has a matching signature
Anything += PrintHelloWorld;
```

You can also un-assign or unsubscribe methods that you previously added to the delegate in a similar manner

```
// Un-assign a method
Anything -= PrintHelloWorld;
```

To Invoke or Raise the delegate we simply use the .Invoke() method of the delegate

```
// Invoke or Raise your Delegate to notify its Subscribers
Anything?.Invoke();
// ?. is called null propagation it would be written as
if(Anything != null) Anything.Invoke()
```

Now how do we pass parameters to a delegate

```
// Blueprint of your Delegate
public delegate void DoSomethingPara(int a);

// Instance of your Delegate
public DoSomethingPara AnythingPara;

// Some Random method matching the signature of your Delegate
```

```
public void PrintAge(int age) => Debug.Log($"My Age is {age}");

// Assign or Subscribe the method to your delegate
AnythingPara += PrintAge;

// Invoke or Raise the delegate
Anything?.Invoke(28);

// Un-assign or UnSubscribe from your delegate
AnythingPara -= PrintAge;
```

What about delegates that return values

```
public delegate void DoReturn(int a, int b);

public DoReturn ReturnSum;

public void SumOf(int a, int b) => return a + b;

ReturnSum += SumOf;

int result = ReturnSum?.Invoke(20, 8);

ReturnSum -= SumOf;
```

▼ Action<optional in>

Actions are short hand for delegates of type void. Actions can also take in up to 16 parameters of any type.
Actions do not need “instances” to be created before using them.

```
// Define your Actions as Fields of your class
public Action SomeAction;
public Action<int> SomeActionWithPara;

// Random Methods
public void PrintHelloUnity() => Debug.Log("Hello, Unity!");
public void PrintAge(int age) => print($"My Age is {age}");

// Assign or Subscribe methods with a similar signiture to your Actions
SomeAction += PrintHelloUnity;
SomeActionWithPara += PrintAge;

// Invoke your actions at some point in your code
SomeAction?.Invoke();
SomeActionWithPara?.Invoke(28);

// Clean up your Subscribers (Done on the Subscriber Class)
SomeAction -= PrintHelloUnity;
SomeActionWithPara -= PrintAge;
```

You can subscribe to an Action anonymously you can do the following:

```
public Action MyAction;

// Breakdown
// () -> Refers to the parameters of your Action, in this case no parameters
// {} -> This is where all the logic of your Anonymous method goes e.g. print("Choco");
MyAction += () => { // Do stuff here };

// The downfall of assigning anonymouse methods to your Actions are
// they cannot be removed.
// You will have to set the Action to null to remove any Subscribers.
// This only happens at the next GC Call
```

▼ Func<optional in, required out>

Func are short hand for delegates that return a type. Func can also take in up to 16 parameters of any type and must return a single type.
Func do not need “instances” to be created before using them.

```
// Define your Func as Fields of your class
public Func<string> SomeFunc;
public Func<int, int, int> SomeFuncWithPara;

// Random Methods
public string SayHelloUnity() => return "Hello, Unity!";
public int SumOf(int a, int b) => return a + b;

// Assign or Subscribe methods with a similar signiture to your Func
SomeFunc += SayHelloUnity;
SomeFuncWithPara += SumOf;

// Invoke your Func at some point in your code
string message = SomeFunc?.Invoke();
int age = SomeFuncWithPara?.Invoke(20, 8);

// Clean up your Subscribers (Done in the Subscriber Class)
SomeFunc -= SayHelloUnity;
SomeFuncWithPara -= SumOf;
```

You can subscribe to an Func anonymously you can do the following:

```
public Func<int, string> MyAction;

// Breakdown
// () -> Refers to the parameters of your func, in this case we have int parameter
// {} -> This is where all the logic of your Anonymous method goes e.g. print("Choco");
```

```
MyAction += (age) => {
  // Do stuff here
  return $"{age}";
};

string message = MyAction(10);

// The downfall of assigning anonymouse methods to your Func are
// they cannot be removed.
// You will have to set the Func to null to remove any Subscribers.
// This only happens at the next GC Call
```

▼ Unity Events

Unity Events are event structures that allow you to create an event like actions within the inspector.

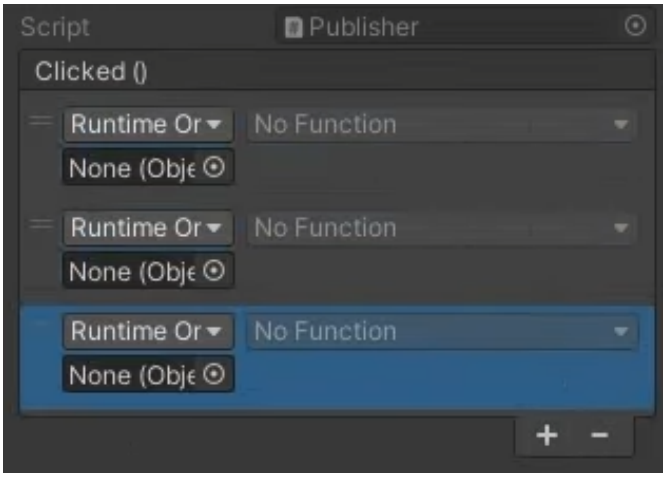
They give you a visual representation of the subscribers of your events and are invoked in a similar manner a traditional event would.

If you have ever used a button in Unity and added methods to the Button OnClick event, this OnClick event is a UnityEvent

To create your own unity event, you simply need the following.

```
UnityEvent Clicked = new UnityEvent();
```

In the Inspector you UnityEvent will look like this



To Invoke or Raise Unity Events you do the following

```
Clicked?.Invoke();
```

You can assign or subscribe to Unity events via code.

```
Clicked.AddListeners(YourMethod);
```

To un-assign or unsubscribe from the Unity event you can do the following

```
Clicked.RemoveListener(YourMethod);
// or
Clicked.RemoveListeners();
```

UnityEvents unlike other traditional “delegates”, “Action”, “Func” they require additional work if you wish to extend their functionality. The base functionality remains when subscribing and unsubscribing in the inspector or via code.

For example, if you wish to pass only ints to your UnityEvent, you will need to create a serializable class inheriting from the UnityEvent class.

For example

```
[Serializable]
public class UnityIntEvent : UnityEvent<int>
{
}
```

To use this new UnityIntEvent would be similar to the original

```
public IntEvent = new UnityIntEvent();
```