Delegates

.NET

*A **delegate** 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 (call) the method through the **delegate** instance.*

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/

Binding - Overview

https://docs.microsoft.com/en-us/dotnet/desktop-wpf/data/data-binding-overview#what-is-data-binding
https://docs.microsoft.com/en-us/dotnet/csharp/delegates-overview
https://en.wikipedia.org/wiki/Late_binding#Late_binding_in_.NET

Binding - This process establishes a connection between an app UI and the data it displays. When the data changes its value, the elements (in memory) that are bound to the data reflect changes automatically. If an outer representation of the data in an element changes, then the underlying data will reflect the change.

- Early Binding (AKA,Static Binding, AKA, Static Dispatch, AKA, Compile-Time Binding) – The compiler (or linker) directly associates a stack memory address to the function call. It replaces the call with a machine language instruction that tells the mainframe to leap to the address of the function.
- Late Binding (AKA Dynamic Binding, AKA Dynamic Dispatch, AKA, Dynamic Linkage, AKA, Run-Time Binding) - When an algorithm is created that uses a caller-supplied method that implements part of the algorithm. (In C++ these are known as function pointers.) In .NET, late binding refers to overriding a virtual method. The compiler builds virtual tables for every virtual or interface method call which is used at run-time to determine the implementation to execute.

Delegates provide a ***late binding*** mechanism in .NET.

Delegate – Overview

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/

Delegates are used to pass methods as arguments to other methods. This ability to refer to a method as a parameter makes ***delegates*** ideal for defining callback methods.

A ***delegate*** is a ***type*** that represents references to methods with a 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.

This example declares a ***delegate*** named Del that can encapsulate a method that takes a string as an argument and returns void:

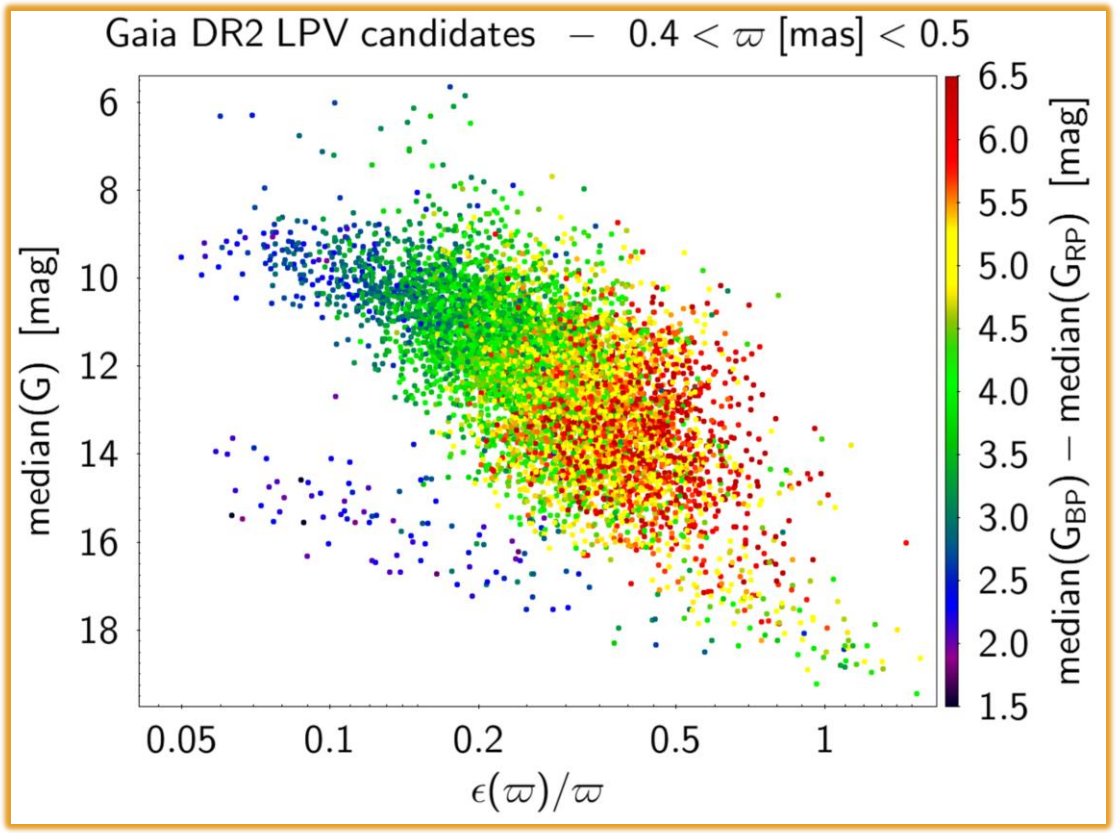
```
public delegate void Del(string message);
```

ANY method from ANY accessible class or struct that matches the ***delegate*** type and return type can be assigned to the ***delegate***. ***Delegates*** are object-oriented, type safe, and secure.

Delegate – Usage Example

Consider sorting a list of stars in an astronomy application. You may choose to sort those stars by their distance from the earth, or the magnitude of the star, or their perceived brightness.

In all those cases, the Sort() method does essentially the same thing: arranges the items in the list based on some comparison. The code that compares two stars is different for each of the sort orderings.



Delegate – Step by Step

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/using-delegates

1. Create a method to assign to a delegate.

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

2. Create a delegate type (named ***Del)*** *with the same return type and argument.*

```
public delegate void Del(string message);
```

3. Instantiate a delegate type variable and assign to it the chosen function (DelegateMethod). The delegate variable is what gets passed to a function.

```
// Instantiate the delegate.
Del handler = DelegateMethod;
```

4. Invoke the delegate variable. If the original method (DelegateMethod) returns a value, it will be returned through the delegate variabel (handler).

```
// Call the delegate.
handler("Hello World");
```

Delegate – Multicasting

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/how-to-combine-delegates-multicast-delegates

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 (ordered) method invocation list use the addition assignment operators ('+' or '+=').

allMethodsDelegate is a ***delegate type*** object that points to other ***delegate type*** objects that point to a ***delegate*** variable that refers to a function or invocation list. ***Arguments*** are passed from ***delegate*** obj to ***delegate*** obj to function.

```
public class MethodClass
{
    public void Method1(string message) { }
    public void Method2(string message) { }
}
```

```
var 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;
```

Delegate - Multicasting

Reference arguments are passed sequentially to each method in turn. Any changes by one method are visible to the next method.

If any method in the method invocation list throws an uncaught exception, execution stops and the exception is passed to the caller of the ***delegate***.

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 subtraction or subtraction assignment operators (- or -=).

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Del oneMethodDelegate = allMethodsDelegate - d2;
```

Action Delegate

https://docs.microsoft.com/en-us/dotnet/api/system.action?view=netframework-4.8

You can use an ***Action delegate*** to pass a method as a parameter without explicitly declaring a custom ***delegate***. The encapsulated method must correspond to the method signature that is defined by this ***delegate***. This means that the encapsulated method must have no parameters and no return value.

Instantiate an Action ***delegate*** instead of explicitly defining a new ***delegate*** and assigning a named method to it.

```
using System;
using System.Windows.Forms;

public class Name
{
   private string instanceName;

   public Name(string name)
   {
      this.instanceName = name;
   }

   public void DisplayToConsole()
   {
      Console.WriteLine(this.instanceName);
   }

   public void DisplayToWindow()
   {
      MessageBox.Show(this.instanceName);
   }
}

public class testTestDelegate
{
   public static void Main()
   {
      Name testName = new Name("Koani");
      Action showMethod = testName.DisplayToWindow;
      showMethod();
   }
}
```

Func<TResult> Delegate

https://docs.microsoft.com/en-us/dotnet/api/system.func-1?view=netframework-4.8

TResult = The type of the return value of the method that the ***Func<TResult> delegate*** encapsulates. This type parameter is covariant (use your specified type or a more derived type).

Use ***Func<TResult> delegate*** to represent a method that can be passed as a parameter without explicitly declaring a custom ***delegate***. The encapsulated method must correspond to the method signature that is defined by this ***delegate***. ***Func<TResult>*** means that the encapsulated method must have no parameters and must return a value.

Func<T,TResult>, **Func<T1, T2, TResult>**, *etc, are variations that have arguments and return types.*

Func<TResult> Delegate - Example

https://docs.microsoft.com/en-us/dotnet/api/system.func-1?view=netframework-4.8#examples

```
using System;

static class Func1
{
   public static void Main()
   {
      // Note that each lambda expression has no parameters.
      LazyValue<int> lazyOne = new LazyValue<int>(() => ExpensiveOne());
      LazyValue<long> lazyTwo = new LazyValue<long>(() => ExpensiveTwo("apple"));

      Console.WriteLine("LazyValue objects have been created.");

      // Get the values of the LazyValue objects.
      Console.WriteLine(lazyOne.Value);
      Console.WriteLine(lazyTwo.Value);
   }

   static int ExpensiveOne()
   {
      Console.WriteLine("\nExpensiveOne() is executing.");
      return 1;
   }

   static long ExpensiveTwo(string input)
   {
      Console.WriteLine("\nExpensiveTwo() is executing.");
      return (long)input.Length;
   }
}
```

```
class LazyValue<T> where T : struct
{
   private Nullable<T> val;
   private Func<T> getValue;

   // Constructor.
   public LazyValue(Func<T> func)
   {
      val = null;
      getValue = func;
   }

   public T Value
   {
      get
      {
         if (val == null)
            // Execute the delegate.
            val = getValue();
         return (T)val;
      }
   }
}
/* The example produces the following output:

    LazyValue objects have been created.

    ExpensiveOne() is executing.
    1

    ExpensiveTwo() is executing.
    5
*/
```

Events – Overview

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/

Events are a special kind of ***multicast delegate*** that can only be invoked from within the class or struct where they are declared (the publisher class). If other classes or structs subscribe to the event, their event handler methods will be called when the publisher class raises the event.

Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or *raises*) the **event** is called the **publisher** and the classes that receive (or *handle*) the **event** are called **subscribers**.

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.

Events – Properties

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/

- The ***publisher*** determines when an ***event*** is raised; the ***subscribers*** determine what action is taken in response to the ***event***.
- An ***event*** can have multiple ***subscribers***. A ***subscriber*** can handle multiple ***events*** from multiple ***publishers***.
- ***Events*** that have no ***subscribers*** are never raised.
- ***Events*** are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- When an ***event*** has multiple ***subscribers***, the event handlers are invoked synchronously when an ***event*** is raised.
- In the .NET Framework class library, ***events*** are based on the ***EventHandler delegate*** and the ***EventArgs*** base class.

Events – Publishing Based on the ***EventHandler*** Pattern

https://docs.microsoft.com/en-us/dotnet/api/system.eventhandler?view=netframework-4.8
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-publish-events-that-conform-to-net-framework-guidelines

---

All ***events*** in the .NET Framework class library are based on the ***EventHandler*** delegate. The standard signature of an ***eventHandler delegate*** defines a method that does not return a value. This method's first parameter is of ***type*** Object and refers to the instance that raises the ***event***. Its second parameter is derived from ***type EventArgs*** and holds the ***event*** data.

```
[System.Runtime.InteropServices.ComVisible(true)]
[System.Serializable]
public delegate void EventHandler(object sender, EventArgs e);
```

It is recommended that all events be based on the .NET Framework pattern by using ***EventHandler***.

Events – Publishing Based on the ***EventHandler*** Pattern

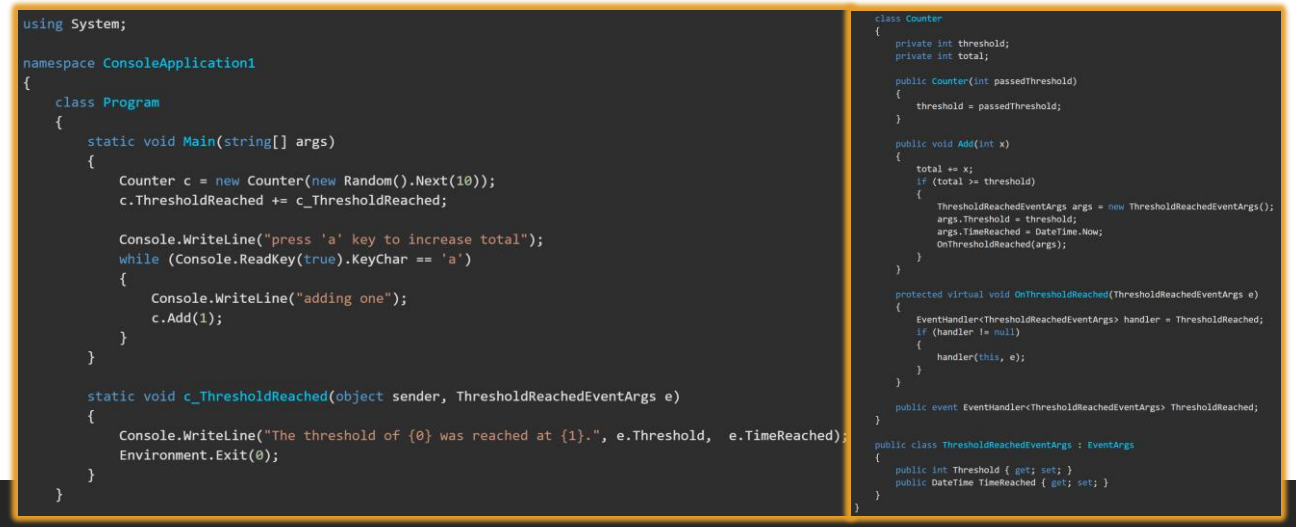
https://docs.microsoft.com/en-us/dotnet/api/system.eventhandler?view=netframework-4.8#examples

---

This example shows an event named 'ThresholdReached' that is associated with an 'EventHandler' delegate. The method assigned to the ***EventHandler*** delegate is called in the 'OnThresholdReached' method.

A delegate connects an event with its handler. To raise an event, two elements are needed:

- A delegate identifying the method that provides the response to the event.
- (Optional) A class that holds the event data, if provided.



```
using System;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            Counter c = new Counter(new Random().Next(10));
            c.ThresholdReached += c_ThresholdReached;

            Console.WriteLine("press 'a' key to increase total");
            while (Console.ReadKey(true).KeyChar == 'a')
            {
                Console.WriteLine("adding one");
                c.Add(1);
            }
        }

        static void c_ThresholdReached(object sender, ThresholdReachedEventArgs e)
        {
            Console.WriteLine("The threshold of {0} was reached at {1}.", e.Threshold,  e.TimeReached);
            Environment.Exit(0);
        }
    }
```

```
class Counter
{
    private int threshold;
    private int total;

    public Counter(int passedThreshold)
    {
        threshold = passedThreshold;
    }

    public void Add(int x)
    {
        total += x;
        if (total >= threshold)
        {
            ThresholdReachedEventArgs args = new ThresholdReachedEventArgs();
            args.Threshold = threshold;
            args.TimeReached = DateTime.Now;
            OnThresholdReached(args);
        }
    }

    protected virtual void OnThresholdReached(ThresholdReachedEventArgs e)
    {
        EventHandler<ThresholdReachedEventArgs> handler = ThresholdReached;
        if (handler != null)
        {
            handler(this, e);
        }
    }

    public event EventHandler<ThresholdReachedEventArgs> ThresholdReached;
}

public class ThresholdReachedEventArgs : EventArgs
{
    public int Threshold { get; set; }
    public DateTime TimeReached { get; set; }
}
```

Events – Publishing Based on the ***EventHandler*** Pattern

https://docs.microsoft.com/en-us/dotnet/api/system.eventhandler?view=netframework-4.8#examples
https://docs.microsoft.com/en-us/dotnet/standard/events/?view=netframework-4.8

The EventHandler ***delegate*** is a predefined ***delegate*** that specifically represents an ***event*** handler method for an ***event*** that does not generate data. If your ***event*** does generate data, you must use the generic EventHandler<TEventArgs> ***delegate*** class.

To associate the ***event*** with the method that will handle the ***event***, add an instance of the ***eventHandler delegate*** to the event. The ***event*** handler is called whenever the ***event*** occurs, unless you remove the ***delegate***.

The ***delegate*** model follows the observer design pattern, which enables a ***subscriber*** to register with and receive notifications from a ***provider***. An ***event*** sender pushes a notification that an event has happened, and an ***event*** receiver receives that notification and defines a response to it.

Events – Publishing Based on the ***EventHandler*** Pattern

https://docs.microsoft.com/en-us/dotnet/api/system.eventhandler?view=netframework-4.8#examples
https://docs.microsoft.com/en-us/dotnet/standard/events/?view=netframework-4.8

The object that raises the ***event*** is called the ***event*** sender. The ***event*** is typically a member of the ***event*** sender.

To define an ***event***, use the ***event*** keyword in the signature of your event class and specify the type of ***delegate*** for the event. Add a method that is marked as ***protected*** and ***virtual***, conventionally named 'On[EventName]' ('OnDataReceived'). The method should take one parameter that specifies an ***event*** data object of ***type EventArgs*** or a derived ***type***. Derived classes can override this method to change the logic for raising the ***event***. A derived class should always call the On[EventName] method of the base class to ensure that registered ***delegates*** receive the event.

```
class Counter
{
    public event EventHandler ThresholdReached;

    protected virtual void OnThresholdReached(EventArgs e)
    {
        EventHandler handler = ThresholdReached;
        handler?.Invoke(this, e);
    }

    // provide remaining implementation for the class
}
```

Events – Publishing Based on the ***EventHandler*** Pattern – Step by Step

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-publish-events-that-conform-to-net-framework-guidelines

---

1. (If not sending custom data with your event, go to 3a) Declare the class for your custom data at a scope that is visible to both your publisher and subscriber classes. Then, add the required members to hold your custom event data. ➔ ➔ ➔ ➔ ➔ ➔ ➔

```
public class CustomEventArgs : EventArgs
{
    public CustomEventArgs(string s)
    {
        msg = s;
    }
    private string msg;
    public string Message
    {
        get { return msg; }
    }
}
```

2. (Skip if using generic version of *EventHandler<TEventArgs>*). Declare a delegate in your publishing class. Use a name ending with ***EventHandler***. The second parameter specifies your custom ***EventArgs*** type.

```
public delegate void CustomEventHandler(object sender, CustomEventArgs a);
```

Events – Publishing Based on the ***EventHandler*** Pattern – Step by Step

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-publish-events-that-conform-to-net-framework-guidelines

---

3. Declare the event in your publishing class by using one of the following steps.

a. If you have no custom EventArgs class, your Event type will be the non-generic EventHandler delegate. You do not have to declare the delegate because it is already declared in the System namespace. Add the following code to your publisher class.

```
public event EventHandler RaiseCustomEvent;
```

b. If you are using the non-generic version of EventHandler AND have a custom class derived from EventArgs, declare your event inside your publishing class and use your delegate from step 2 as the type.

```
public event CustomEventHandler RaiseCustomEvent;
```

c. If you are using the generic version, you do not need a custom delegate. Instead, in your publishing class, you specify your event type as ***EventHandler<[CustomEventArgs]>***, substituting the name of your own class between the angle brackets.

```
public event EventHandler<CustomEventArgs> RaiseCustomEvent;
```

Events – Subscribing

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-subscribe-to-and-unsubscribe-from-events#to-subscribe-to-events-programmatically

1. Define an event handler method whose signature matches the delegate signature for the event.

```
void HandleCustomEvent(object sender, CustomEventArgs a)
{
    // Do something useful here.
}
```

2. Use the addition assignment operator (+=) to attach an event handler to the event. In the following example, assume that an object named publisher has an event named RaiseCustomEvent.

```
publisher.RaiseCustomEvent += HandleCustomEvent;
```

Events – Unsubscribing

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/how-to-subscribe-to-and-unsubscribe-from-events#unsubscribing

---

To prevent your ***event handler*** from being invoked when the ***event*** is raised, ***unsubscribe*** from the event. In order to prevent resource leaks, you should ***unsubscribe*** from ***events*** before you dispose of a ***subscriber*** object. Until you ***unsubscribe*** from an event, the multicast ***delegate*** that underlies the ***event*** in the ***publishing*** object has a reference to the ***delegate*** that encapsulates the ***subscriber's event handler***. As long as the ***publishing*** object holds that reference, ***garbage collection*** will not delete your ***subscriber*** object.

```
publisher.RaiseCustomEvent -= HandleCustomEvent;
```

When all subscribers have unsubscribed from an event, the event instance in the publisher class is set to null.