## 1. Delegates and Events

1. A delegate is a pointer to method

Ex

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

1. Using delegates

public delegate void Del(string message);

// Create a method for a delegate.

public static void DelegateMethod(string message)

{

Console.WriteLine(message);

}

// Instantiate the delegate.

Del handler = DelegateMethod;

// Call the delegate.

handler("Hello World");

1. Delegates allow us the freedom to pass different methods conforming to the delegate signature to another method

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

{

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

}

MethodWithCallback(1, 2, handler);

1. A delegate can wrap a method inside an object. The delegate does not care about the class (object).

public class MethodClass

{

public void Method1(string message) { }

public void Method2(string message) { }

}

A delegate can call more than one method when invoked. This is referred to as multicasting.

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;

1. 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:

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

1. Using a delegate

// Declare a delegate

delegate void Del(int i, double j);

class MathClass

{

static void Main()

{

MathClass m = new MathClass();

// Delegate instantiation using "MultiplyNumbers"

Del d = m.MultiplyNumbers;

// Invoke the delegate object.

Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");

for (int i = 1; i <= 5; i++)

{

d(i, 2);

}

// Keep the console window open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

// Declare the associated method.

void MultiplyNumbers(int m, double n)

{

Console.Write(m \* n + " ");

}

}

/\* Output:

Invoking the delegate using 'MultiplyNumbers':

2 4 6 8 10

\*/

## 2. Multicast Delegates

using System;

// Define a custom delegate that has a string parameter and returns void.

delegate void CustomDel(string s);

class TestClass

{

// Define two methods that have the same signature as CustomDel.

static void Hello(string s)

{

Console.WriteLine($" Hello, {s}!");

}

static void Goodbye(string s)

{

Console.WriteLine($" Goodbye, {s}!");

}

static void Main()

{

// Declare instances of the custom delegate.

CustomDel hiDel, byeDel, multiDel, multiMinusHiDel;

// In this example, you can omit the custom delegate if you

// want to and use Action<string> instead.

//Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

// Create the delegate object hiDel that references the

// method Hello.

hiDel = Hello;

// Create the delegate object byeDel that references the

// method Goodbye.

byeDel = Goodbye;

// The two delegates, hiDel and byeDel, are combined to form multiDel.

multiDel = hiDel + byeDel;

// Remove hiDel from the multicast delegate, leaving byeDel,

// which calls only the method Goodbye.

multiMinusHiDel = multiDel - hiDel;

Console.WriteLine("Invoking delegate hiDel:");

hiDel("A");

Console.WriteLine("Invoking delegate byeDel:");

byeDel("B");

Console.WriteLine("Invoking delegate multiDel:");

multiDel("C");

Console.WriteLine("Invoking delegate multiMinusHiDel:");

multiMinusHiDel("D");

}

}

/\* Output:

Invoking delegate hiDel:

Hello, A!

Invoking delegate byeDel:

Goodbye, B!

Invoking delegate multiDel:

Hello, C!

Goodbye, C!

Invoking delegate multiMinusHiDel:

Goodbye, D!

\*/

## 3. C# - Action Delegate

Action<int> Print ⬄ public delegate void Print(int val);

Action is a delegate type defined in the System namespace. An Action type delegate is the same as [Func delegate](https://www.tutorialsteacher.com/csharp/csharp-func-delegate) except that the Action delegate doesn't return a value. In other words, an Action delegate can be used with a method that has a void return type.

For example, the following delegate prints an int value.

public delegate void Print(int val);

static void ConsolePrint(int i)

{

Console.WriteLine(i);

}

static void Main(string[] args)

{

Print prnt = ConsolePrint;

prnt(10);

}

- You can use an Action delegate instead of defining the above Print delegate, for example:

Example: Action delegate

static void ConsolePrint(int i)

{

Console.WriteLine(i);

}

static void Main(string[] args)

{

Action<int> printActionDel = ConsolePrint;

printActionDel(10);

}

An Action delegate can take up to 16 input parameters of different types.

An Anonymous method can also be assigned to an Action delegate, for example:

Example: Anonymous method with Action delegate

static void Main(string[] args)

{

Action<int> printActionDel = delegate(int i)

{

Console.WriteLine(i);

};

printActionDel(10);

}

## 4. C# - Func Delegate

C# includes built-in generic delegate types Func and Action, so that you don't need to define custom delegates manually in most cases.

Func is a generic delegate included in the System namespace. It has zero or more *input* parameters and one *out* parameter. The last parameter is considered as an out parameter.

The Func delegate that takes one input parameter and one out parameter is defined in the System namespace, as shown below:

Signature: Func

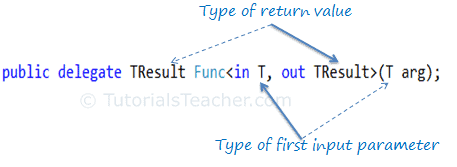
namespace System

{

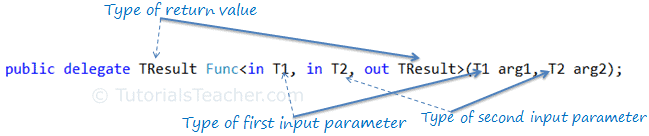
public delegate TResult Func<in T, out TResult>(T arg);

}

The last parameter in the angle brackets <> is considered the return type, and the remaining parameters are considered input parameter types, as shown in the following figure.

[](https://www.tutorialsteacher.com/Content/images/csharp/func-delegate.png)Func delegate

A Func delegate with two input parameters and one out parameters will be represented as shown below.

[](https://www.tutorialsteacher.com/Content/images/csharp/func-delegate2.png)Func delegate

The following Func delegate takes two input parameters of int type and returns a value of int type:

Func<int, int, int> sum;

You can assign any method to the above func delegate that takes two *int* parameters and returns an *int* value.

Example: Func

class Program

{

static int Sum(int x, int y)

{

return x + y;

}

static void Main(string[] args)

{

Func<int,int, int> add = Sum;

int result = add(10, 10);

Console.WriteLine(result);

}

}

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-2yWptk)

Output:

20

A Func delegate type can include 0 to 16 input parameters of different types. However, it must include an out parameter for the result. For example, the following Func delegate doesn't have any input parameter, and it includes only an out parameter.

Example: Func with Zero Input Parameter

Func<int> getRandomNumber;

As if it was writen

Func(,int) getOnly outoput of type int.

## 5. Observer Design Pattern

The pattern defines a provider (also known as a subject or an observable) and zero, one, or more observers. Observers register with the provider, and whenever a predefined condition, event, or state change occurs, the provider automatically notifies all observers by calling one of their methods.

 In .NET, the observer design pattern is applied by implementing the generic [System.IObservable<T>](https://learn.microsoft.com/en-us/dotnet/api/system.iobservable-1) and [System.IObserver<T>](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1) interfaces. The generic type parameter represents the type that provides notification information.

Implementing the pattern requires that you provide the following:

A provider or subject, which is the object that sends notifications to observers. A provider is a class or structure that implements the [IObservable<T>](https://learn.microsoft.com/en-us/dotnet/api/system.iobservable-1) interface. The provider must implement a single method, [IObservable<T>.Subscribe](https://learn.microsoft.com/en-us/dotnet/api/system.iobservable-1.subscribe), which is called by observers that wish to receive notifications from the provider.

An observer, which is an object that receives notifications from a provider. An observer is a class or structure that implements the [IObserver<T>](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1) interface. The observer must implement three methods, all of which are called by the provider:

[IObserver<T>.OnNext](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1.onnext), which supplies the observer with new or current information.

[IObserver<T>.OnError](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1.onerror), which informs the observer that an error has occurred.

[IObserver<T>.OnCompleted](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1.oncompleted), which indicates that the provider has finished sending notifications.

## 6. Implementing the Pattern

The following example uses the observer design pattern to implement an airport baggage claim information system. A BaggageInfo class provides information about arriving flights and the carousels where baggage from each flight is available for pickup. It is shown in the following example.

A BaggageHandler class is responsible for receiving information about arriving flights and baggage claim carousels. Internally, it maintains two collections:

* observers - A collection of clients that will receive updated information.
* flights - A collection of flights and their assigned carousels.

Both collections are represented by generic [List<T>](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) objects that are instantiated in the BaggageHandler class constructor. The source code for the BaggageHandler class is shown in the following example.

Clients that wish to receive updated information call the BaggageHandler.Subscribe method. If the client has not previously subscribed to notifications, a reference to the client's [IObserver<T>](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1) implementation is added to the observers collection.

The overloaded BaggageHandler.BaggageStatus method can be called to indicate that baggage from a flight either is being unloaded or is no longer being unloaded. In the first case, the method is passed a flight number, the airport from which the flight originated, and the carousel where baggage is being unloaded. In the second case, the method is passed only a flight number. For baggage that is being unloaded, the method checks whether the BaggageInfo information passed to the method exists in the flights collection. If it does not, the method adds the information and calls each observer's OnNext method. For flights whose baggage is no longer being unloaded, the method checks whether information on that flight is stored in the flights collection. If it is, the method calls each observer's OnNext method and removes the BaggageInfo object from the flights collection.

When the last flight of the day has landed and its baggage has been processed, the BaggageHandler.LastBaggageClaimed method is called. This method calls each observer's OnCompleted method to indicate that all notifications have completed, and then clears the observers collection.

The provider's [Subscribe](https://learn.microsoft.com/en-us/dotnet/api/system.iobservable-1.subscribe) method returns an [IDisposable](https://learn.microsoft.com/en-us/dotnet/api/system.idisposable) implementation that enables observers to stop receiving notifications before the [OnCompleted](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1.oncompleted) method is called. The source code for this Unsubscriber(Of BaggageInfo) class is shown in the following example. When the class is instantiated in the BaggageHandler.Subscribe method, it is passed a reference to the observers collection and a reference to the observer that is added to the collection. These references are assigned to local variables. When the object's Dispose method is called, it checks whether the observer still exists in the observers collection, and, if it does, removes the observer.

he following example provides an [IObserver<T>](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1) implementation named ArrivalsMonitor, which is a base class that displays baggage claim information. The information is displayed alphabetically, by the name of the originating city. The methods of ArrivalsMonitor are marked as overridable (in Visual Basic) or virtual (in C#), so they can all be overridden by a derived class.

The ArrivalsMonitor class includes the Subscribe and Unsubscribe methods. The Subscribe method enables the class to save the [IDisposable](https://learn.microsoft.com/en-us/dotnet/api/system.idisposable) implementation returned by the call to [Subscribe](https://learn.microsoft.com/en-us/dotnet/api/system.iobservable-1.subscribe) to a private variable. The Unsubscribe method enables the class to unsubscribe from notifications by calling the provider's [Dispose](https://learn.microsoft.com/en-us/dotnet/api/system.idisposable.dispose) implementation. ArrivalsMonitor also provides implementations of the [OnNext](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1.onnext), [OnError](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1.onerror), and [OnCompleted](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1.oncompleted) methods. Only the [OnNext](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1.onnext) implementation contains a significant amount of code. The method works with a private, sorted, generic [List<T>](https://learn.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1) object that maintains information about the airports of origin for arriving flights and the carousels on which their baggage is available. If the BaggageHandler class reports a new flight arrival, the [OnNext](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1.onnext) method implementation adds information about that flight to the list. If the BaggageHandler class reports that the flight's baggage has been unloaded, the [OnNext](https://learn.microsoft.com/en-us/dotnet/api/system.iobserver-1.onnext) method removes that flight from the list. Whenever a change is made, the list is sorted and displayed to the console.

The following example contains the application entry point that instantiates the BaggageHandler class as well as two instances of the ArrivalsMonitor class, and uses the BaggageHandler.BaggageStatus method to add and remove information about arriving flights. In each case, the observers receive updates and correctly display baggage claim information.

<https://learn.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern#implementing-the-pattern>

## 7.0 Handle and raise events

## Events

An event is a message sent by an object to signal the occurrence of an action. The action can be caused by user interaction, such as a button click, or it can result from some other program logic, such as changing a property's value. The object that raises the event is called the event sender. The event sender doesn't know which object or method will receive (handle) the events it raises. The event is typically a member of the event sender; for example, the [Click](https://learn.microsoft.com/en-us/dotnet/api/system.web.ui.webcontrols.button.click) event is a member of the [Button](https://learn.microsoft.com/en-us/dotnet/api/system.web.ui.webcontrols.button) class, and the [PropertyChanged](https://learn.microsoft.com/en-us/dotnet/api/system.componentmodel.inotifypropertychanged.propertychanged) event is a member of the class that implements the [INotifyPropertyChanged](https://learn.microsoft.com/en-us/dotnet/api/system.componentmodel.inotifypropertychanged) interface.

To define an event, you use the C# [event](https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/event) or the Visual Basic [Event](https://learn.microsoft.com/en-us/dotnet/visual-basic/language-reference/statements/event-statement) keyword in the signature of your event class, and specify the type of delegate for the event. Delegates are described in the next section.

Typically, to raise an event, you add a method that is marked as protected and virtual (in C#) or Protected and Overridable (in Visual Basic). Name this method OnEventName; for example, OnDataReceived. The method should take one parameter that specifies an event data object, which is an object of type [EventArgs](https://learn.microsoft.com/en-us/dotnet/api/system.eventargs) or a derived type. You provide this method to enable derived classes to override the logic for raising the event. A derived class should always call the OnEventName method of the base class to ensure that registered delegates receive the event.

The following example shows how to declare an event named **ThresholdReached**. The event is associated with the [**EventHandler**](https://learn.microsoft.com/en-us/dotnet/api/system.eventhandler) delegate and raised in a method named **OnThresholdReached**.

class Counter

{

public event EventHandler ThresholdReached;

protected virtual void OnThresholdReached(EventArgs e)

{

ThresholdReached?.Invoke(this, e);

}

// provide remaining implementation for the class

}

EventHandler is the delegate. It is defined as follows:

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

namespace EventsInCsharp.Services   
{

public class FoodOrderingService   
 {

public event EventHandler<FoodPreparedEventArgs> FoodPrepared;

public void PrepareOrder(Order order)

{

Console.WriteLine($"Preparing your order ' {order.Item}', please wait...");

OnFoodPrepared(order);

}

protected virtual void OnFoodPrepared(Order order)

{

FoodPrepared?.Invoke(this, new FoodPreparedEventArgs { Order = order });

}

} }

##### class Program

##### {

##### 

##### static void Main(string[] args)

##### {

##### 

##### var order = new Order { Item = "Pizza with extra cheese" };

##### 

##### var orderingService = new FoodOrderingService();

##### var appService = new AppService();

var mailService = new MailService();

orderingService.FoodPrepared += appService.OnFoodPrepared;

orderingService.FoodPrepared += mailService.OnFoodPrepared;

orderingService.PrepareOrder(order);

Console.ReadKey();

}   
   
}}

public class MailService

{

public Order Order { get; set; }

public void OnFoodPrepared(object source, FoodPreparedEventArgs eventArgs)

{

Order = eventArgs.Order as Order;

Console.WriteLine($"MailService: your food '{eventArgs.Order.Item}' is prepared.");

}

}