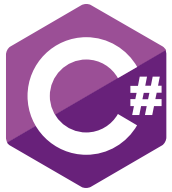
Delegates, events and lambdas

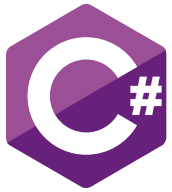
Agenda

- Delegates
- Multicast delegates
- Anonymous methods
- Events
- Handling events
- Expression body definitions

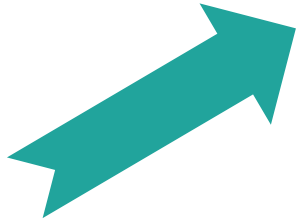
Intro



Delegates



Delegate



Signature

0 to n typed parameters

Return type

Delegates

It’s a type that defines a signature and a return type

It can be assigned a method with the same signature and return type

By invoking the delegate, the method is called

Declare and instantiate a delegate

=

Create a method contract between the delegate type and its handler method

Delegate declaration

- Delegate type : *“Here are the terms of the contract, you must have this signature and return this…”*

Delegate instantiation

- Delegate handler : *“Agreed ! Done.”*

Declaration

Can be declared in a namespace or in another type

Can be also defined in the global namespace but not a good practice

Delegate declaration

```
namespace MyNamespace
{
    public delegate int PerformCalculation(int x, int y);
}
```

Delegate declaration

```
namespace MyNamespace
{
    public class MyClass
    {
        private delegate int PerformCalculation(int x, int y);
    }
}
```

Delegate property

```
namespace MyNamespace
{
    public delegate int PerformCalculation(int x, int y);
    public class MyClass
    {
        public PerformCalculation CalculationDelegate { get; set; }
    }
}
```

Instantiation

Instantiated with the new keyword

Or can be inferred

Delegate declaration

```
PerformCalculation handler;
```

Delegate instantiation

```
PerformCalculation handler =
                    new PerformCalculation(CalculationMethod);
```

Delegate inference

```
PerformCalculation handler = CalculationMethod;
```

The handler method

```
private int CalculationMethod(int int1, int int2)
{
    return int1 + int2;
}
```

Call the delegate

```
int i = handler(7, 5);
```

Benefits

Change programmatically the behavior of a type

Event based programming

Two methods ...

```
private int CalculationMethod1(int int1, int int2)
{
    return int1 + int2;
}

private int CalculationMethod2(int int1, int int2)
{
    return Math.Abs(int1 - int2);
}
```

... One handler

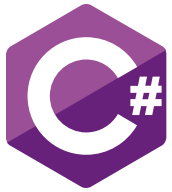
```
PerformCalculation handler = CalculationMethod1;
Console.WriteLine($"handler(7,5) = {handler(7, 5)}");
handler = CalculationMethod2;
Console.WriteLine($"handler(7,5) = {handler(7, 5)}");
// handler(7,5) = 12
// handler(7,5) = 2
```

Demo

Declare delegates

Instantiate and call delegates

Multicast delegates



Multicast delegate

A delegate can call multiple methods

Multicasting is when you call multiple methods from one delegate

The methods must have the same signature and return type

The methods attached to a delegate are part of its invocation list

Two methods

```
private int CalculationMethod1(int int1, int int2)
{
    int result = int1 + int2;
    Console.WriteLine($"Calculate {int1} + {int2}");
    return result;
}
private int CalculationMethod2(int int1, int int2)
{
    int result = int1 - int2;
    Console.WriteLine($"Calculate {int1} - {int2}");
    return result;
}
```

Multicast

```
PerformCalculation handler1 = CalculationMethod1;
PerformCalculation handler2 = CalculationMethod2;
PerformCalculation handlerWithMulticasting = handler1 + handler2;
int result = handlerWithMulticasting(7,5);
Console.WriteLine($"handlerWithMulticasting(7,5) = {result}");
```

Output

```
// Calculate 7 + 5
// Calculate 7 - 5
// handlerWithMulticasting(7,5) = 2
```

Class hierarchy

Delegates inherit from the `MulticastDelegate` base class

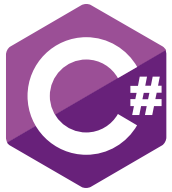
`MulticastDelegate` inherits from `Delegate` base class

Cannot be used, are derived internally after compiling

Demo

Declare, create and call multicast delegates

Anonymous methods



Anonymous methods

- Method with no name
- A delegate can be attached to an anonymous method
- Better to use lambdas

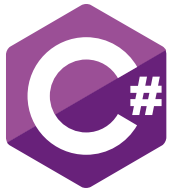
Anonymous method

```
PerformCalculation handler = delegate (int int1, int int2)
{
    return int1 + int2;
};
```

Demo

Attach a delegate to an anonymous method

Events



What is an event ?

Notification of something that happens

Provided by the publisher, raises the event

Sent to all the subscribers

Declaration

An event can be declared in a class or struct

Declared with the **event** keyword

A delegate type is specified during the event declaration

Event delegate

Declare a delegate or use the `EventHandler` class

Event declaration

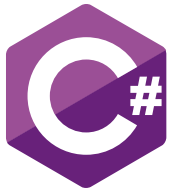
```
namespace MyNamespace
{
    public delegate void PerformTaskHandler(int taskId);
    public class MyClass
    {
        public event PerformTaskHandler TaskPerformed;
    }
}
```

Demo

Create events on a service class

Call the service work method

Handling events



Delegates

Makes the link between an event and its handler

The delegate is called when the event is raised, thereby the handler is called to

Event handler

It’s the method that will handle the event

It contains the handling logic of the event it is attached to

Event declaration and call

```
public class MyClass
{
    public event PerformTaskHandler TaskPerformed;
    public void PerformTask(int taskNumber)
    {
        // Perform a task
        TaskPerformed(taskNumber);
    }
}
```

Subscribe

You attach an event to its handler through a delegate

You must unsubscribe to avoid resource leaks

Event handler subscription

```
MyClass myClass = new();
myClass.TaskPerformed += TaskCompleted;
```

Perform task

```
myClass.PerformTask(100);
```

Unsubscribe

```
myClass.TaskPerformed -= TaskCompleted;
```

EventHandler

`EventHandler` generic class can be used instead of creating a delegate

EventHandler

```
public class MyClass
{
    public event EventHandler<PerformTaskEventArgs> TaskPerformed;
    public void PerformTask(int taskNumber)
    {
        // Perform a task
        TaskPerformed(this, new PerformTaskEventArgs(taskNumber));
    }
}
```

Event arguments

Wrap the data sent by the event raiser to the subscribers

Built-in event arguments classes can be used, but you can create your own

Derived from `EventArgs` base class

EventArgs

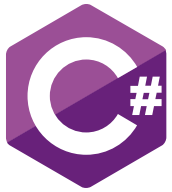
```
public class PerformTaskEventArgs : EventArgs
{
    public int TaskId { get; set; }
    public PerformTaskEventArgs(int taskId)
    {
        TaskId = taskId;
    }
}
```

Demo

Subscribe to events

Attach handler methods to events

Action, Func and lambdas



Action and Func

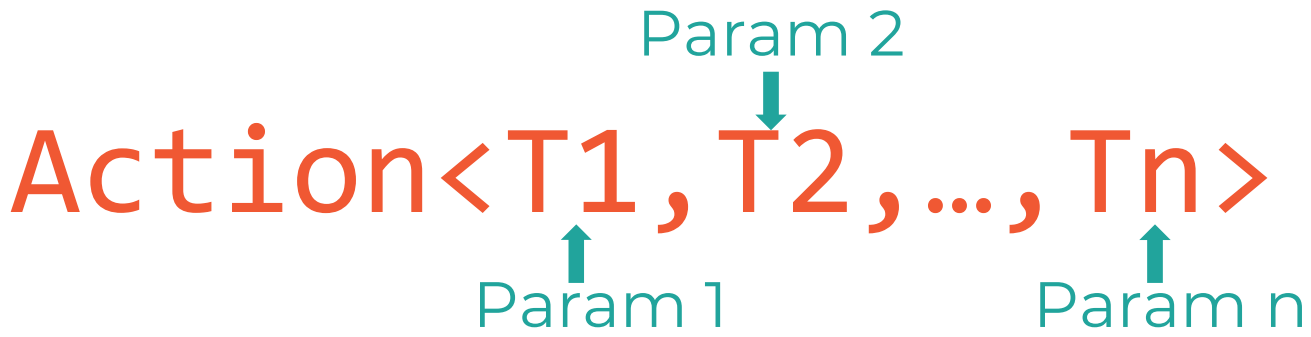
A way to define a delegate without creating one

C# delegate type

Action

Delegate with no, one or more parameters and returns void

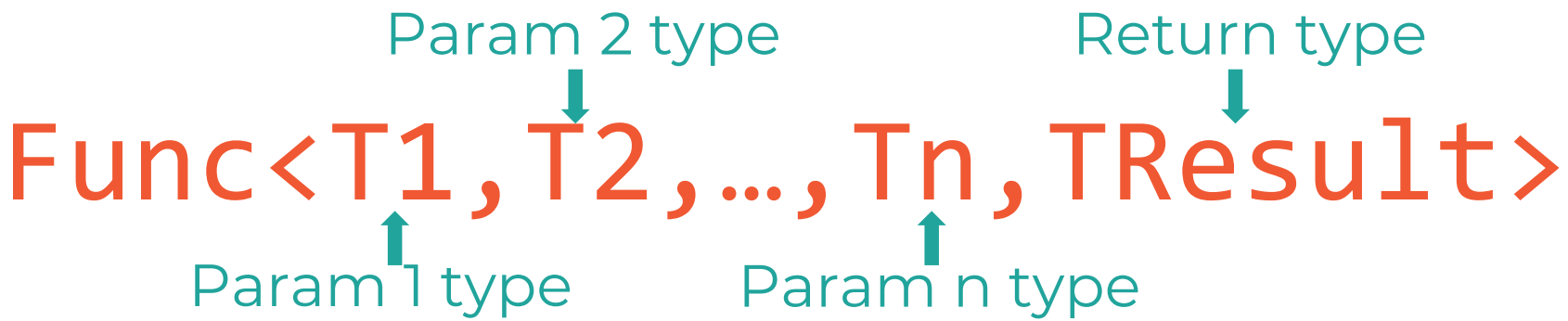
Action



Func

Delegate with no, one or more parameters and has a return type

Func


Param 2 type
Return type
Func<T1,T2,…,Tn,TResult>
Param 1 type
Param n type

Lambda expressions

To create an anonymous method

Can be converted or assigned to a delegate like an anonymous method

Lambda expressions

Expression lambda

Statement lambda

Lambda expressions

On the left side, zero, one or more parameters

On the right side, An expression or a block of statement

Separated with the lambda operator

Expression lambdas

```
Action Print = () => Console.WriteLine("I love C#");
Action<string, string> PrintKeyValue = (key, value) =>
    Console.WriteLine($"key = {key} - value = {value}");
```

Expression lambdas

```
Func<int> GiveMe5 = () => 5;
Func<int,int,int> Multiply = (int1,int2) => int1 * int2;
```

Statement lambda

```
Func<int, int, int> Multiply = (int1, int2) =>
{
    int result = int1 * int2;
    Console.WriteLine($"result : {result}");
    return result;
};
```

Demo

Create Action and Func

Use lambdas

Summary

A delegate is a type that defines a signature and a return type

A handler method can be attached to a delegate. It can be an anonymous method

An event can be raised by a publisher when something happens

A handler method can be attached to the event through a delegate (handler subscription)

Event arguments can be sent when the event is raised

You can use predefined delegates with Action and Func

A lambda expression is another way to define a delegate