



DELEGATES AND EVENTS

.NET & JS LAB, RD BELARUS

Anzhelika Kravchuk

Module Overview

- Declaring and Using Delegates
- Using Lambda Expressions
- Events

Why Decouple an Operation from a Method?

Methods determined dynamically at run time

Method A or method B or method C



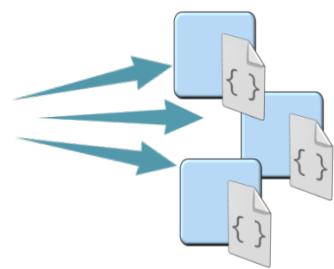
Callback methods

Enable you to specify a method (or methods) to run when an asynchronous method call completes, particularly when you use third-party assemblies



Multicast operations

Method A and method B and method C



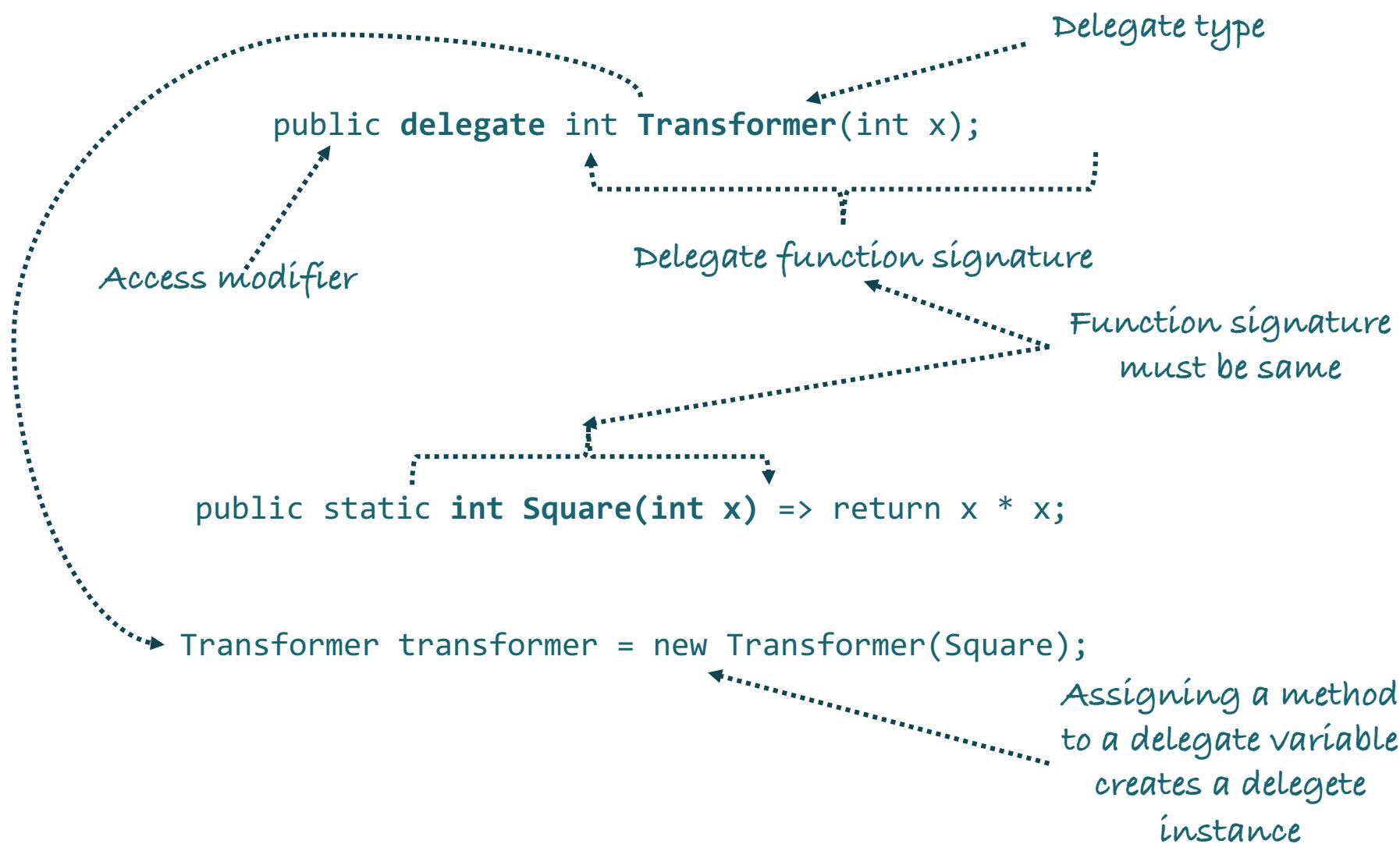
Defining a Delegate

Callback functions are an extremely useful programming mechanism that has been around for years. The Microsoft .NET Framework exposes a callback function mechanism by using delegates. Unlike callback mechanisms used in other platforms, such as unmanaged C++, delegates offer much more functionality. For example, delegates ensure that the callback method is type-safe, in keeping with one of the most important goals of the common language runtime (CLR). Delegates also integrate the ability to call multiple methods sequentially and support the calling of static methods as well as instance methods.

A delegate is an object that knows how to call a method.

A *delegate type* defines the kind of method that *delegate instances* can call. Specifically, it defines the method's *return type* and its *parameter types*. The following defines a delegate type called Transformer:

Defining a Delegate



Invoking a Delegate

```
Transformer transformer = Square;
```



The statement
is shorthand for

```
Transformer transformer = new Transformer (Square);
```

```
transformer(1);
```



Invoking Delegate
is shorthand for

```
transformer.Invoke(2);
```

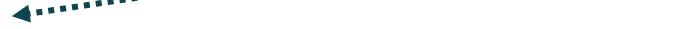


```
if (transformer != null) transformer(1);
```



Invoking Delegate

```
transformer?.Invoke(2);
```



Delegates in Depth

```
public delegate int Transformer(int x);  
  
public class Transformer: MulticastDelegate  
{  
    public Transformer(Object object, IntPtr method);  
    public virtual Int32 Invoke(Int32 x);  
    public virtual IAsyncResult BeginInvoke(Int32 x, AsyncCallback  
callback, Object object);  
    public virtual void EndInvoke(IAsyncResult result);  
}
```

Compiler C#

Delegates in Depth

Transformer

Base Types

System.MulticastDelegate

Delegate

Object

~ ICloneable

~ System.Runtime.Serialization.ISerializable

≡ .ctor(Object, IntPtr)

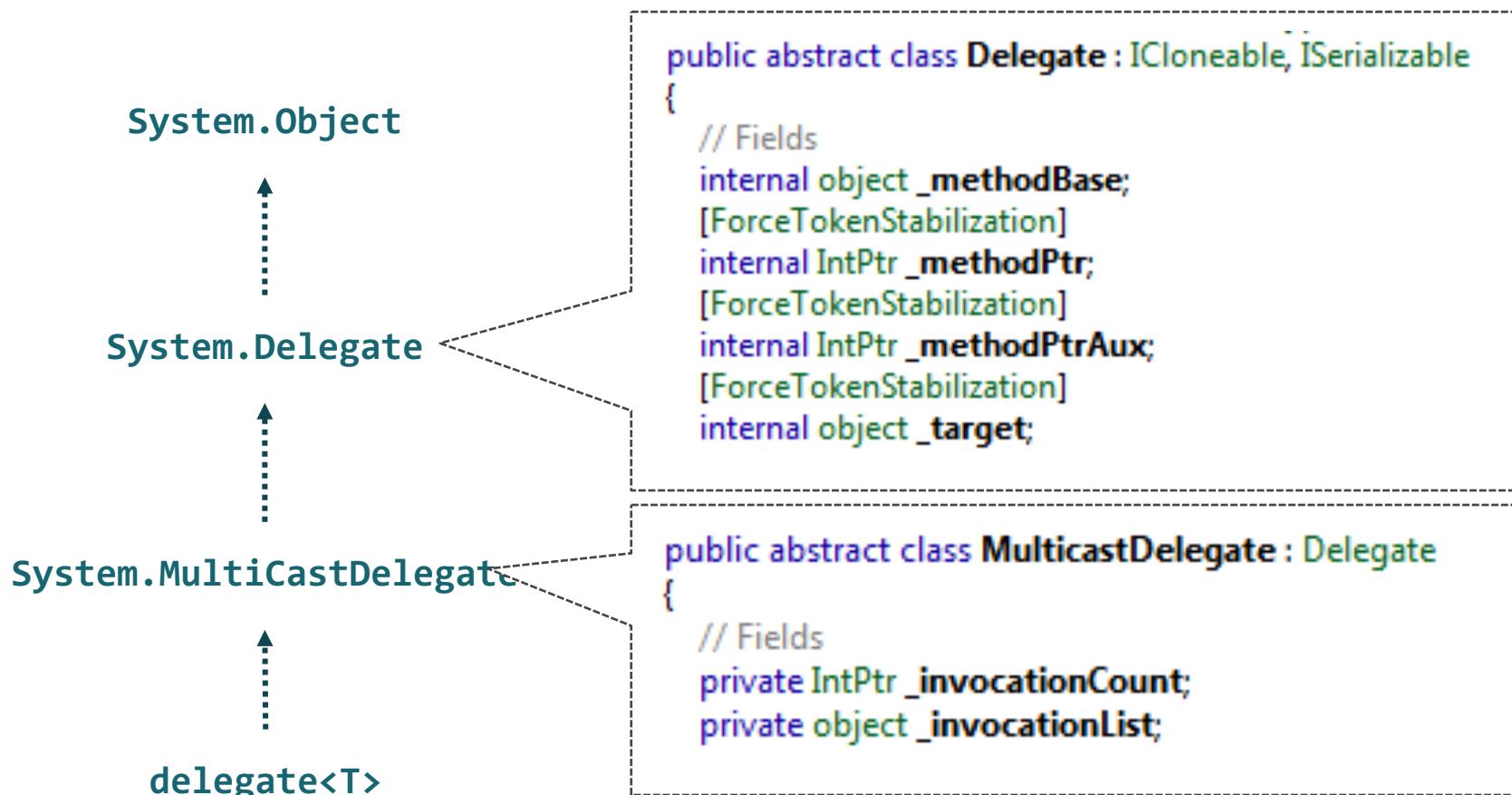
≡ BeginInvoke(Int32, AsyncCallback, Object) : IAsyncResult

≡ EndInvoke(IAsyncResult) : Int32

≡ Invoke(Int32) : Int32

Compiler C#

Delegates in Depth



Delegates in Depth

The `System.MulticastDelegate` class is derived from `System.Delegate`, which is itself derived from `System.Object`. The reason why there are two delegate classes is historical and unfortunate; there should be just one delegate class in the FCL. Sadly, you need to be aware of both of these classes because even though all delegate types you create have `MulticastDelegate` as a base class, you'll occasionally manipulate your delegate types by using methods defined by the `Delegate` class instead of the `MulticastDelegate` class. For example, the `Delegate` class has static methods called `Combine` And `Remove`. The signatures for both of these methods indicate that they take `Delegate` parameters. Because your delegate type is derived from `MulticastDelegate`, which is derived from `Delegate`, instances of your delegate type can be passed to these methods.

Delegates in Depth

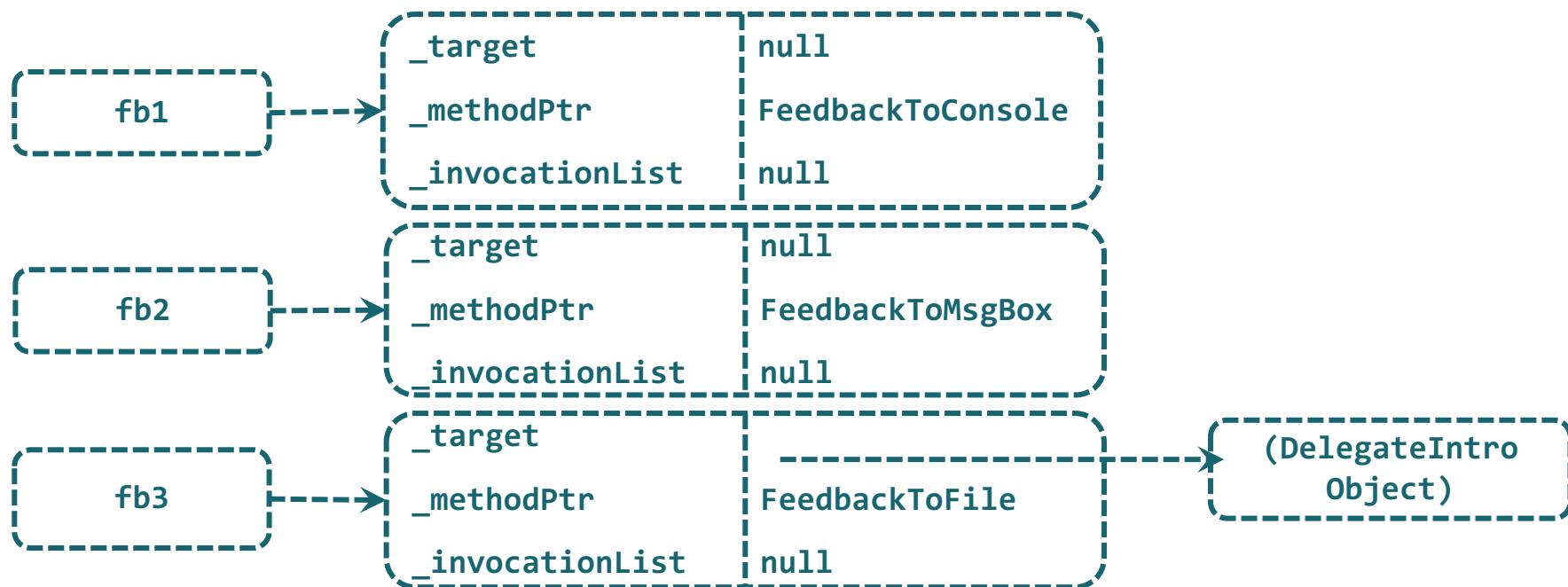
```
public abstract class Delegate : ICloneable, ISerializable
{
    // Methods for interacting with a list of functions.
    public static Delegate Combine(params Delegate[] delegates);
    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Remove(Delegate source, Delegate value);
    public static Delegate RemoveAll(Delegate source, Delegate value);
    // Overloaded operations.
    public static bool operator ==(Delegate d1, Delegate d2);
    public static bool operator !=(Delegate d1, Delegate d2);
    // Properties showing delegate purpose.
    public MethodInfo Method { get; }
    public object Target { get; }
}
```

Delegates in Depth

```
public abstract class MulticastDelegate : Delegate
{
    // Returns a list of methods that the delegate “points to”.
    public sealed override Delegate[] GetInvocationList ();
    // Overloaded operations
    public static bool operator ==(MulticastDelegate d1, MulticastDelegate d2);
    public static bool operator != (MulticastDelegate d1, MulticastDelegate d2);
    // Used internally to manage a list of methods,
    // supported by the delegate
    private IntPtr _invocationCount;
    private object _invocationList;
}
```

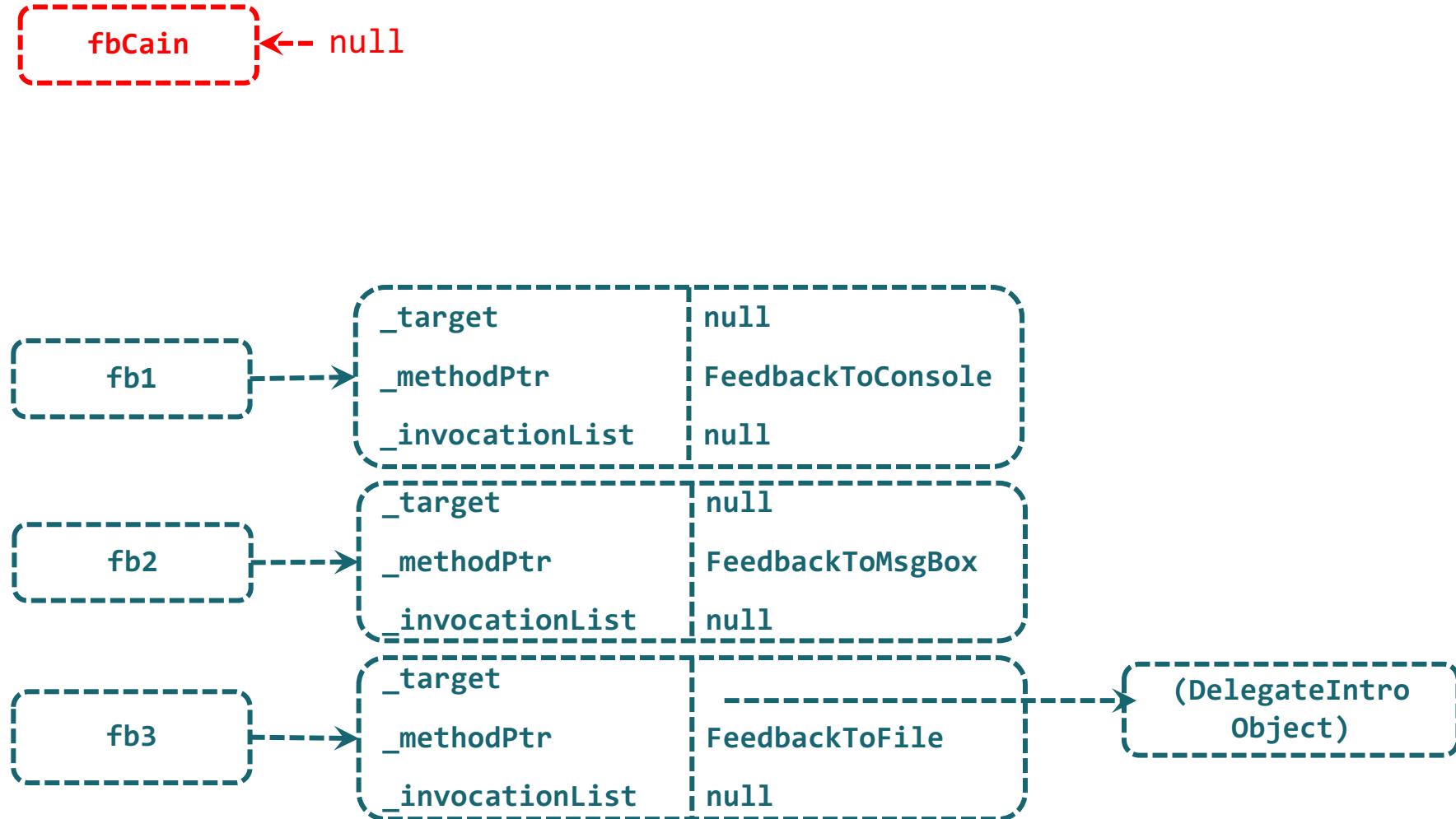
Delegates in Depth

```
public delegate void Feedback(int value);
. . .
DelegateIntro di = new ...;
. . .
Feedback fb1 = new Feedback(FeedbackToConsole);
Feedback fb2 = new Feedback(FeedbackToMsgBox);
Feedback fb3 = new Feedback(di.FeedbackToFile);
```



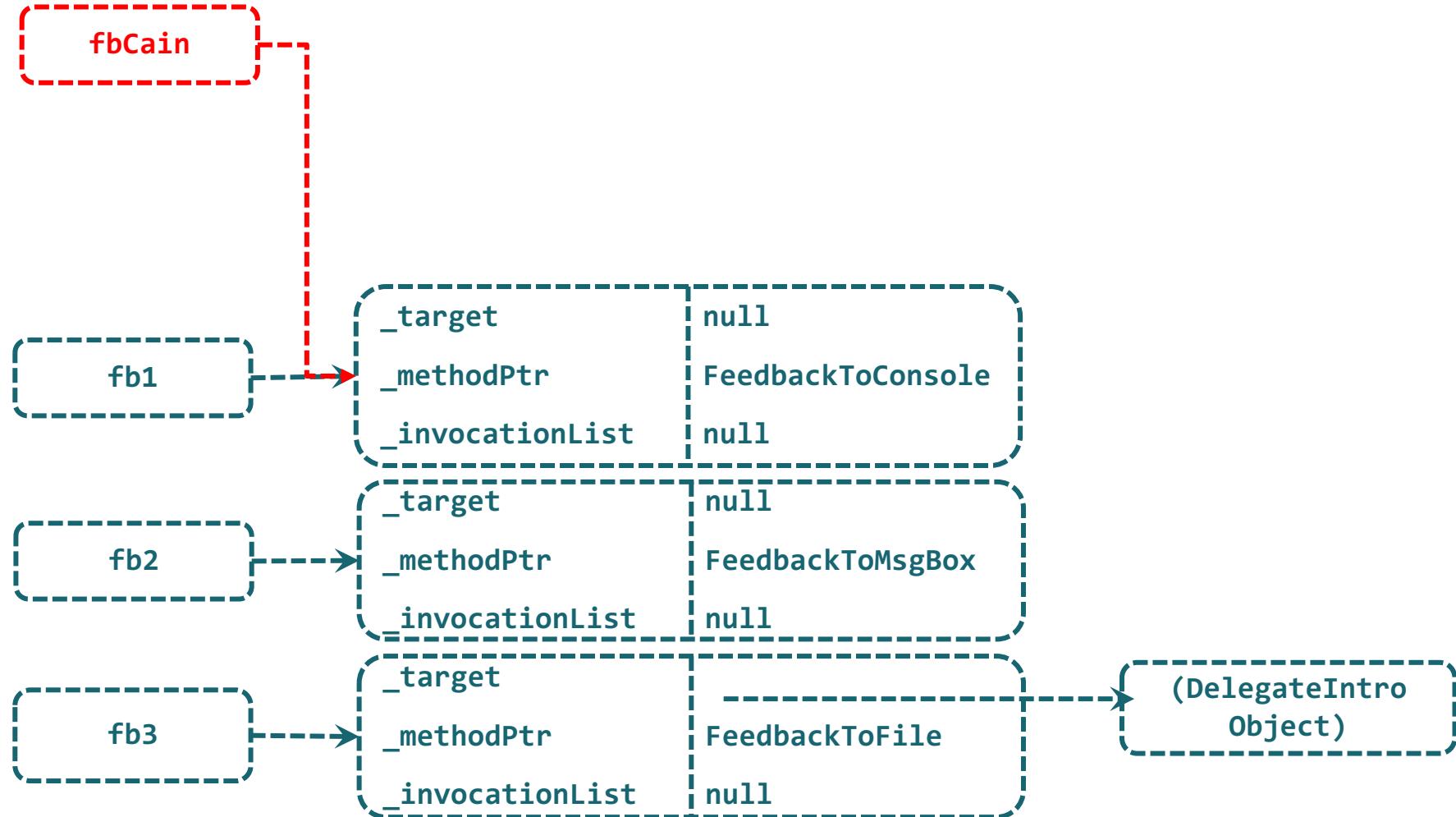
Delegates in Depth

```
fbChain = null;
```



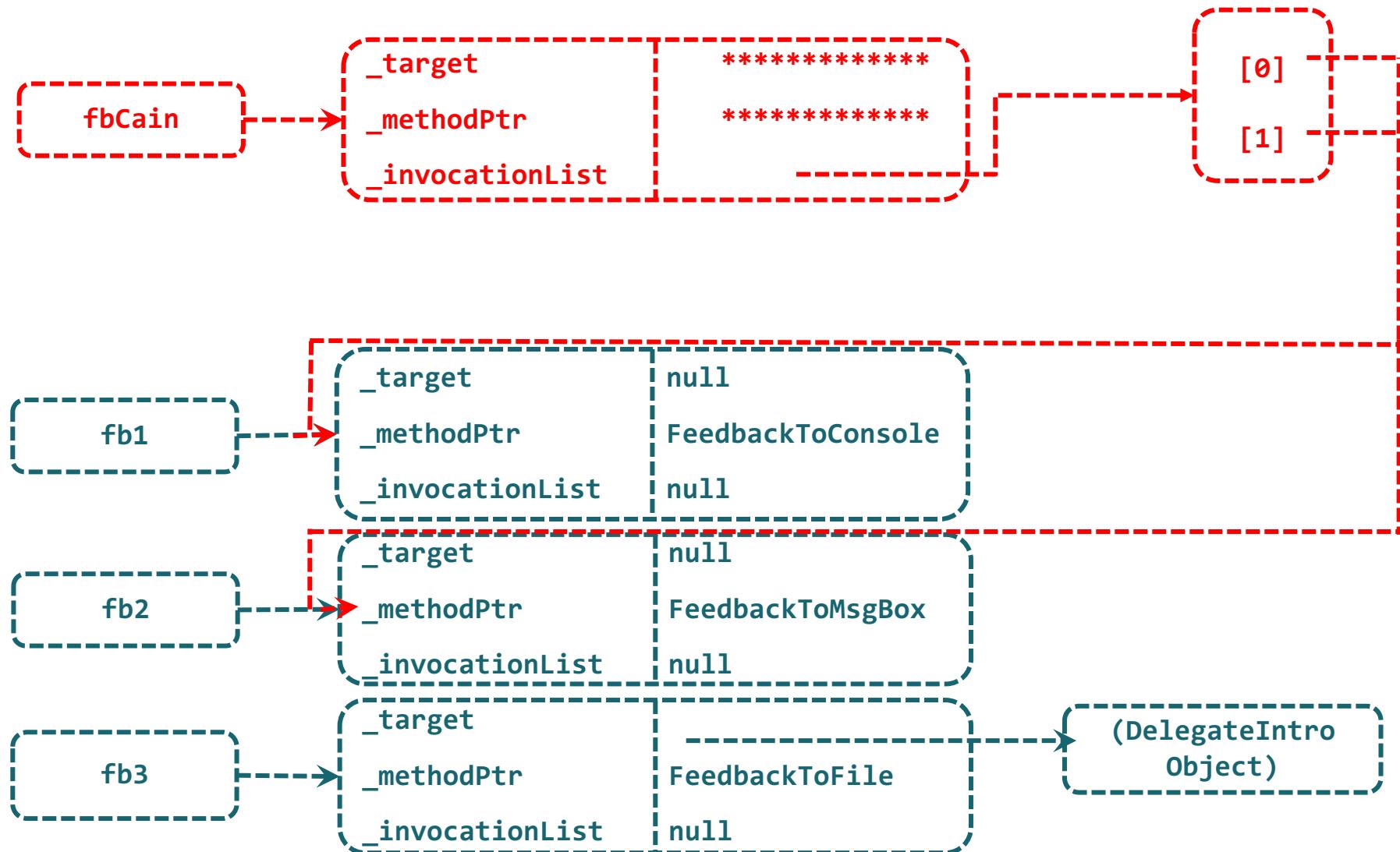
Delegates in Depth

```
fbChain = (Feedback)Delegate.Combine(fbChain, fb1);
```



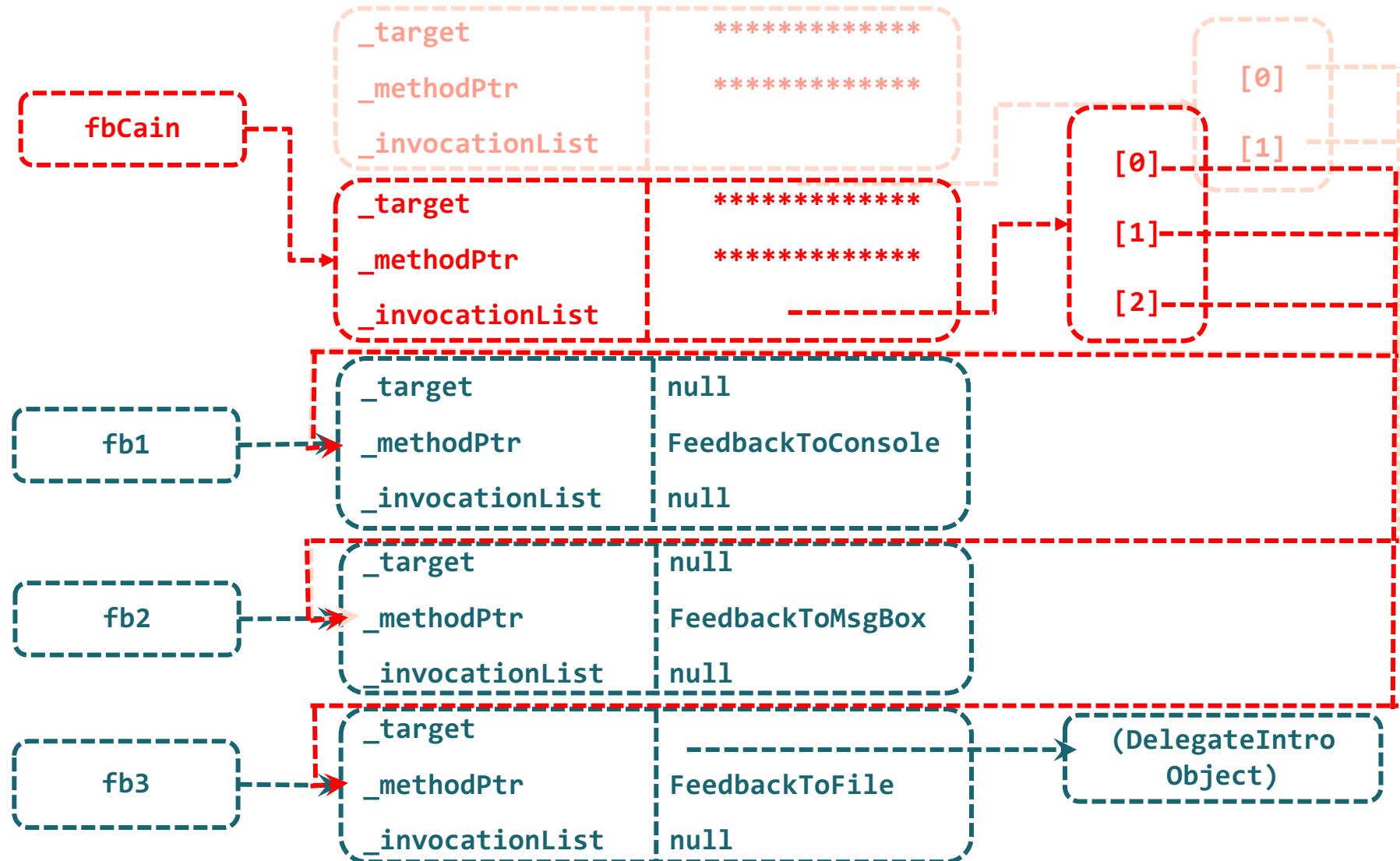
Delegates in Depth

```
fbChain = (Feedback)Delegate.Combine(fbChain, fb2);
```



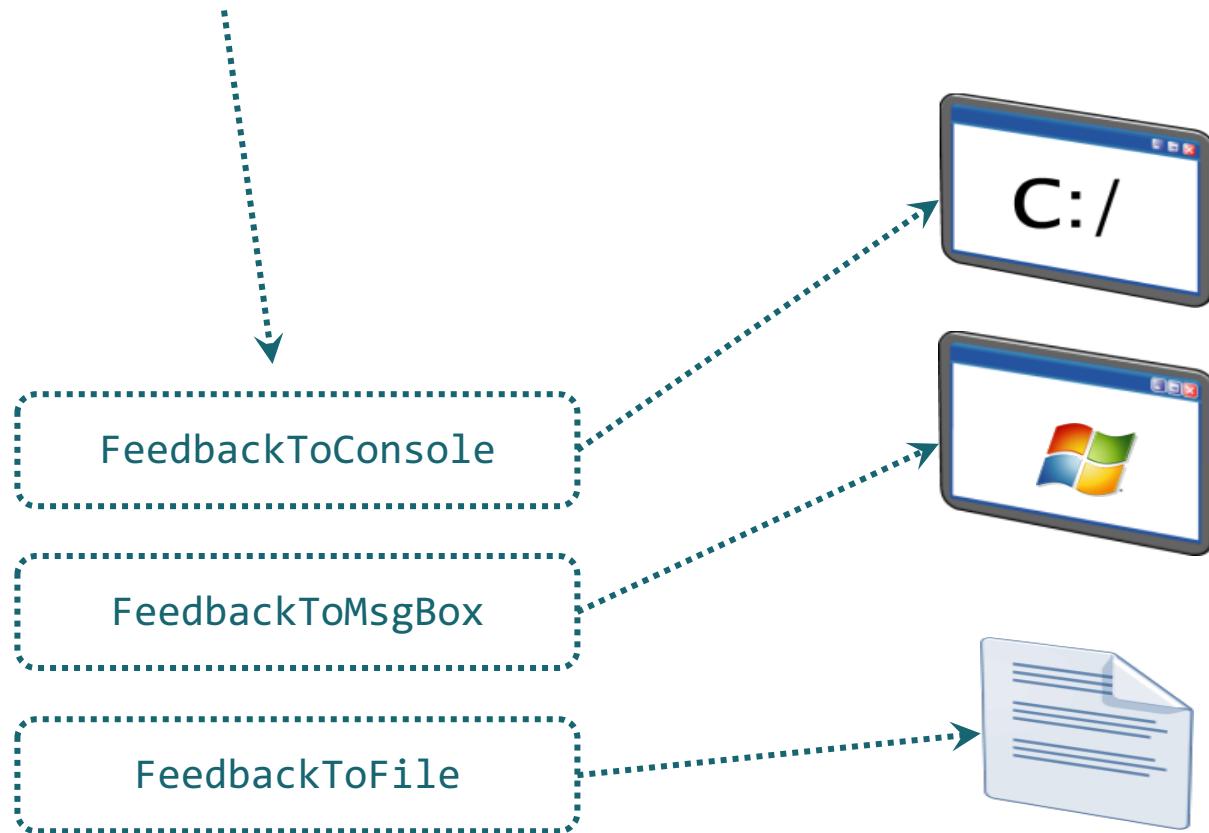
Delegates in Depth

```
fbChain = (Feedback)Delegate.Combine(fbChain, fb3);
```



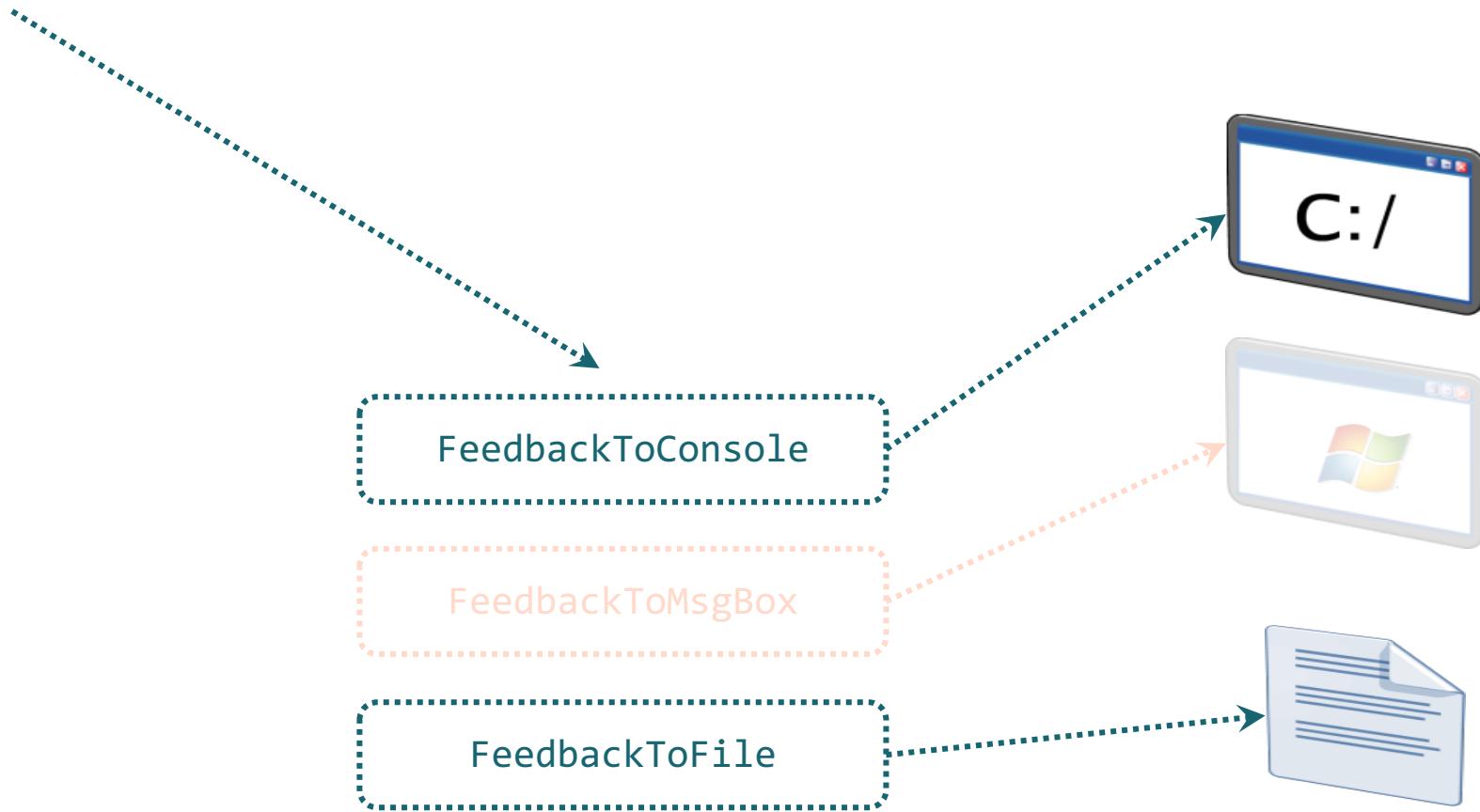
Multicast Delegates

```
int value = . . .;  
. . .  
if (fbChain != null) fbChain (value);
```



Multicast Delegates

```
fbChain = (Feedback)Delegate.Remove(fbChain, new Feedback(FeedbackToMsgBox));  
. . .  
fbChain (value);
```



Multicast Delegates

```
DelegateIntro di;  
. . .  
Feedback fb1 = FeedbackToConsole;  
Feedback fb2 = FeedbackToMsgBox; ↗  
Feedback fb3 = di.FeedbackToFile;  
. . .  
Feedback fbChain = null;  
  
fbChain += fb1; ← shorthand syntax  
fbChain += fb2;  
fbChain += fb3;  
. . .  
fbChain -= FeedbackToMsgBox;
```

Delegates are *immutable*, so when you call `+=` or `-=`, you're in fact creating a *new* delegate instance and assigning it to the existing variable

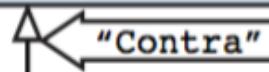
Multicast Delegates

Expression	Result
null + d1	d1
d1 + null	d1
d1 + d2	[d1, d2]
d1 + [d2, d3]	[d1, d2, d3]
[d1, d2] + [d2, d3]	[d1, d2, d2, d3]
[d1, d2] - d2	d1
[d1, d2] - d1	d2
[d1, d2, d1] - d1	[d1, d2]
[d1, d2, d3] - [d1, d2]	d3
[d1, d2, d3] - [d2, d1]	[d1, d2, d3]
[d1, d2, d3, d1, d2] - [d1, d2]	[d1, d2, d3]
[d1, d2] - [d1, d2]	null

Covariant and contravariant of delegates

Contravariant
Parameter Types

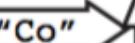
```
delegate void StringSender( string arg);
```



```
void Target ( object arg);
```

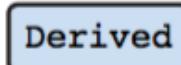
Covariant
Return Types

```
delegate object StringReceiver();
```

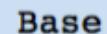


```
string Target ();
```

Legend



"Derived can be used where Base is expected"



Anonymous methods

Add the anonymous method as a handler for a delegate

Use the overloaded delegate keyword

Optionally specify the parameter names and types for the anonymous method

```
SomeDelegate del += new SomeDelegate (delegate (int parameter)
{
    // Perform operation.
    return 10;
});
```

Add the method body for the anonymous method

Do not specify a return type; the compiler will work out the return type based on the context

Anonymous methods

```
delegate void DelegateType();  
. . .  
DelegateType delegateInstance = delegate() { Console.WriteLine("Hello"); };  
delegateInstance += delegate() { Console.WriteLine("Bonjour"); };  
delegateInstance();
```



```
delegate void DelegateType(params int[] arr);  
. . .  
return delegate(params int[] arr){ Console.WriteLine("Hello");};
```



Capturing variables in anonymous methods

An *outer variable* is a local variable or parameter (excluding ref and out parameters) whose scope includes an anonymous method. The this reference also counts as an outer variable of any anonymous method within an instance member of a class.

A *captured outer variable* (usually shortened to *captured variable*) is an outer variable that's used within an anonymous method. To go back to closures, the function part is the anonymous method, and the environment it can interact with is the set of variables captured by it.

Capturing variables in anonymous methods

```
void EnclosingMethod()
{
    int outerVariable = 5;
    string capturedVariable = "captured";
    if (DateTime.Now.Hour == 23)
    {
        int normalLocalVariable = DateTime.Now.Minute;
        Console.WriteLine(normalLocalVariable);
    }

    MethodInvoker x = delegate()
    {
        string anonLocal = "local to anonymous method";
        Console.WriteLine(capturedVariable + anonLocal);
    };
    x();
}
```

The diagram illustrates the variable capture mechanism in the provided C# code. It uses numbered callouts to point from specific code elements to their corresponding variable types:

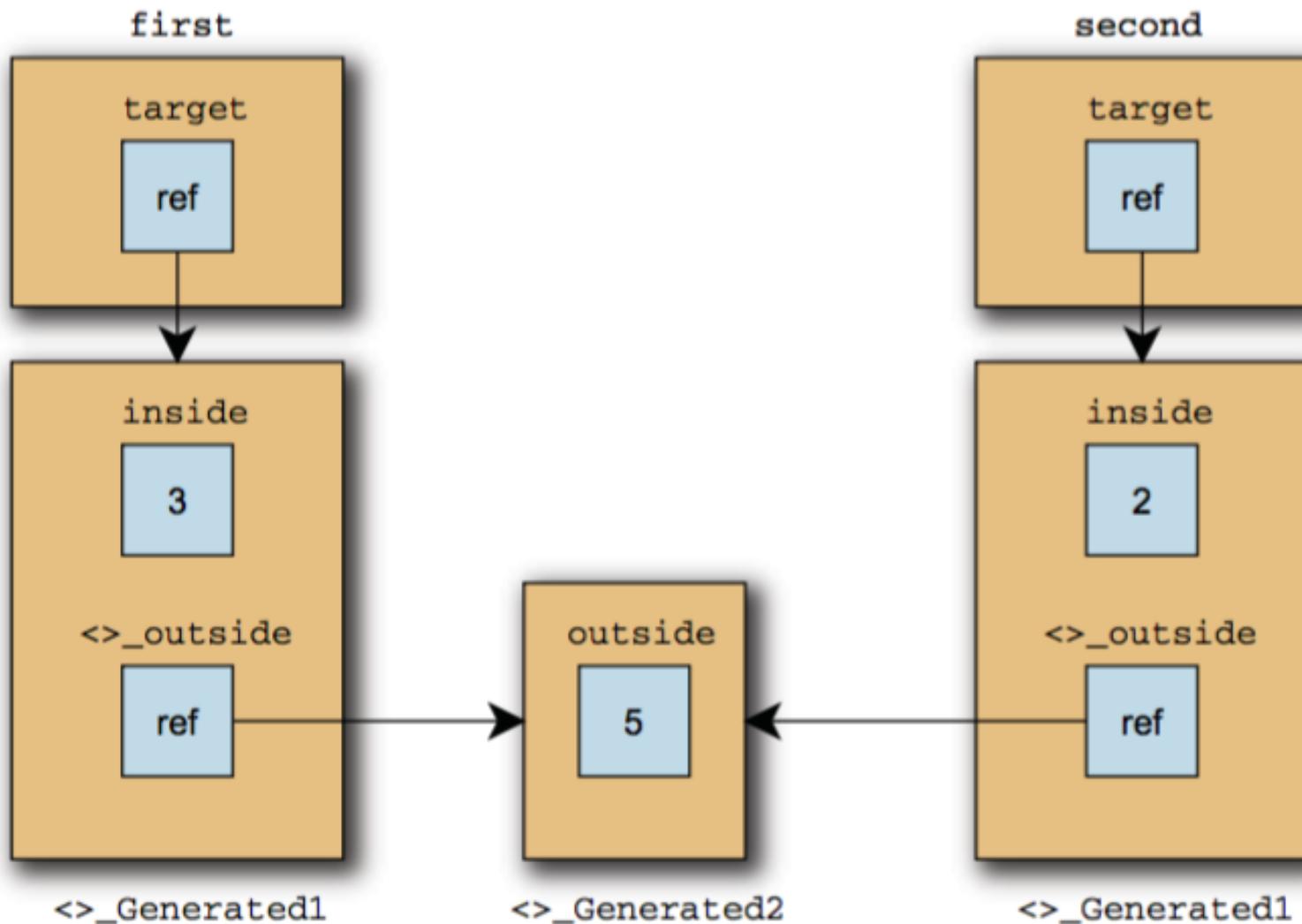
- 1 Outer variable (uncaptured)**: Points to the declaration of `outerVariable` (line 3).
- 2 Outer variable captured by anonymous method**: Points to the declaration of `anonLocal` (line 6), which captures the `capturedVariable` from the enclosing scope.
- 3 Local variable of normal method**: Points to the declaration of `normalLocalVariable` (line 4).
- 4 Local variable of anonymous method**: Points to the declaration of `anonLocal` (line 6).
- 5 Capture of outer variable**: Points to the assignment of `capturedVariable` to the parameter of the anonymous method (line 6).

Capturing variables in anonymous methods

```
MethodInvoker[] delegates = new MethodInvoker[2];  
  
int outside = 0; ① Instantiates variable once  
  
for (int i = 0; i < 2; i++)  
{  
    int inside = 0; ② Instantiates variable multiple times  
  
    delegates[i] = delegate  
    {  
        Console.WriteLine ("({0},{1})", outside, inside);  
        outside++;  
        inside++;  
    };  
}  
  
MethodInvoker first = delegates[0];  
MethodInvoker second = delegates[1];  
  
first();  
first();  
first();  
  
second();  
second();
```

③ Captures variables with anonymous method

Capturing variables in anonymous methods

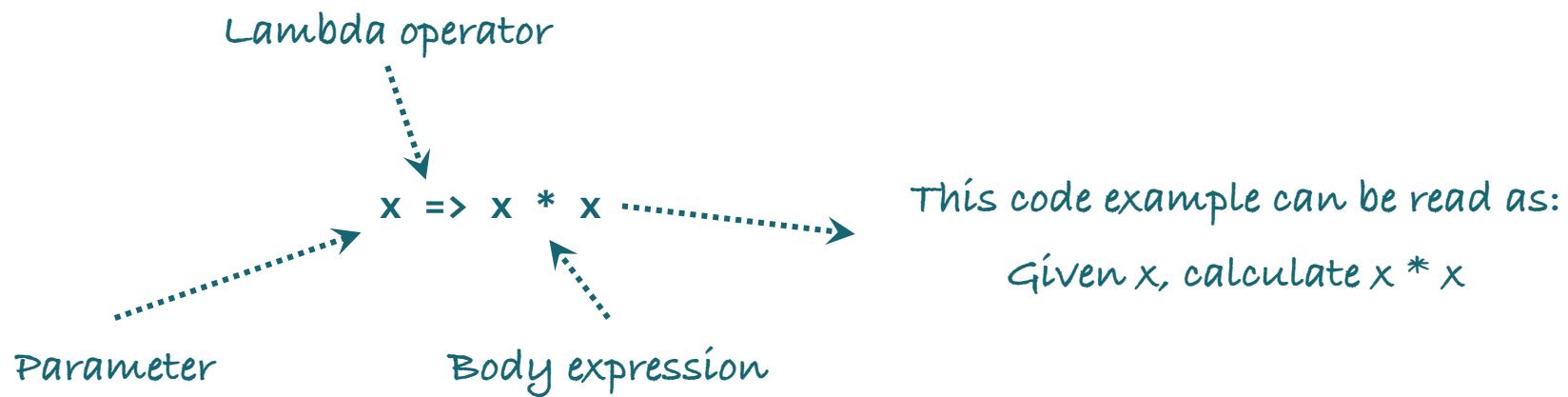


Lambda Expression

A lambda expression is an expression that returns a method

Lambda expressions are defined by using the **=> operator**

Lambda expressions use a natural and concise syntax



(parameters) => expression-or-statement-block

Lambda Expression

```
delegate(string text) { return text.Length; }
```

Start with anonymous method

```
(string text) => { return text.Length; }
```

Convert to lambda expression

```
(string text) => return text.Length
```

Single expression, no braces required

```
(text) => text.Length
```

Let the compiler infer the parameter type

```
text => text.Length
```

Remove unnecessary parentheses

Lambda Expression

Lambda Expression with Multiple parameters

```
(s, youngAge) => s.Age >= youngage;
```

```
(Student s, int youngAge) => s.Age >= youngage;
```

Lambda expression without any parameter:

```
() => Console.WriteLine("Parameter less lambda expression")
```

Multiple statements in body expression

```
(s, youngAge) =>
{
    Console.WriteLine("Lambda expression with multiple statements in the
body");
    Return s.Age >= youngAge;
}
```

Variable Scope in Lambda Expressions

Lambda expressions can use any variables that are in scope when they are declared

Lambda expressions can define variables

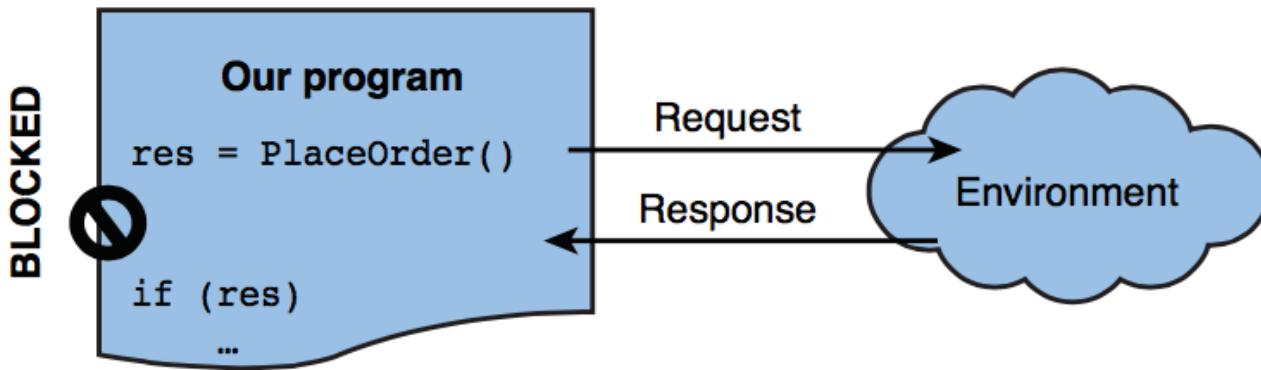
```
string name; // Class level field.  
...  
  
void SomeMethod() {  
    int count = 0;  
    del += new SomeDelegate(() =>  
    {  
        int temp = 10;  
        CheckName(name);  
        count++; // Do something more interesting.  
    });  
}
```

Using an outer variable in a lambda expression can extend its life cycle

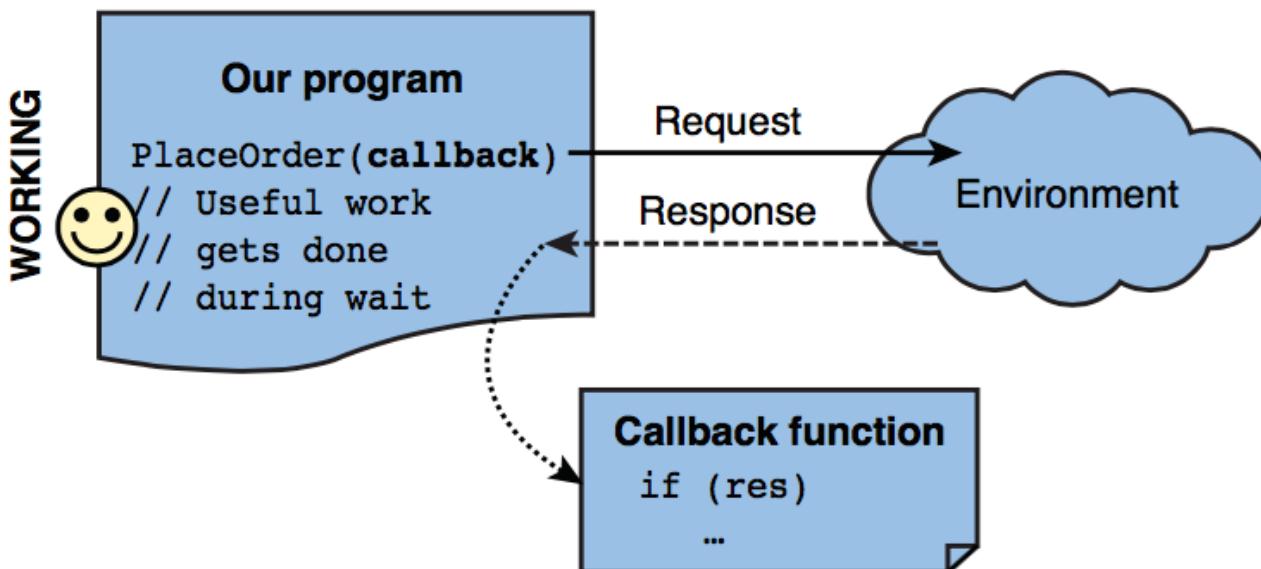
Use this feature carefully!

Interactive versus reactive programming

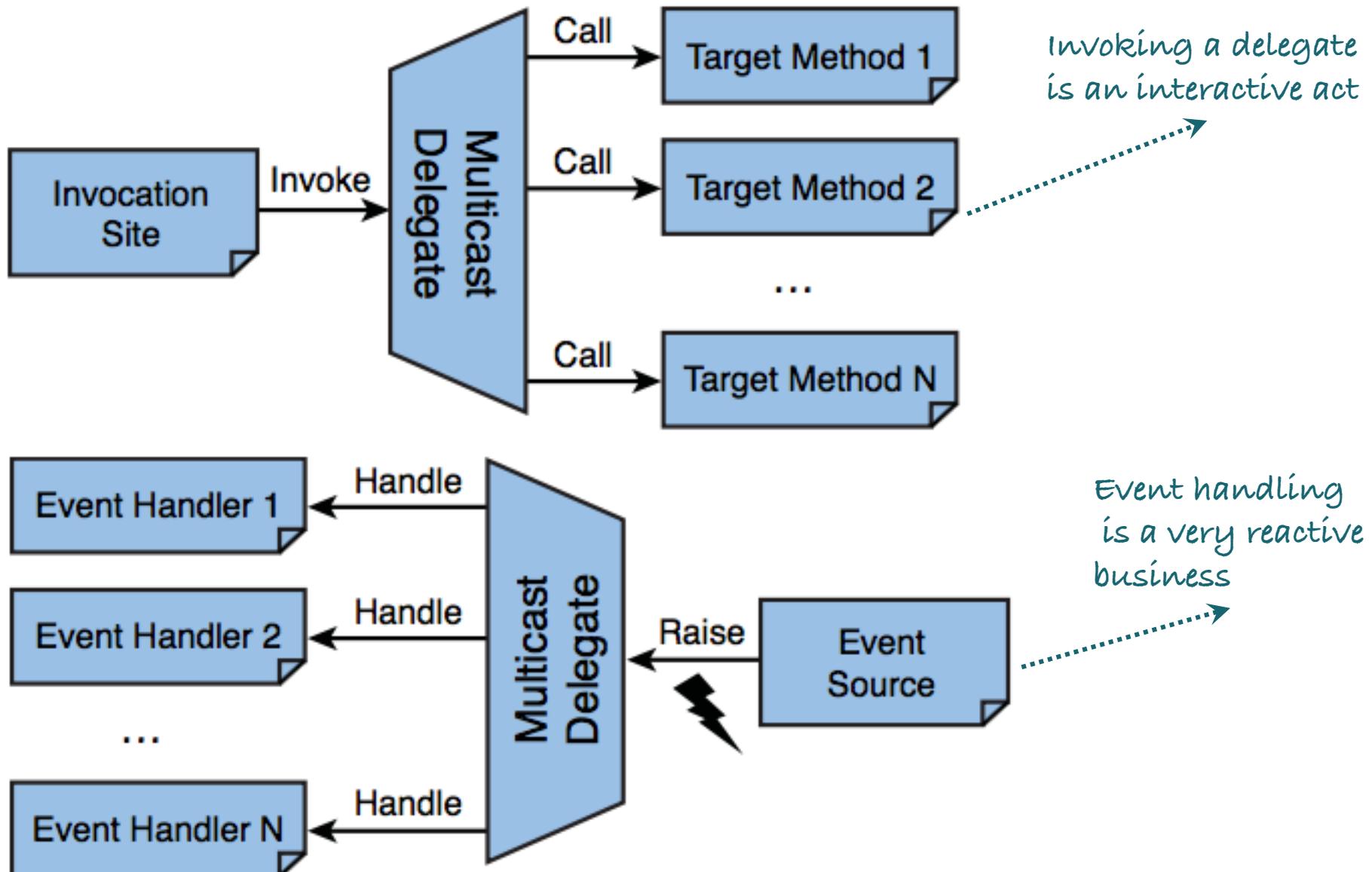
Interactive



Reactive



The Two Sides of Delegates



Events

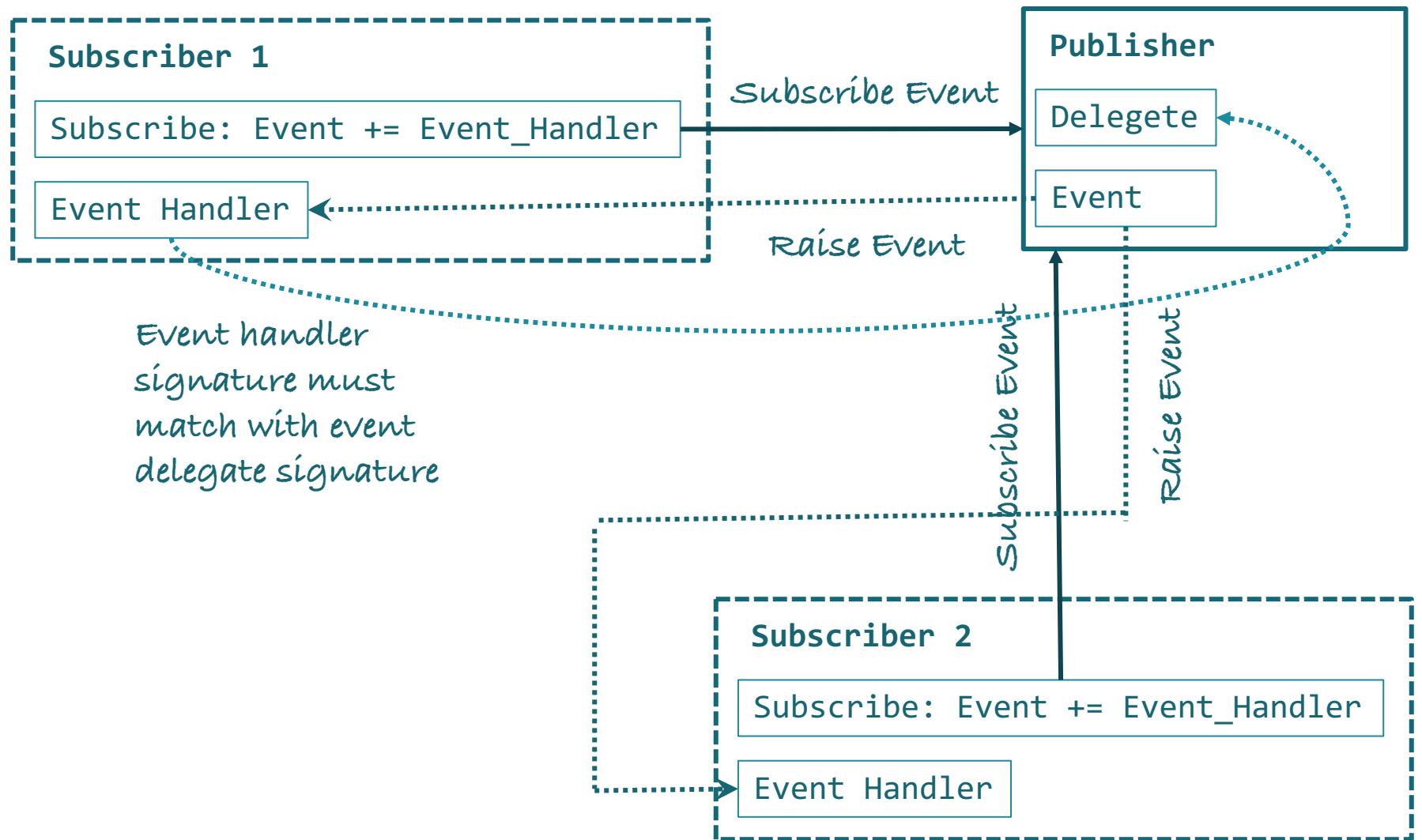
When using delegates, two emergent roles commonly appear: *broadcaster (publisher)* and *listener (subscriber)*.

The *broadcaster* is a type that contains a delegate field. The broadcaster decides when to broadcast by invoking the delegate.

The *subscribers* are the method target recipients. A subscriber decides when to start and stop listening by calling `+=` and `-=` on the broadcaster's delegate. A subscriber does not know about, or interfere with, other subscribers.

Events are a language feature that formalizes this pattern. An event is a construct that exposes just the subset of delegate features required for the broadcaster/subscriber model. The main purpose of events is to *prevent subscribers from interfering with one another*.

Events

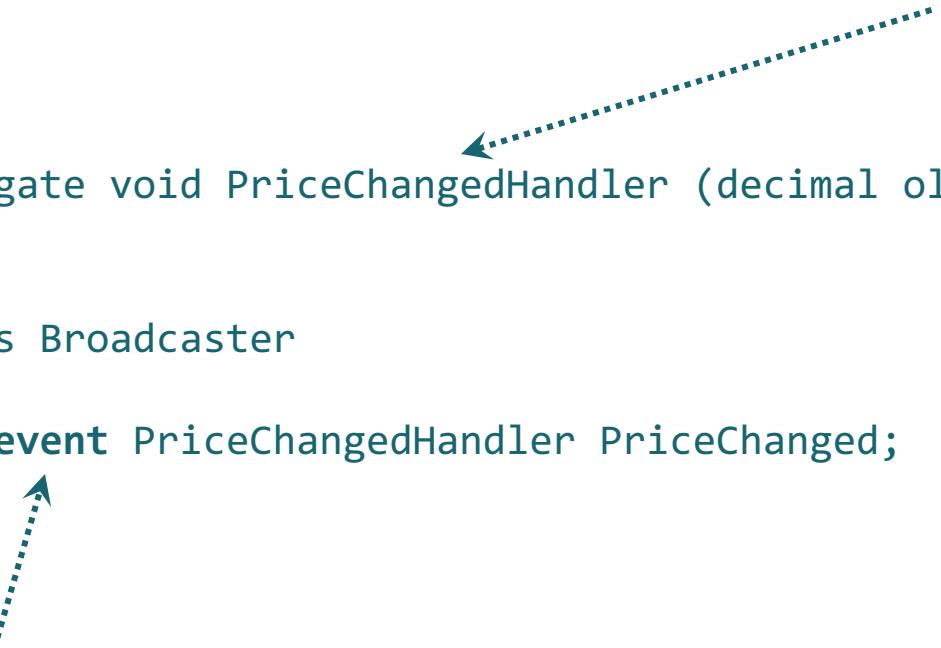


Events

```
public delegate void PriceChangedHandler (decimal oldPrice, decimal newPrice);  
  
public class Broadcaster  
{  
    public event PriceChangedHandler PriceChanged;  
}
```

Event declaration

Delegate declaration



The easiest way to declare an event is to put the event keyword in front of a delegate member

Events in depth

```
public delegate void PriceChangedHandler (decimal oldPrice, decimal newPrice);

public class Broadcaster
{
    public event PriceChangedHandler PriceChanged;
}

PriceChangedHandler priceChanged
```

Compiler C#

```
PriceChangedHandler PriceChanged
{
    add
    {
        priceChanged += value;
    }
    remove
    {
        priceChanged -= value;
    }
}
```

A private delegate field

A public pair of event accessor functions (`add_PriceChanged` and `remove_PriceChanged`), whose implementations forward the `+=` and `-=` operations to the private delegate field

Events in depth

```
public delegate void PriceChangedHandler (decimal oldPrice, decimal newPrice);
```

```
public class Broadcaster  
{  
    public event PriceChangedHandler PriceChanged;  
}
```

Compiler C#

```
Stock  
  + Base Types  
  + Derived Types  
  .ctor(String)  
  OnPriceChanged(PriceChangedEventArgs) : Void  
  Price : Decimal  
  PriceChanged  
    add_PriceChanged(EventHandler<PriceChangedEventArgs>) : Void  
    remove_PriceChanged(EventHandler<PriceChangedEventArgs>) : Void  
  price : Decimal  
  PriceChanged : EventHandler<PriceChangedEventArgs>  
  symbol : String
```

The event accessors

The private delegate field

Standard Event Pattern. First Step

EventArgs subclass is named according to the information it contains (rather than the event for which it will be used)

```
public class PriceChangedEventArgs : System.EventArgs  
{  
    public readonly decimal LastPrice;  
    public readonly decimal NewPrice;  
  
    public PriceChangedEventArgs (decimal lastPrice, decimal newPrice)  
    {  
        LastPrice = lastPrice;  
        NewPrice = newPrice;  
    }  
}
```

EventArgs is a base class for conveying information for an event

It typically exposes data as properties or as read-only fields

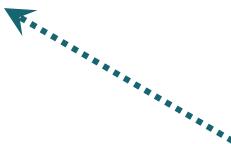
Standard Event Pattern. Second Step

There are three rules to define a delegate for the event:

- It must have a void return type.
- It must accept two arguments: the first of type object, and the second a subclass of EventArgs. The first argument indicates the event broadcaster, and the second argument contains the extra information to convey.
- Its name must end with *EventHandler*.

```
public delegate void EventHandler<TEventArgs> (object source, TEventArgs e)  
    where TEventArgs : EventArgs;
```

```
public event EventHandler<PriceChangedEventArgs> PriceChanged;
```



The generic EventHandler
delegate has been used here

Standard Event Pattern. Last Step

```
public class Stock
{
    ...
    public event EventHandler<PriceChangedEventArgs> PriceChanged;
    protected virtual void OnPriceChanged (PriceChangedEventArgs e)
    {
        if (PriceChanged != null) PriceChanged (this, e);
    }
}
```

The name must match the name of the event, prefixed with the word `On`, and then accept a single `EventArgs` argument

```
var temp = PriceChanged;
if (temp != null) temp (this, e);
```

PriceChanged?.Invoke (this, e);

The pattern requires that you write a protected virtual method that raise the event

In multithreaded scenarios

C# 6.0 – now the best general way to invoke events

Event Modifiers

```
public class Foo
{
    public static event EventHandler<EventArgs> StaticEvent;
    public virtual event EventHandler<EventArgs> VirtualEvent;
}
```

Events can be static

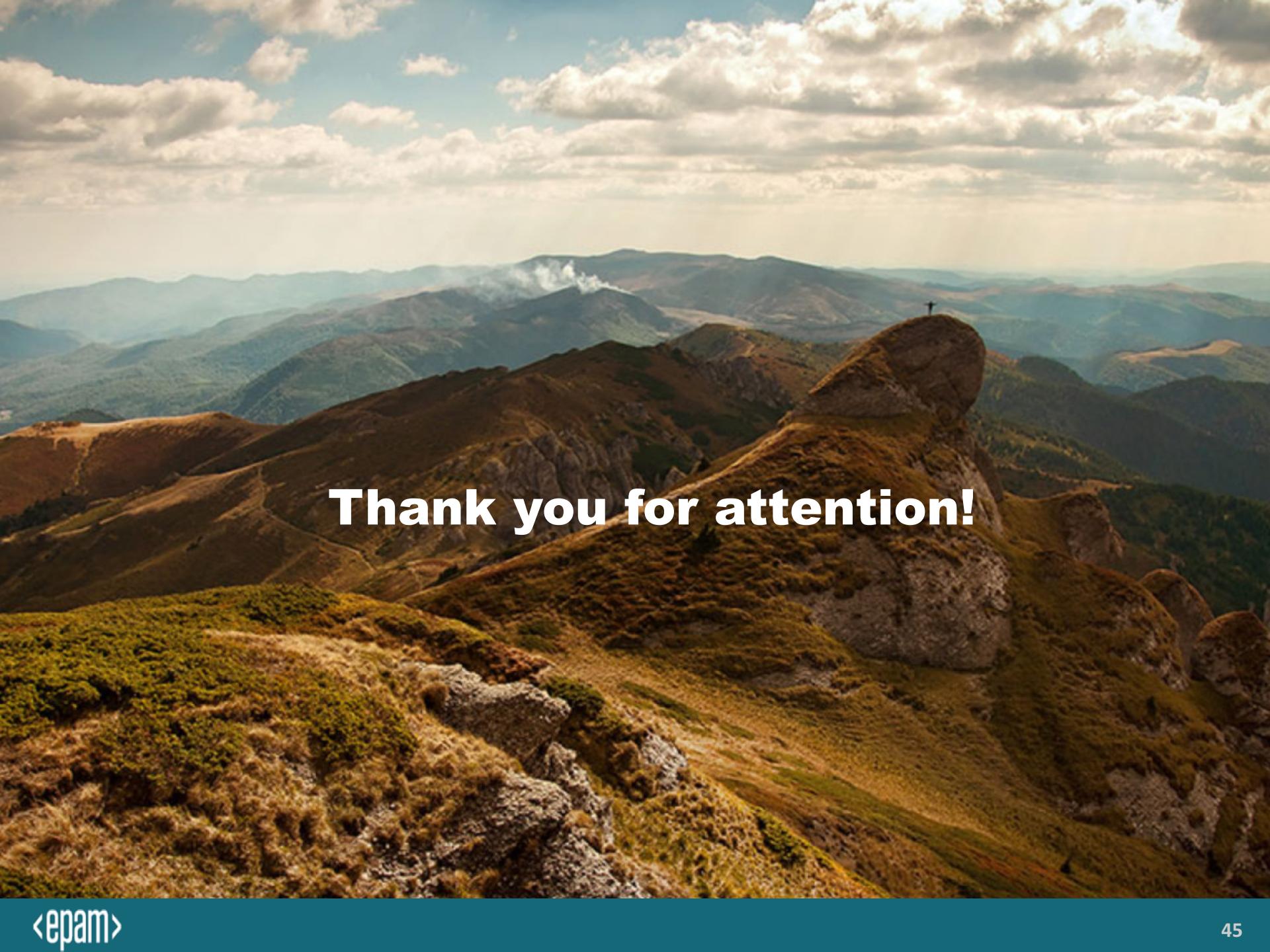
Like methods, events can be virtual,
overridden, abstract, or sealed

Best Practices for Using Events

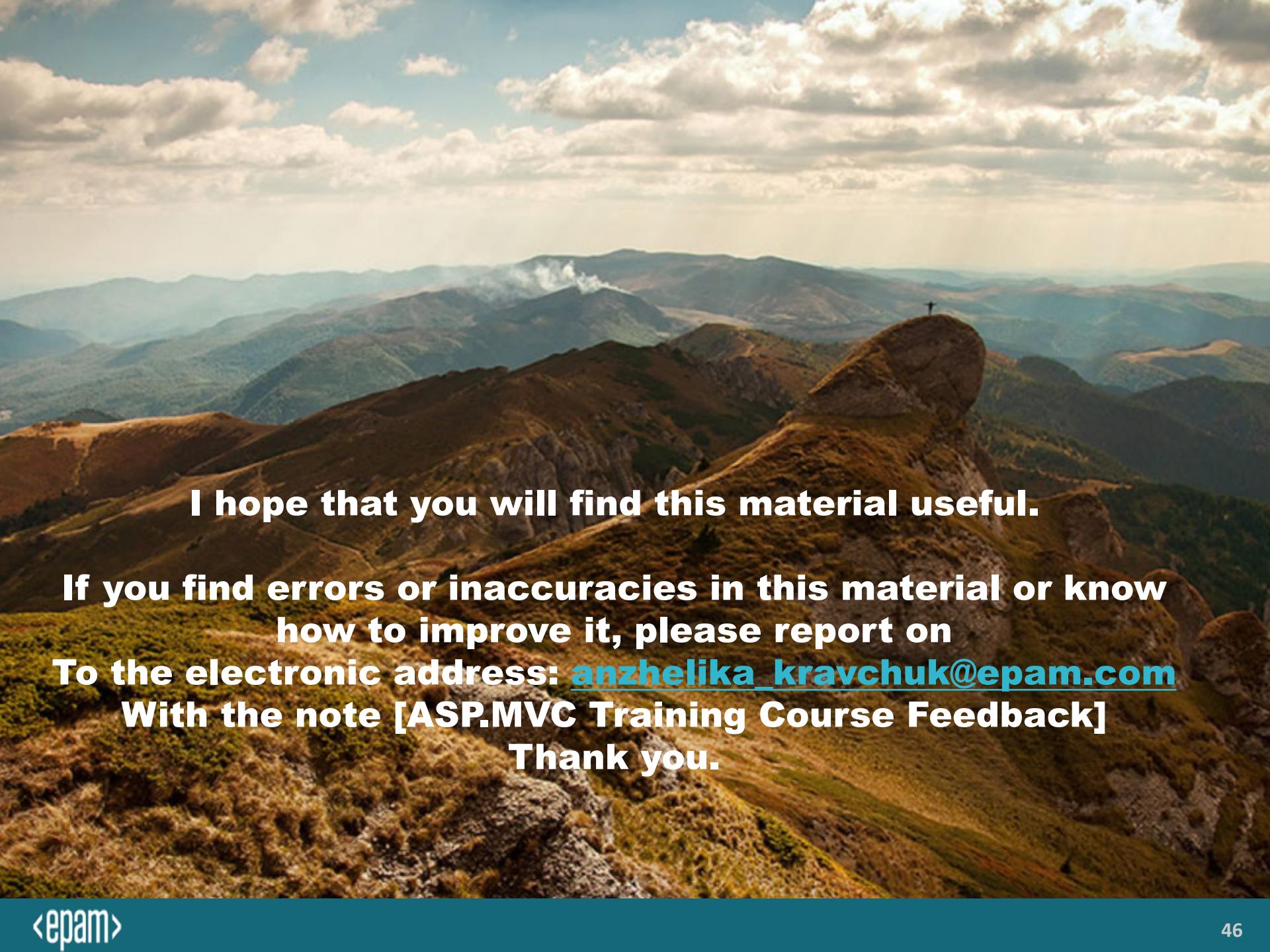
Use the standard event signature

Use a protected virtual method to raise an event

Do not pass null to an event

A wide-angle photograph of a mountain range under a dramatic sky. In the foreground, rocky terrain and green slopes are visible. In the middle ground, a person stands on a prominent, rounded mountain peak. The background features multiple layers of mountains, with a large plume of white smoke or steam rising from the middle layer. The sky is filled with scattered, bright clouds.

Thank you for attention!

A wide-angle photograph of a mountainous landscape. In the foreground, there are rocky, grassy slopes. In the middle ground, several mountain ridges are visible, with one prominent peak on the right side where a small figure of a person stands. The background shows more distant mountain ranges under a sky filled with scattered, fluffy clouds.

I hope that you will find this material useful.

**If you find errors or inaccuracies in this material or know
how to improve it, please report on
To the electronic address: anzhelika_kravchuk@epam.com
With the note [ASP.MVC Training Course Feedback]
Thank you.**