

CHAPTER 9

C#

DELEGATES, LAMBDAS AND EVENTS

Contents

1. Delegates
2. Lambda
3. Events

Delegates

- A **delegate** is a type that **represents a method** with a specific signature and return type.
  - The declaration of a delegate looks **exactly like the declaration of a method**, except with the **keyword delegate** in front of it.
- Just like classes and interfaces, we can declare delegates **outside of classes or nested within classes**. We can mark them private, public, or internal.

- **Important Points About Delegates:**
  - Provides a good way to **encapsulate the methods**.
  - Delegates are the library class in System namespace.
  - These are the type-safe pointer of any method.
  - Delegates are mainly used in implementing the **call-back methods** and **events**.
  - Delegates can be chained together as two or more methods can be called on a single event.
  - Delegates can also be used in "**anonymous methods**" invocation.

- The method signature must match the signature of ***delegate*** *Type*
  - same number of parameters
  - same parameter types (including the return type)
  - same parameter attributes (ref, out)
- There are **three steps** in using delegates. These include:
  1. Declaration
  2. Instantiation, and
  3. Invocation

Example: Declaration

1. **Declaration**
   - Syntax

     [modifier] ***delegate*** *result-type identifier ([parameters])*

   - Example: A delegate representing a method that adds two numbers and returns a result:
     public delegate int AddNumbers(int value1, int value2);

2. Instantiation

- You should always instantiate a delegate before you can use it.
- The statement given next shows how you can instantiate the delegate declared above.

```
AddNumbers  an= new
AddNumbers(findSum);
```

3. Invocation

- Once you have declared and instantiated the delegate, you can invoke the method that the delegate points to easily.

  an(9,8);

- You can also invoke the method that the delegate instance points to using the **Invoke()** method on the delegate instance as shown below.

  an. Invoke(9,8);

Example: 1


```
delegate double MathCalculation(float value1, float value2);

public static class Calculator
{
    public static double AddNumbers(float value1, float value2)
    {
        return value2 + value2;
    }
    public static double DivideNumbers(float value1, float value2)
    {
        return value1 / value2;
    }
}
```

```
MathCalculation add = Calculator.AddNumbers;
MathCalculation divide = Calculator.DivideNumbers;

var result = add.Invoke(2, 3);
var result = divide(100, 3);
```

Example: 2


```
public delegate void myDelegate(string value);

static void Main(string[] args)
{
    // myDelegate delegate points to Console.WriteLine
    myDelegate print = Console.WriteLine;

    print("Hello Ethiopia!");

}
```

Exercise 1:

- Write a delegate for built-in method Console.WriteLine that can print all type of objects?
  - Sample output
    - print(“Hello Ethiopia!”);
    - print(123);
    - print(56.78);

Solution


```
public delegate void myDelegate<T>(T value);

static void Main(string[] args)
{

    // myDelegate delegate points to Console.WriteLine
    myDelegate<object> print = Console.WriteLine;

    print("Hello Ethiopia!");
    print(123);
    print(56.78);
}
```

`Action<>` and `Func<>`

- **Action** and **Func** are **delegates** that we can use instead of defining our own delegate types.
  - That's important to remember: Action and Func *are* delegates.
  - For example, instead of declaring the MathCalculation delegate and assigning the following,

MathCalculation add = Calculator.AddNumbers;

- We could assign:

Func<float, float, double> add = Calculator.AddNumbers;

Func<>

- We use **Func<>** to represent a method that returns something.
  - If the function has parameters, the first generic argument(s) represent those parameters.
  - The last generic argument indicates the return type.
  - Example
    - Func<int, DateTime, string> is a function with an int and DateTime parameter that returns a string.

Action<>

- **Action<>** represent methods that return nothing.
  - **Action<string, int>** represents a method with a string and int parameter.
  - Example:

```
Action<string> myAction = Console.WriteLine;
myAction("Hello Ethiopia");
```

- Should I Use Action or Func or Declare a delegate?
  - Action or Func or declaring a delegate are, essentially, **interchangeable**.
  - Declaring a delegate allows us to **give it a name** that indicates what it's for.
    - A delegate called MathCalculation is clearly intended to do math. Func<float, float, double> doesn't tell us what the method does.
  - It's a preference and a case-by-base decision. Delegate may be clearer, but Action and Func save us from having to declare more delegate types and usually suffice when the use is obvious.

Multicasting of a Delegate

- Multicasting of delegate is an extension of the normal delegate(sometimes termed as Single Cast Delegate).
- It helps the user to **point more than one method in a single call.**
- Properties
  - Delegates are combined and when you call a delegate then a complete list of methods is called.
  - All methods are called in **First in First Out(FIFO)** order.
    - '+' or '+=' Operator is used to **add the methods** to delegates.
    - '–' or '-=' Operator is used to **remove the methods** from the delegates list.

- **Note:** Remember, multicasting of delegate should have a **return type of Void** otherwise it will throw a runtime exception.
  - Also, the multicasting of delegate will return the value only from the last method added in the multicast. Although, the other methods will be executed successfully.

Lambda

- A **lambda expression** is an **anonymous function** and it is mostly used to create delegates in **LINQ**.
  - Simply put, it's a method without a declaration, i.e., access modifier, return value declaration, and name.
- It's a shorthand that allows you to write a method in the same place you are going to use it.
  - Especially useful in places where a **method is being used only once**, and the method definition is short.
  - It **saves you the effort of declaring and writi**ng a separate method to the containing class.
  - Usually lambda expressions are used as **predicates** or instead of delegates (a type that references a method instance), which can be applied on collections, processing their elements and/or returning a certain result.

- The '=>' is the lambda operator which is used in all lambda expressions.
  - The Lambda expression is divided into two parts, the left side is the input and the right is the expression.
- The Lambda Expressions can be of two types:
  1. **Expression Lambda:** Consists of the input and the expression.
     - *Syntax:* input => expression;
  2. **Statement Lambda:** Consists of the input and a set of statements to be executed.
     - *Syntax:* input => { statements };

Example


```
List<int> list = new List<int>() { 1, 2, 3, 4, 5, 6 };
List<int> evenNumbers = list.FindAll(x => (x % 2) == 0);
foreach (var num in evenNumbers)
{
    Console.Write("Sum: ", num);
}
```

- The example above loops through the whole collection of numbers and for each element (named x) a check, if the number is multiple of 2, is made (through the Boolean expression (x % 2) == 0).

- Sorting with Lambda Expressions
  - If we want to sort the elements in a certain collection, we can use the extension methods OrderBy(…) and OrderByDescending(…), by defining the way of sorting in a lambda function.
  - Example: Student collection

```
var sortedStuds = studs.OrderByDescending(x => x.Age);
```

- Statements in Lambda Expressions Lambda functions can also have a body. So far we have used lambda functions with only one statement.
  - Now we will pay more attention to lambda functions that have a body. Let’s return to the example with the even numbers.
  - Suppose we want to print to the console the values of all numbers, to which our lambda function is applied to and to return the result if they are even or not. We can do it the following way:

Example


```
List<int> list = new List<int>() { 20, 1, 4, 8, 9, 44 };
// Process each argument with code statements
var evenNumbers = list.FindAll((i) =>
{
    Console.WriteLine("Value of i is: {0}", i);
    return (i % 2) == 0;
});
```

Exercise 2:

- Write a program that prints to the console all numbers from a given list, that are multiples of 7 and 3 at the same time. Use the built-in extension methods with lambda expressions.
- Given List

List<int> number = new List<int>() { 7, 3, 12, 4, 21, 40, 51, 93};

Event

- An **event** is something special that is going to happen.
  - An event has a **publisher**, **subscriber**, **notification** and a **handler**.
  - Generally, UI controls use events extensively. For example, the button control in a Windows form has multiple events such as click, mouseover, etc.
- A custom class can also have an event to notify other subscriber classes about something that has happened or is going to happen.

- An event is nothing but an **encapsulated delegate**.
  - As we have learned in the previous section, a delegate is a reference type data type.
- Now, to declare an event, use the **event** keyword before declaring a variable of delegate type, as below:

  *public **event** someEvent someEvent;*
  - Thus, a delegate becomes an event using the **event** keyword.

Example

- Consider the following **PrintHelper** class that prints integers in different formats like money and decimal. It includes a **beforePrintEvent** to notify the subscriber of the **BeforePrint** event before it going to print the number.

```
public class PrintHelper
{
      // declare delegate
      public delegate void BeforePrint();
      //declare event of type delegate
      public event BeforePrint beforePrintEvent;
      public PrintHelper() {}
      public void PrintDecimal(int dec)
      {
          if (beforePrintEvent != null)
              beforePrintEvent();
          Console.WriteLine("Decimal: {0:G}", dec);
      }
      public void PrintMoney(int money)
      {
          if (beforePrintEvent != null)
              beforePrintEvent();
          Console.WriteLine("Money: {0:C}", money);
      }
}
```

- The delegate can also be invoked using the Invoke() method, e.g., beforePrintEvent.Invoke().
- PrintHelper is a **publisher** class that publishes the **beforePrint** event.
- Notice that in each print method, it first checks to see if beforePrintEvent is not null and then it calls *beforePrintEvent()*.
  - beforePrintEvent is an object of type BeforePrint delegate, so it would be **null if no class is subscribed to the event** and that is why it is necessary to check for null before calling a delegate.

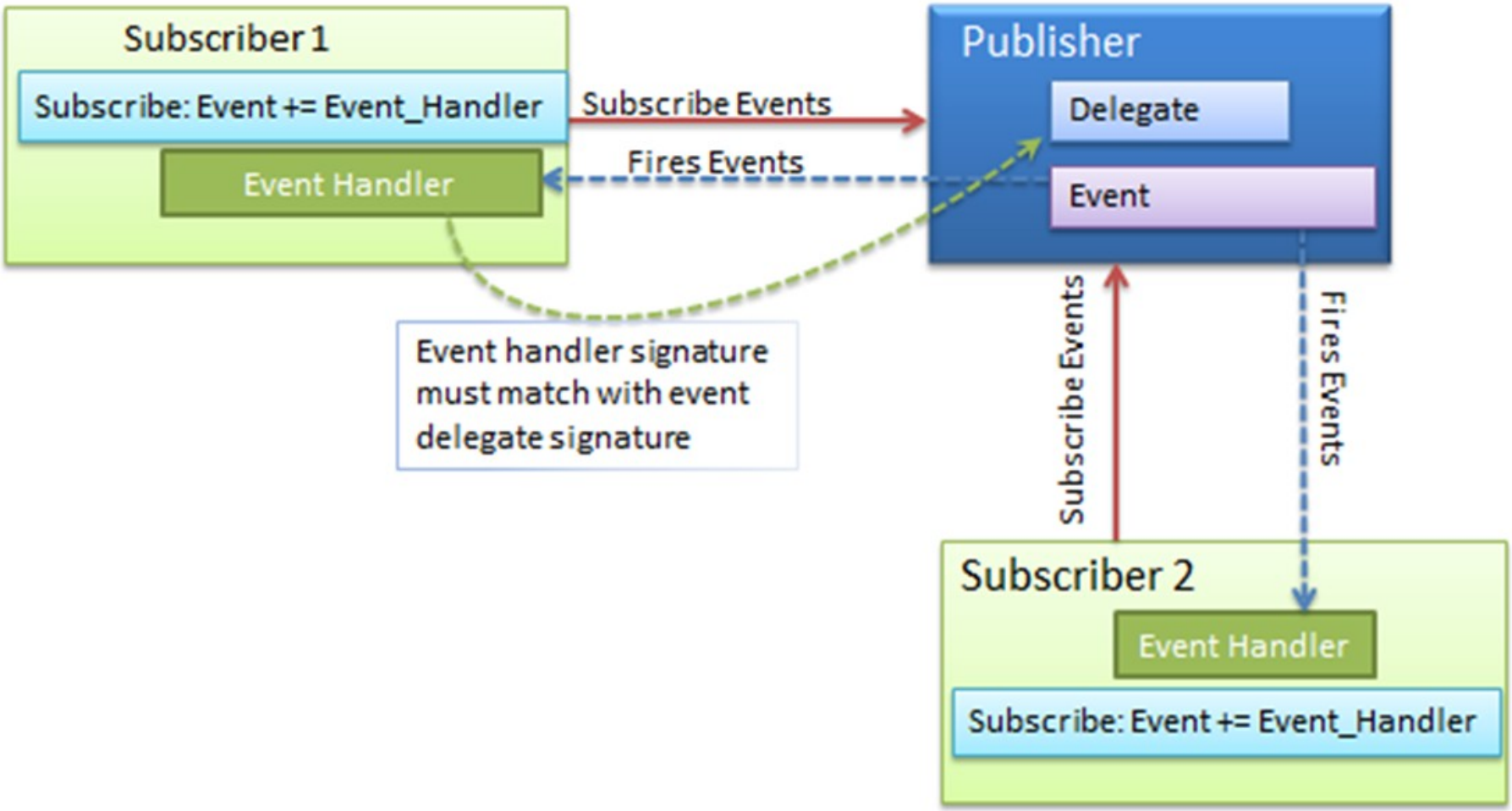
```
class Number
{
    private int myData;
    public int Value {
        get { return myData; }
        set { myData = value; }
    }
    private PrintHelper printer1;
    public Number(int val)
    {
        myData = val;
        printer1 = new PrintHelper();
        //subscribe to beforePrintEvent event
        printer1.beforePrintEvent += beforePrintEvent_handler;
    }
    //beforePrintevent handler
    void beforePrintEvent_handler()
    {
        Console.WriteLine("BeforPrintEventHandler: PrintHelper is going to print a value");
    }
    public void PrintMoney()
    {
        printer1.PrintMoney(myData);
    }
    public void PrintDecimal()
    {
        printer1.PrintDecimal(myData);
    }
}
```

- All the subscribers must provided a handler function, which is going to be called when a publisher raises an event.
- In the above example, the Number class creates an instance of PrintHelper and subscribes to the beforePrintEvent with the "+=" operator and gives the name of the function which will handle the event (it will be get called when publish fires an event).
  - **beforePrintEvent_handler** is the event handler that has the same signature as the BeforePrint delegate in the PrintHelper class.

```
class Program
{
    static void Main(string[] args)
    {
        Number myNumber = new Number(100000);
        myNumber.PrintMoney();
        myNumber.PrintDecimal();
        Console.ReadKey();

    }
}
```

```
BeforPrintEventHandler: PrintHelper is going to print a value
Money: $100,000.00
BeforPrintEventHandler: PrintHelper is going to print a value
Decimal: 100000
```


Subscriber 1
Subscribe: Event += Event_Handler
Event Handler
Subscribe Events
Fires Events
Publisher
Delegate
Event
Event handler signature must match with event delegate signature
Subscribe Events
Fires Events
Subscriber 2
Event Handler
Subscribe: Event += Event_Handler

Event Arguments

- Events can also pass data as an argument to their subscribed handler. An event passes arguments to the handler as per the delegate signature.
  - **Example**: PrintHelper declares the BeforePrint delegate that accepts a string argument. So now, you can pass a string

```
public delegate void BeforePrint(string message);
public event BeforePrint beforePrintEvent;
public void PrintMoney(int num)
{
    if (beforePrintEvent != null)
        beforePrintEvent.Invoke("PrintMoney");
```

- Important points about Events
  - Use event keyword with delegate type to declare an event.
  - Check event is null or not before raising an event.
  - Subscribe to events using "+=" operator. Unsubscribe it using "-=" operator.
  - Function that handles the event is called event handler. Event handler must have same signature as declared by event delegate.
  - Events can have arguments which will be passed to handler function.
  - Events can also be declared static, virtual, sealed and abstract.
  - An Interface can include event as a member.

- Events will not be raised if there is no subscriber
- Event handlers are invoked synchronously if there are multiple subscribers
- The .NET framework uses an EventHandler delegate and an EventArgs base class.

Questions?

Thank You