# Diving into OOP (Day 10): Delegates in C# (A Practical Approach)
# - Akhil Mittal (www.codeteddy.com)

## Table of Contents:

## Introduction:

This article of the series "Diving into OOP" will explain all about delegates in C#. The article focusses more on practical implementations and less on theory. The article explains the concept in-depth.

## Delegates (The definition):

Let's start with the definition taken from MSDN

"A **delegate** declaration defines a reference type that can be used to encapsulate a method with a specific signature. A delegate instance encapsulates a static or an instance method. Delegates are roughly similar to function pointers in C++; however, delegates are type-safe and secure."

## Other Articles of the series:

Following is the list of all the other articles of the OOP series.

1. **Diving in OOP (Day 1): Polymorphism and Inheritance(Early Binding/Compile Time Polymorphism)**
2. Diving in OOP (Day 2): Polymorphism and Inheritance (Inheritance)
3. Diving in OOP (Day 3): Polymorphism and Inheritance (Dynamic Binding/Run Time Polymorphism)
4. Diving in OOP (Day 4): Polymorphism and Inheritance (All about Abstarct classes in C#)
5. Diving in OOP (Day 5): All about access modifiers in C# (Public/Private/Protected/Internal/Sealed/Constants/Readonly Fields)
6. Diving in OOP (Day 6): Understanding Enum in C# (A Practical Approach)
7. Diving into OOP (Day 7): Properties in C# (A Practical Approach)
8. Diving into OOP (Day 8): Indexers in C# (A Practical Approach)
9. Diving into OOP (Day 9): Understanding Events in C# (An Insight)
10. Diving into OOP (Day 10): Delegates in C# (A Practical Approach)

## Delegates

A delegate is one of the most interesting features of C# programming language. And it can directly be placed under namespace as classes are placed. Delegates completely follow the rule of object oriented programming. Delegates extend System.Delegate class.

Delegates are the best suit for anonymous invocations as they can call any method provided the signature of the method i.e. the return type and the parameters is same. Let's try to cover the topic via practical examples.

## Lab 1:

Create a console application and name it as per your choice, I named it EventsAndDelegates. Add a public class named DelegateExercises with the following implementation.

**DelegateExercises**

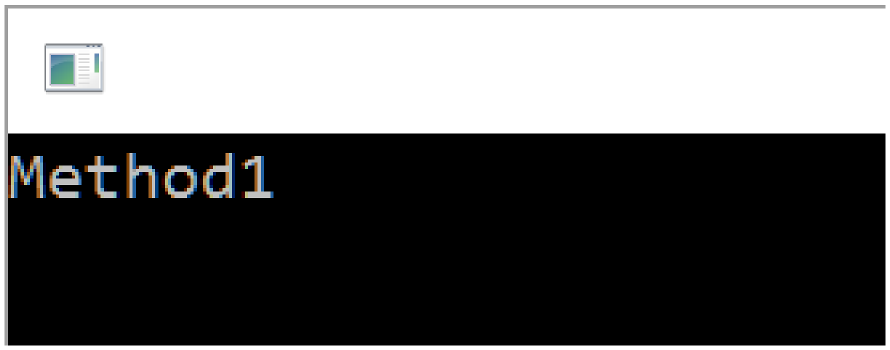
```
using System;

namespace DelegatesAndEvents
{
    public class DelegateExercises
    {
        public delegate void MyDelegate();
        void Method1()
        {
            Console.WriteLine("Method1");
            Console.ReadLine();
        }
        public void Method2()
        {
            MyDelegate myDelegate = new MyDelegate(Method1);
            myDelegate();
        }

    }
}
```

Call the method Method2 from Program.cs class.

**Program**

```
namespace DelegatesAndEvents
{
    class Program
    {
        static void Main(string[] args)
        {
            DelegateExercises delegateExercises=new DelegateExercises();
            delegateExercises.Method2();
        }
    }
}
```

**Output**



```
Method1
```

In the program class,  delegateExercises is an instance of DelegateExercises class and delegateExercises.Method2() method is invoked. While creating the instance we follow the rule of creating instance using new keyword, in the similar way we can also use new with delegate name as shown in Method2 of class delegateExercises. A delegate is very similar to properties or indexers in C#, i.e. it is a first class member of the class. It seems to be a function but is defined with a keyword named delegate. In the above example of Method2, we created the instance of delegate and passed the whole function i.e. Method1 as a parameter. That means a method itself can also be passed as a parameter using delegates. This is the way in which C# normally handles call back methods or event handlers. To instantiate a delegate, the traditional "new" keyword is used with one parameter i.e. the methods name itself "Method1". Method1 is the member of class "DelegateExercises" with void return type and taking no parameters. The new keyword as usual creates an object of type delegate, and as we can notice a method
"Method1" is invoked in a nontraditional way i.e. without using Method1() syntax. "myDelegate" is the delegate object of method "Method1" because the method is passed as a parameter when the object was created. Therefore when "myDelegate" is called, it means "Method1" is called, thus providing a level of abstraction.

## Lab 2:

**DelegateExercises**

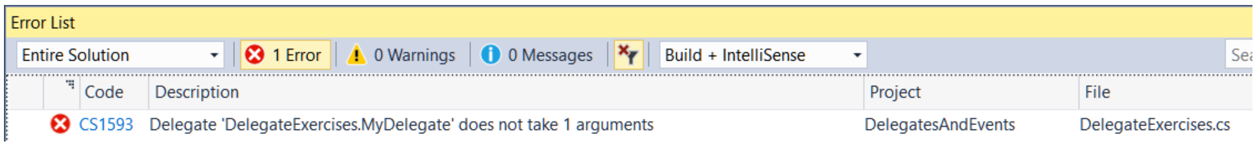
```
using System;

namespace DelegatesAndEvents
{
    public class DelegateExercises
    {
        public delegate void MyDelegate();
        void Method1()
        {
            Console.WriteLine("Method1");
            Console.ReadLine();
        }
        public void Method2()
        {
            MyDelegate myDelegate = new MyDelegate(Method1);
            myDelegate(50);
        }

    }
}
```

Call the method Method2 from Program.cs class.

**Program**

```
namespace DelegatesAndEvents
{
    class Program
    {
        static void Main(string[] args)
        {
            DelegateExercises delegateExercises=new DelegateExercises();
            delegateExercises.Method2();
        }
    }
}
```

**Output**



```
Code    Description                                                  Project             File
CS1593  Delegate 'DelegateExercises.MyDelegate' does not take 1 arguments  DelegatesAndEvents  DelegateExercises.cs
```

We get an error “Delegate 'DelegateExercises.MyDelegate' does not take 1 arguments” when we try to call delegate with a parameter. We passed 50 as a parameter for the method Method1, but you can clearly see that Method1 does not takes an integer parameter, therefore an error is shown at compilation.

## Lab 3:

**DelegateExercises**

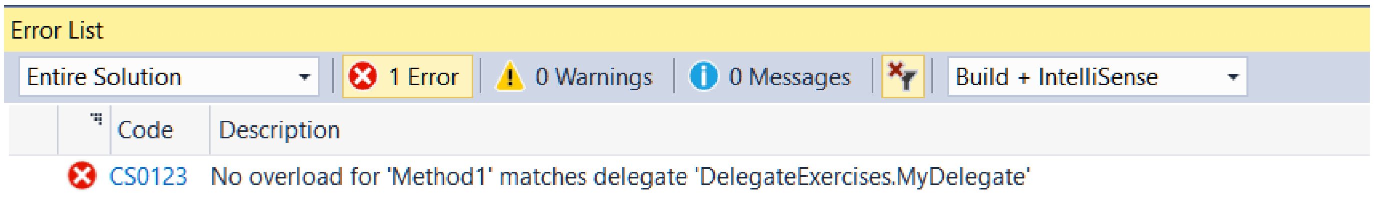
```
using System;
```

```
namespace DelegatesAndEvents
{
    public class DelegateExercises
    {
        public delegate void MyDelegate();
        void Method1(int i)
        {
            Console.WriteLine("Method1");
            Console.ReadLine();
        }
        public void Method2()
        {
            MyDelegate myDelegate = new MyDelegate(Method1);
            myDelegate();
        }

    }
}
```

**Program**

```
namespace DelegatesAndEvents
{
    class Program
    {
        static void Main(string[] args)
        {
            DelegateExercises delegateExercises=new DelegateExercises();
            delegateExercises.Method2();
        }
    }
}
```

**Output**

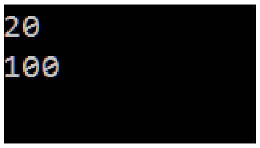

```
Error List
Entire Solution    1 Error    0 Warnings    0 Messages    Build + IntelliSense
    Code     Description
    CS0123   No overload for 'Method1' matches delegate 'DelegateExercises.MyDelegate'
```

We again got an error message, but this is different, we assumed that adding an integer parameter will solve the earlier error, but this error says that the delegate signature should match the method's signature.

## Lab 4:

```
using System;

namespace DelegatesAndEvents
{

  public class DelegateExercises
  {
    public delegate int MyDelegate(int intValue);

    public int Method1(int intMethod1)
    {
      return intMethod1*2;
    }

    public int Method2(int intMethod2)
    {
      return intMethod2*10;
    }

    public void Method3()
    {
      MyDelegate myDelegate = new MyDelegate(Method1);
      int result1 = myDelegate(10);
      System.Console.WriteLine(result1);
      myDelegate = new MyDelegate(Method2);
      int result2 = myDelegate(10);
      System.Console.WriteLine(result2);
    }
  }

  public class Program
  {
    public static void Main()
    {
      DelegateExercises delegateExercises = new DelegateExercises();
      delegateExercises.Method3();
      Console.ReadLine();

    }
  }
}
```

**Output:**



```
20
100
```

To get rid of the error, an integer parameter is added to the delegate MyDelegate. This means the method that will call the delegate will automatically handle this parameter. So this delegate now returns an integer instead of a void . Finally, when the method is run through delegate, compiler checks for the return patrameters too. In Method3, we now make the implementation more tricky, and created one more delegate type having another method name as an argument, and we again execute the delegate with the same parameter having integer value : 10, but the methot to be called changes each time, therefore the code could be written more dynamically ☺.

## Lab 5:

```
namespace DelegatesAndEvents
{
  public class Program
  {
    public static void Main()
    {
      DelegateExercises delegateExercises = new DelegateExercises();
      delegateExercises.Method3();
    }
  }

  public class DelegateExercises
  {
    public delegate int MyDelegate();

    void Method1()
    {
      System.Console.WriteLine("MyDelegate");
    }

    public void Method3()
    {
      MyDelegate myDelegate = new MyDelegate(Method1);
      myDelegate();
    }
  }
}
```

**<u>Output</u>**

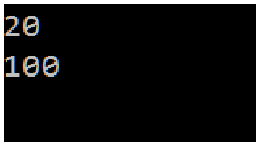
Error : 'void DelegateExercises.Method1()' has the wrong return type

Here compiler finds a mismatch, the return type of the delegate should ideally be integer if used in the context as mentioned in above code, but here Method1 returns void.

## Lab 6:

```
using System;

namespace DelegatesAndEvents
{
  public class Program
  {
    public static void Main()
    {
      DelegateExercises delegateExercises = new DelegateExercises();
      delegateExercises.Method3();
      Console.ReadLine();
    }
  }

  public class DelegateExercises
  {
    public delegate int MyDelegate(int intValue);

    int Method1(int intMethod1)
    {
      return intMethod1*2;
    }

    int Method2(int intMethod1)
    {
      return intMethod1*10;
    }

    public void Method4(MyDelegate myDelegate)
    {
      int result = myDelegate(10);
      Console.WriteLine(result);
    }

    public void Method3()
    {
      MyDelegate myDelegate = new MyDelegate(Method1);
      Method4(myDelegate);
      myDelegate = new MyDelegate(Method2);
      Method4(myDelegate);
    }
  }
}
```

**<u>Output</u>**



```
20
100
```

The above written code, just tries to explain that a delegate is nothing but a class. Delegate’s objects could be passed as a parameter as shown in Method4.

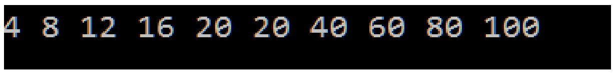
*Point to remember:* Since a delegate is a datatype, it could be passed to a method.

In the first and second invocation of method Method4 , the same delegate is passed a type of MyDelegate. The first time it calls for the method MyDelegate and the second time the method Method2. Using the object myDelegate, different method is executed each time. Indirectly  a different method is given each time to Method4 as a parameter. This is like writing a generic and abstract code, where Method4 really doesn’t care about how it is called, and what parameters are passed to it. It segregates execution from the implementation.

## Lab 7:

```
using System;

namespace DelegatesAndEvents
{
  public class Program
  {
    public static void Main()
  {
   DelegateExercises delegateExercises = new DelegateExercises();
   delegateExercises.Method3();
   Console.ReadLine();
  }
 }

 public class DelegateExercises
 {
  public delegate int MyDelegate(int intValue);

  int Method1(int intMethod1)
  {
   return intMethod1*4;
  }

  int Method2(int intMethod1)
  {
   return intMethod1*20;
  }

  public void Method4(MyDelegate myDelegate)
  {
   for (int i = 1; i <= 5; i++)
    System.Console.Write(myDelegate(i) + " ");
  }

  public void Method3()
  {
   MyDelegate myDelegate = new MyDelegate(Method1);
   Method4(myDelegate);
   myDelegate = new MyDelegate(Method2);
   Method4(myDelegate);
  }
 }
}
```

**Output**



```
4 8 12 16 20 20 40 60 80 100
```

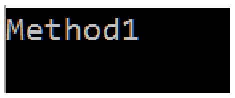
In the above implementation, a more segregation and abstraction is achieved. This implementation says that What we are repeating over and over again is that a delegate can also be executed from within a loop construct as well. A delegate in above code is  passed as an argument to a method, and it executes the name of the method at run time, without knowing its detail at compile time.

## Lab 8:

```
using System;

namespace DelegatesAndEvents
{
 public class Program
```

```
  {
    public static void Main()
    {
      DelegateExercises delegateExercises = new DelegateExercises();
      delegateExercises.Method3();
      Console.ReadLine();
    }

  }
  public delegate void MyDelegate();

  public class DelegateExercises
  {
    void Method1()
    {
      System.Console.WriteLine("Method1");
    }
    public void Method3()
    {
      MyDelegate myDelegate = new MyDelegate(Method1);
      myDelegate();
    }
  }

}
```

**Output**



```
Method1
```

The output is Method1 because MyDelegate being a delegate defines a class that extends System.Delegate.

## Lab 9:

```
using System;

namespace DelegatesAndEvents
{
  public delegate void MyDelegate();
  public class DelegateExercises : MyDelegate
  {
  }
}
```

**Output:**

**Error : 'DelegateExercises': cannot derive from sealed type 'MyDelegate'**

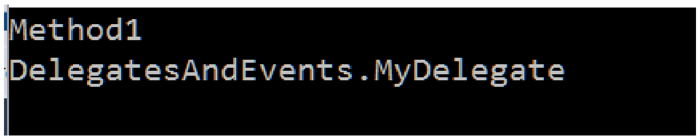
A delegate is represented internally with a class of same name. In the above code, the class MyDelegate is implicitly sealed and another class cannot derive from a sealed class.

*Point to remember:* System.Delegate is an abstract class and all the delegates automatically derive from this class.

## Lab 10:

```
using System;

namespace DelegatesAndEvents
{
  public class Program
  {
    public static void Main()
    {
      DelegateExercises delegateExercises = new DelegateExercises();
      delegateExercises.Method3();
      Console.ReadLine();
    }
  }

  public delegate void MyDelegate();

  public class DelegateExercises
  {
    void Method1()
    {
      System.Console.WriteLine("Method1");


    }

    public void Method3()
    {
      MyDelegate myDelegate = new MyDelegate(Method1);
      myDelegate();
      System.Console.WriteLine(myDelegate.ToString());
    }
  }


}
```

**Output**



```
Method1
DelegatesAndEvents.MyDelegate
```

This simply shows that .ToString() method could be called over delegates as well because System.Delegate also derives from the ultimate base class i.e. Object.

## Lab 11:

```
public delegate void MyDelegate();

 public class DelegateExercises
 {
   void Method3()
   {
     System.Console.WriteLine(MyDelegate.ToString());
   }
 }
```

**Output:**
**Error : An object reference is required for the non-static field, method, or property 'object.ToString()'**

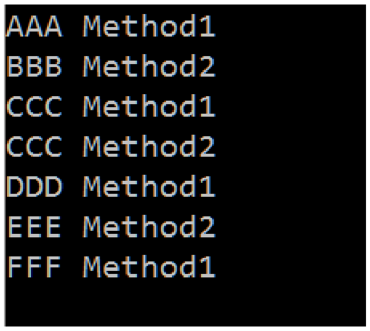
The name of the delegate MyDelegate could not be used with .ToString() method because it is not static where as myDelegate as an object of the class could be used with no errors.

## Lab 12:

```
using System;

namespace DelegatesAndEvents
{
  using System;

  delegate void ExampleDelegate(string xyz);

  class Program
  {
    public static void Method1(string xyz)
    {
      Console.WriteLine(xyz + " Method1");
    }

    public static void Method2(string xyz)
    {
      Console.WriteLine(xyz + " Method2");
    }

    public static void Main()
    {
      ExampleDelegate ex1Delegate, ex2Delegate, ex3Delegate, myDelegate;

      ex1Delegate = new ExampleDelegate(Method1);
      ex2Delegate = new ExampleDelegate(Method2);
      ex3Delegate = ex1Delegate + ex2Delegate;
      myDelegate = ex1Delegate - ex2Delegate;
      ex1Delegate("AAA");
      ex2Delegate("BBB");
      ex3Delegate("CCC");
      myDelegate("DDD");
      myDelegate = ex3Delegate - ex1Delegate;
      myDelegate("EEE");
      myDelegate = ex3Delegate - ex2Delegate;
      myDelegate("FFF");
```

```
        Console.ReadLine();
      }
    }
}
```

**Output**



```
AAA Method1
BBB Method2
CCC Method1
CCC Method2
DDD Method1
EEE Method2
FFF Method1
```

Now this is a bit more complex scenario, but worth understanding. Instances ex1Delegate, ex2Delegate, ex3Delegate, myDelegate are delegate type objects. Instance ex1Delegate represents Method1 and ex2Delegate represents Method2.  So ex1Delegate("AAA") calls Method1 and ex2Delegate("BBB") calls Method2. Instance ex3Delegate shows the addition of ex1Delegate and ex2Delegate. Here addition does not mean literally adding two delegates as a number, but it means that both the delegates should be invoked and executed. First, Method1 is called and then Method2, and it is clearly visible in the output as well. Now the instance myDelegate is initialized to ex1Delegate – ex2Delegate, this again is not a mathematical subtraction, but this implementation will remove methods contained in Method2 from Method1.Since we do not have a common method in method2 , it has no significance and Method1 is invoked.

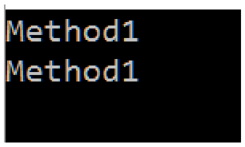
The instance myDelegate is again made equal to ex3Delegate- ex1Delegate. Now this will eliminate all the methods that delegate ex1Delegate represents from object ex3Delegate. The instance ex3Delegate as shown earlier stands for methods Method1 and Method2, now since ex1Delegate represents method Method1,therefore Method1 gets eliminated from myDelegate. So Method2 is called.
In another case of ex3Delegate – ex2Delegate, had the instance ex3Delegate executed , both Method1 and Method2 would get called. But since we are doing a subtraction, of ex2Delegate, only Method1 is called.

*Point to remember:* One can call as many methods we want through Delegates.

## Lab 13:

```
using System;

namespace DelegatesAndEvents
{
  public class Program
  {
    public static void Main()
    {
      DelegateExercises delegateExercises = new DelegateExercises();
      delegateExercises.Method3();
      Console.ReadLine();
    }
  }

  public delegate int MyDelegate(out int i);

  public class DelegateExercises
  {
    int Method1(out int i)
    {
      System.Console.WriteLine("Method1");
      i = 10;
      return 0;
    }

    public void Method3()
    {
      MyDelegate myDelegate = new MyDelegate(Method1);
      MyDelegate myDelegate1 = new MyDelegate(Method1);
      MyDelegate myDelegate2 = myDelegate + myDelegate1;
      int intValue;
      myDelegate2(out intValue);

    }
  }

}
```

**Output**



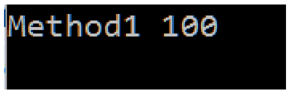
```
Method1
Method1
```

In the above example we see, that a delegate method could also accept out parameters, and hence we get our result too.

## Lab 14:

```
using System;

namespace DelegatesAndEvents
{
  public class Program
  {
    public static void Main()
    {
      DelegateExercises delegateExercises = new DelegateExercises();
      delegateExercises.Method3();
      Console.ReadLine();
    }
  }
  public delegate int MyDelegate(out int i);

  public class DelegateExercises
  {
    int Method1(out int i)
    {
      i = 100;
      System.Console.WriteLine("Method1 " + i);
      return 0;
    }
    public void Method3()
    {
      MyDelegate myDelegate = new MyDelegate(Method1);
      MyDelegate myDelegate1 = null;
      MyDelegate myDelegate2 = myDelegate + myDelegate1;
      int intValue;
      myDelegate2(out intValue);
    }
  }

 }
```

**Output**



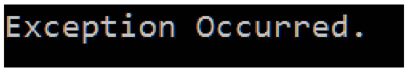
```
Method1 100
```

The same code works now. The only difference is that one of the delegate is assigned to null. These are the rules of simple delegates, but the rules of a multicast delegate are more constrained.
Two separate delegates can point to a same function and target object. So '+' operator helps us add delegates and the '-' operator helps us remove one delegate from another.

## Lab 15:

```
using System;

namespace DelegatesAndEvents
{
  public class Program
  {
    public static void Main()
    {
      DelegateExercises delegateExercises = new DelegateExercises();
      try
      {
        delegateExercises.Method3();
        Console.ReadLine();
      }
      catch (System.Exception ex)
      {
        System.Console.WriteLine("Exception Occurred.");
        Console.ReadLine();

      }
    }
  }
  public delegate void MyDelegate();

  public class DelegateExercises
  {
    void Method1()
    {
      throw new System.Exception();
    }

    public void Method3()
    {
      MyDelegate myDelegate = new MyDelegate(Method1);
      myDelegate();
    }
  }

}
```

**<u>Output</u>**



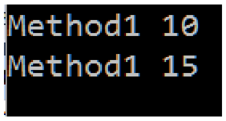
```
Exception Occurred.
```

In the above code, myDelegate() calls method Method1.
This method throws an exception. We choose not to handle this exception in Method3 but in Main method where Method3 is called. We see that exception caused by a delegate keeps on propagating until it is caught.

## Lab 16:

```
using System;

namespace DelegatesAndEvents
{
  public class Program
  {
    public static void Main()
    {
      DelegateExercises delegateExercises = new DelegateExercises();
      delegateExercises.Method3();
      Console.ReadLine();

    }
  }

  public delegate void MyDelegate(ref int intValue);

  public class DelegateExercises
  {
    void Method1(ref int intValue)
    {
      intValue = intValue + 5;
      System.Console.WriteLine("Method1 " + intValue);
    }
    public void Method3()
    {
      MyDelegate myDelegate = new MyDelegate(Method1);
      MyDelegate myDelegate1 = new MyDelegate(Method1);
      MyDelegate myDelegate2 = myDelegate + myDelegate1;
      int intParameter = 5;
      myDelegate2(ref intParameter);
    }
  }

}
```

**<u>Output</u>**



```
Method1 10
Method1 15
```

In the earlier example, we saw 'out' parameter and came to a conclusion that they are not used in multicast delegates. Where as in above example we see, that passing the parameter by 'ref' is allowed in multicast delegate.

## Conclusion

This article covered the topic of Delegate in detail. Delegates are very crucial to understand but are tricky to implement. I hope this post helped the readers to get an insight of delegates.

## Other Series

My other series of articles:

- Learning MVC
- Learning WebAPIs in .Net
- Diving into Visual Studio 2015

For more technical articles you can reach out to **CodeTeddy**.