Learning C# (Day 11): Events in C# (A Practical Approach) – Akhil Mittal

Table of Contents:

Introduction:

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

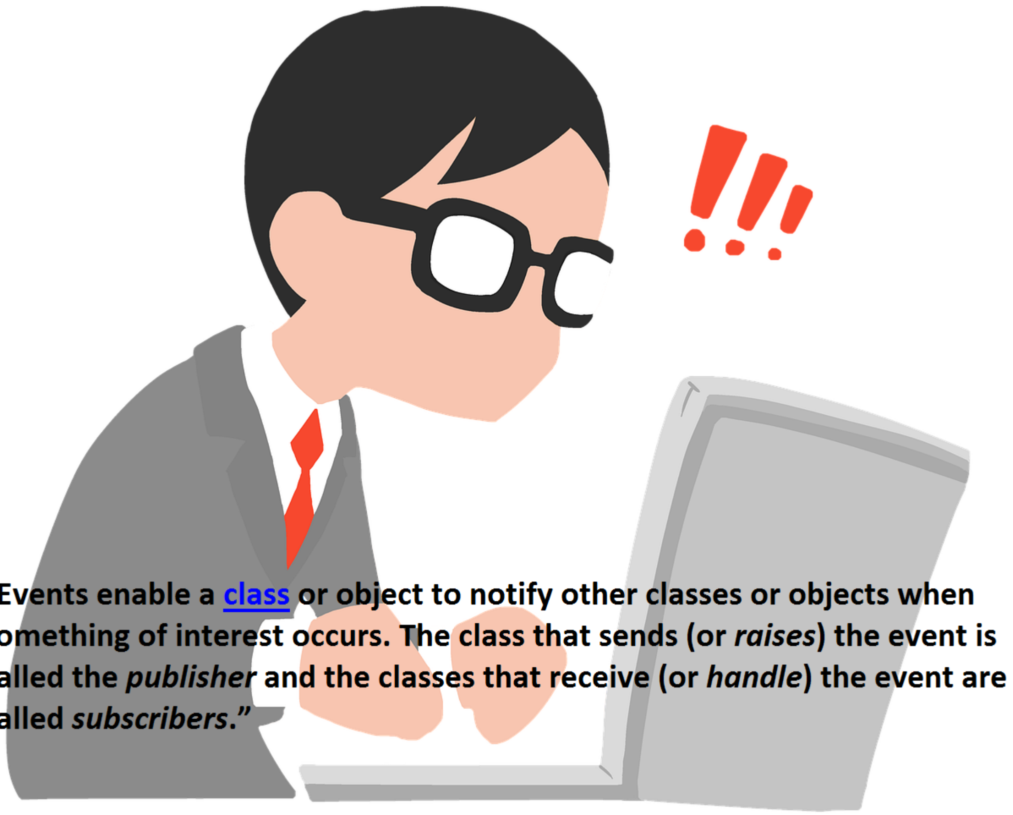


Image source: https://pixabay.com/en/surprised-salaried-worker-computer-1184889/

Events (The definition):

Let’s start with the definition taken from MSDN

“Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or *raises*) the event is called the *publisher* and the classes that receive (or *handle*) the event are called *subscribers*.”

Other Articles of the series:

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

- Diving Into OOP (Day 1) : Polymorphism and Inheritance(Early Binding/Compile Time Polymorphism).
- Diving into OOP (Day 2) : Polymorphism and Inheritance (Inheritance).

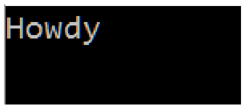
Events

To explain the concept in detail, I have created a solution in Visual Studio 2015 with a console project named DelegatesAndEvents. A new class is created named EventExercises and another class named Program is created that contains Main() method as an entry point for execution.

Lab1

```
using System;

namespace DelegatesAndEvents
{
    class Program
    {
        public static void Main()
        {
            EventExercises eventExercises = new EventExercises();
            eventExercises.Method1();
            Console.ReadLine();
        }
    }

    public class EventExercises
    {
        public void Method1()
        {
            System.Console.WriteLine("Howdy");
        }
    }

}
```

Output



```
Howdy
```

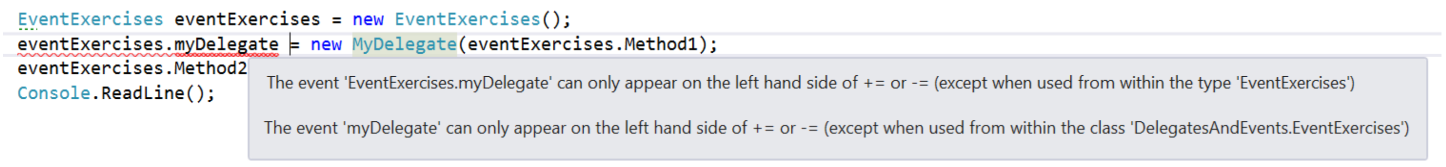
The above mentioned implementation is very common and very straightforward. There is a class named EventExercises that contains a method named Method1 which writes something to console. This method is called in the Main method through an instance of EventExercises class named eventExercises directly. Simple enough. But let's try to call this method indirectly in the following example.

```
using System;

namespace DelegatesAndEvents
{
    public delegate void MyDelegate();
    class Program
    {
        public static void Main()
        {
            EventExercises eventExercises = new EventExercises();
            eventExercises.myDelegate += new MyDelegate(eventExercises.Method1);
            eventExercises.Method2();
            Console.ReadLine();
        }
    }

    public class EventExercises
    {
        public event MyDelegate myDelegate;

        public void Method2()
        {
            if (myDelegate != null)
                Method1();
        }

        public void Method1()
        {
            System.Console.WriteLine("Howdy");
        }
    }
}
```

Output

```
Howdy
```

So , we did the same thing but in a different way. A new delegate named MyDelegate is created, then an object was created for MyDelegate and we passed the name of the method as a parameter to it i.e. Method1. Since Method1 is located in EventExercises class and since we are in Main() method of Program class so we gave it the full name i.e. eventExercises. Here we see that the instance myDelegate is an event and not a delegate and we use the syntax += instead of = else it would result in an error as shown below,



```
EventExercises eventExercises = new EventExercises();
eventExercises.myDelegate = new MyDelegate(eventExercises.Method1);
eventExercises.Method2
Console.ReadLine();
```

The event 'EventExercises.myDelegate' can only appear on the left hand side of += or -= (except when used from within the type 'EventExercises')

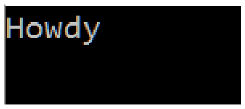
The event 'myDelegate' can only appear on the left hand side of += or -= (except when used from within the class 'DelegatesAndEvents.EventExercises') i.e. The event “EventExercises.myDelegate” can only appear on the left hand side of += or -+= (except when used from within the type 'EventExercises')

The instance myDelegate’s definition is somewhat different. In the former example we basically kept the return value of the object of delegate, but here we are appending a new keyword named “event” to that instance. And the Method2 is invoked from Main. We check the value of the myDelegate in Method2 and then we call Method1() if the value is not null. Therefore if we remove the line new MyDelegate, then myDelegate doesn’t gets initialized and will always be null and so Method1 will not be called.

Lab2

```
using System;

namespace DelegatesAndEvents
{
  public delegate void MyDelegate();

  class Program
  {
    public static void Main()
    {
      EventExercises eventExercises = new EventExercises();
      eventExercises.myDelegate += new MyDelegate(eventExercises.Method1);
      eventExercises.Method2();
      eventExercises.myDelegate -= new MyDelegate(eventExercises.Method1);
      eventExercises.Method2();
    }
  }

  public class EventExercises
  {

    public event MyDelegate myDelegate;

    public void Method2()
    {
      if (myDelegate != null)
        Method1();
    }

    public void Method1()
    {
      System.Console.WriteLine("Howdy");
      Console.ReadLine();
    }
  }

}
```

<u>Output</u>



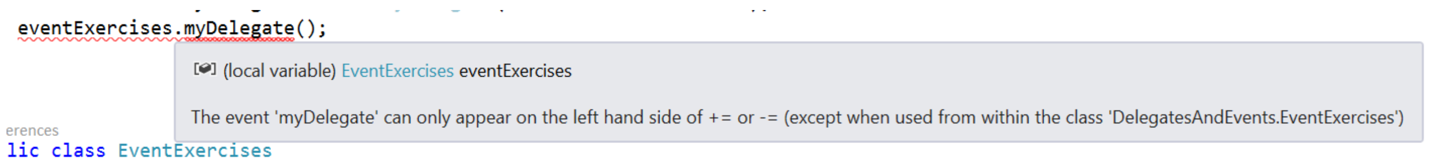
```
Howdy
```

So, Method2 is executed twice in the Main() method. For the first time, the instance myDelegate is not null, and then we do -= in the next line that means we basically subtract MyDelegate from the event myDelegate, but in both the cases the delegate is attached to Method1. When we subtract, the common method is dropped and event instance myDelegate is null, therefore we first added Method1 and then removed it.

Lab3

```
using System;

namespace DelegatesAndEvents
{
  public delegate void MyDelegate();

  class Program
  {
    public static void Main()
    {
      EventExercises eventExercises = new EventExercises();
      eventExercises.myDelegate += new MyDelegate(eventExercises.Method1);
      eventExercises.myDelegate();
    }
  }

  public class EventExercises
  {
    public event MyDelegate myDelegate;

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

}
```

__**Output**__



```
eventExercises.myDelegate();
(local variable) EventExercises eventExercises
The event 'myDelegate' can only appear on the left hand side of += or -= (except when used from within the class 'DelegatesAndEvents.EventExercises')
erences
lic class EventExercises
```

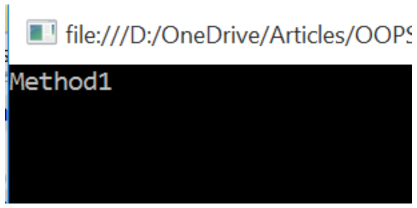
Error CS0070 The event 'EventExercises.myDelegate' can only appear on the left hand side of += or -= (except when used from within the type 'EventExercises').

So , we see here that like delegates, event also cannot be used directly.

Lab4

```
using System;

namespace DelegatesAndEvents
{
  public delegate void MyDelegate();

  class Program
  {
    public static void Main()
    {
      EventExercises eventExercises = new EventExercises();
      eventExercises.myDelegate += new MyDelegate(eventExercises.Method1);
      eventExercises.pqr();
    }
  }
  public class EventExercises
  {
    public event MyDelegate myDelegate;
    public void pqr()
    {
      myDelegate();
    }
    public void Method1()
    {
      System.Console.WriteLine("Method1");
      Console.ReadLine();
    }
  }

}
```

<u>Output</u>


file:///D:/OneDrive/Articles/OOPS

```
Method1
```

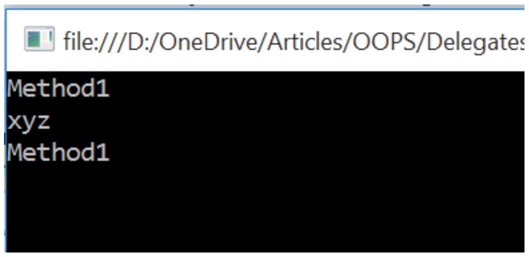
We get the output as Method1.
So, we see here that whatever result we were trying to achieve could also be achieved without events as well. In the code above, we attached an event myDelegate to a method Method1 in class EventExercises. Then method pqr was called by calling myDelegate() from inside it. This eventually invokes the event object myDelegate with a method Method1 in EventExercises class.

Lab5

This Lab explains how we can call more and more methods.

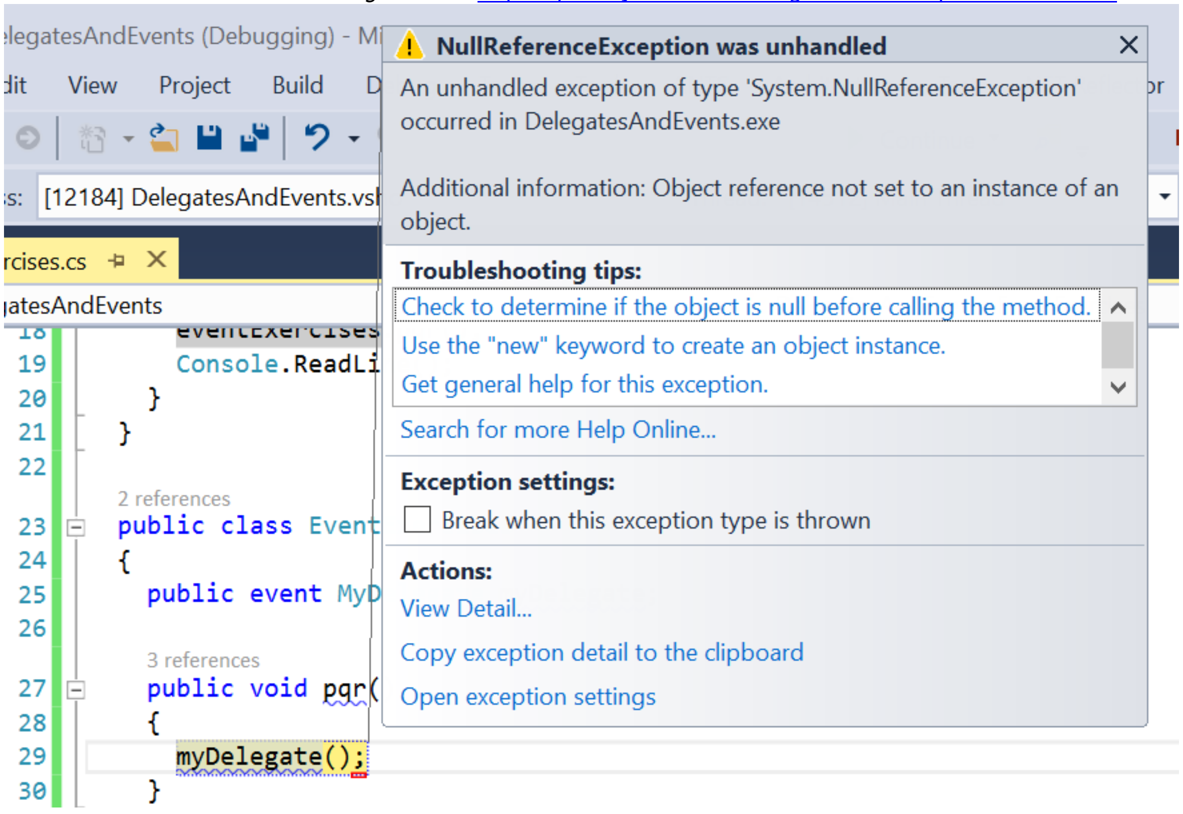
```
using System;

namespace DelegatesAndEvents
{
  public delegate void MyDelegate();

  class Program
  {
    public static void Main()
    {
      EventExercises eventExercises = new EventExercises();
      eventExercises.myDelegate += new MyDelegate(eventExercises.Method1);
      eventExercises.myDelegate += new MyDelegate(eventExercises.xyz);
      eventExercises.pqr();
      eventExercises.myDelegate -= new MyDelegate(eventExercises.xyz);
      eventExercises.pqr();
      eventExercises.myDelegate -= new MyDelegate(eventExercises.Method1);
      eventExercises.pqr();
      Console.ReadLine();
    }
  }

  public class EventExercises
  {
    public event MyDelegate myDelegate;

    public void pqr()
    {
      myDelegate();
    }

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

    public void xyz()
    {
      System.Console.WriteLine("xyz");
    }
  }
}
```

Output



```
file:///D:/OneDrive/Articles/OOPS/Delegates
Method1
xyz
Method1
```

Image source : https://pixabay.com/en/road-sign-usa-trouble-problem-1274312/


legatesAndEvents (Debugging) - Mi
dit
View
Project
Build
[12184] DelegatesAndEvents.vsh
rcises.cs
jatesAndEvents

```
            eventExercises
            Console.ReadLi
        }
    }

    2 references
    public class Event
    {
        public event MyD

        3 references
        public void pqr(
        {
            myDelegate();
        }
```

NullReferenceException was unhandled

An unhandled exception of type 'System.NullReferenceException' occurred in DelegatesAndEvents.exe

Additional information: Object reference not set to an instance of an object.

Troubleshooting tips:

Check to determine if the object is null before calling the method.
Use the "new" keyword to create an object instance.
Get general help for this exception.

Search for more Help Online...

Exception settings:

☐ Break when this exception type is thrown

Actions:

View Detail...
Copy exception detail to the clipboard
Open exception settings

So, we see that executing the code above, we first get the output as method1 xyz Method1 written on console followed by an exception in the code i.e. System.NullReferenceException.
This scenario is more or less like delegates. We add two methods to myDelegate event, therefore firstly we call method pqr in which we execute the myDelegate event, and then we call Method1 and method pqr as if they are associated to the myDelegate event. When we do +=, it adds a method and when we do -= it eliminates the method from the list of event, and when we invoke myDelegate event, only Method1 is called as method xyz is eliminated from the list. At the end we also remove Method1 from the list and event value will be null. So when we execute an event having no methods to notify, we end up having a runtime exception, therefore a null check for the event is always required when we use it.

Lab6

```
using System;
```

```
namespace DelegatesAndEvents
{
  public delegate void MyDelegate();
  class Program
  {
    public static void Main()
    {
      EventExercises eventExercises = new xxx();
      Console.ReadLine();
    }
  }
  public class EventExercises
  {
    public event MyDelegate myDelegate;
    public void Method1()
    {
      System.Console.WriteLine("Method1");
    }
  }
  public class xxx : EventExercises
  {
    public void pqr()
    {
      myDelegate();
    }
  }

}
```

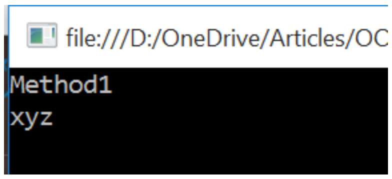
Output
Compile time error: The event 'EventExercises.myDelegate' can only appear on the left hand side of += or -= (except when used from within the type 'EventExercises')

So, basically to call a method of the same class we use an event. EventExercises is the base class to xxx, and it is clear that they are not in the same class. Therefore an event in EventExercises class cannot be used in any other class. Usually derived classes inherit the base class things, but events are exceptional and show the error at compile time rather than on run time.

Lab7

```
using System;

namespace DelegatesAndEvents
{
  public delegate void MyDelegate();
  class Program
  {
    public static void Main()
    {
      EventExercises eventExercises = new EventExercises();
      eventExercises.myDelegate += new MyDelegate(eventExercises.Method1);
      xxx x = new xxx();
      eventExercises.myDelegate += new MyDelegate(x.xyz);
      eventExercises.pqr();
      Console.ReadLine();
    }
  }
  public class EventExercises
  {
    public event MyDelegate myDelegate;
    public void pqr()
    {
      myDelegate();
    }
    public void Method1()
    {
      System.Console.WriteLine("Method1");
    }
  }
  public class xxx
  {
    public void xyz()
    {
      System.Console.WriteLine("xyz");
    }
  }
}
```

Output



```
file:///D:/OneDrive/Articles/OC
Method1
xyz
```

Here we see actual strength of events. We primarily used the same code as in previous example and named the method as xyz. EventExercises does not contain tis message but it is picked from xxx. The event invocation works accordingly. We already saw this example for delegates in the previous article on delegates.

Lab8

```
using System;

namespace DelegatesAndEvents
{
  public delegate void MyDelegate();

  public class Program
  {
    public event MyDelegate myDelegate;
    //public MyDelegate myDelegate;
    public void add_myDelegate(MyDelegate a)
    {
      myDelegate += a;
    }

    public void remove_myDelegate(MyDelegate a)
    {
      myDelegate -= a;
    }
  }
}
```

Output

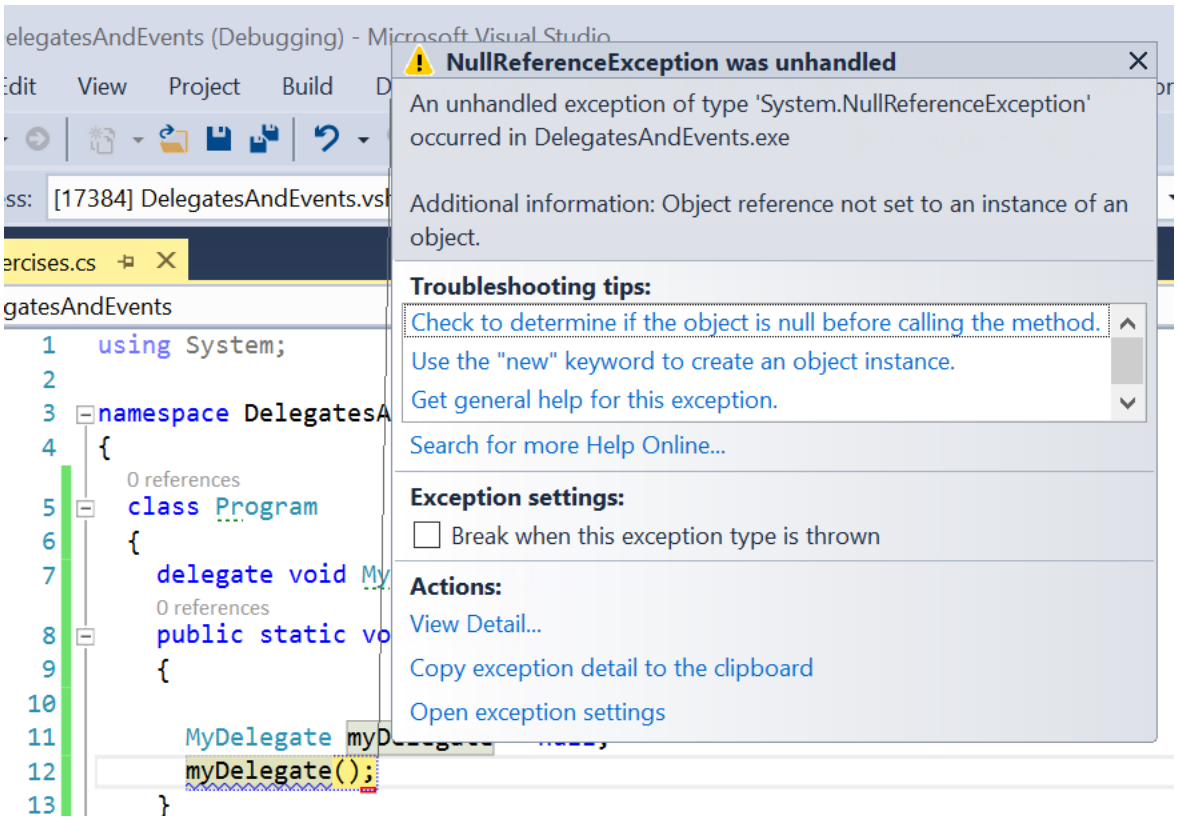
Compile time error : *Type 'Program' already reserves a member called 'add_myDelegate' with the same parameter types*
Type 'Program' already reserves a member called 'remove_myDelegate' with the same parameter types

Here, at every instance when we try to create an event object, there are two methods created automatically in the class. The two methods are the event names prefixed by add_ and remove_. Compiler automatically adds code for these methods. That clearly states that a class could not contain methods with the same name that has an event, on the other hand it also means that the code of the event overwrites the method code. Something similar to C++, happens here, where C++ code was converted to C code by the compiler, which then was executed by the compiler as a C code. The methods that are invoked by the events are named as event handlers and they are responsible for providing notifications to the class.

Lab9

```
using System;

namespace DelegatesAndEvents
{
  class Program
  {
    delegate void MyDelegate();
    public static void Main()
    {

      MyDelegate myDelegate = null;
      myDelegate();
    }
  }

}
```

<u>Output</u>


elegatesAndEvents (Debugging) - Microsoft Visual Studio
dit View Project Build D
ss: [17384] DelegatesAndEvents.vsh
ercises.cs
gatesAndEvents

NullReferenceException was unhandled

An unhandled exception of type 'System.NullReferenceException' occurred in DelegatesAndEvents.exe

Additional information: Object reference not set to an instance of an object.

Troubleshooting tips:

Check to determine if the object is null before calling the method.
Use the "new" keyword to create an object instance.
Get general help for this exception.

Search for more Help Online...

Exception settings:

☐ Break when this exception type is thrown

Actions:

View Detail...
Copy exception detail to the clipboard
Open exception settings

```
using System;

namespace DelegatesA
{
    0 references
    class Program
    {
        delegate void My
        0 references
        public static vo
        {

            MyDelegate myD
            myDelegate();
        }
```

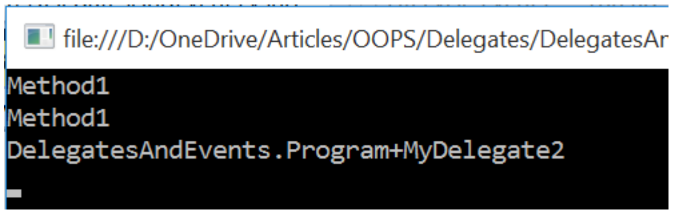
Runtime Exception: System.NullReferenceException

For delegates when executed, a compile time check is not performed, but a run time check is performed. Since delegate myDelegate was not initialized, so it was null, therefore it was obvious to get a null reference exception.

Lab10

```
using System;

namespace DelegatesAndEvents
{
  class Program
  {
    public delegate void MyDelegate();

    public delegate void MyDelegate2();

    public static void Main()
    {
      Program programInstance = new Program();
      MyDelegate del = new MyDelegate(programInstance.Method1);
      del();
      MyDelegate2 del2 = new MyDelegate2(del);
      del2();
      System.Type typeDel = typeof(MyDelegate2);
      System.Console.WriteLine(typeDel.FullName);
      if (del2 is MyDelegate)
        System.Console.WriteLine("true");
      Console.ReadLine();
    }

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

}
```

Output



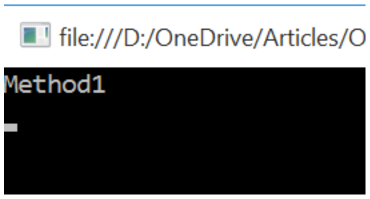
```
file:///D:/OneDrive/Articles/OOPS/Delegates/DelegatesAr
Method1
Method1
DelegatesAndEvents.Program+MyDelegate2
```

We see here that a constructor of a delegate can not only take the method name as a parameter, but also can take the name of any other delegate as a parameter. Delegate del2 of type MyDelegate2 is initialized to del of type MyDelegate1, this basically creates the copy of the prior delegate. Therefore if we see, we now have to objects of delegates pointing to a single method named Method1. But instead of assigning MyDelegate2 to type of MyDelegate1, the data type of MyDelegate2 remains unaffected and it is not changed and remains to datatype of MyDelegate2 only.

Lab11

```
using System;

namespace DelegatesAndEvents
{
  class Program : EventExercises

  {
    public delegate void MyDelegate();
    public static void Main()
    {
      Program programInstance = new Program();
      MyDelegate del = new MyDelegate(programInstance.Method1);
      del();
      Console.ReadLine();
    }
  }
  class EventExercises
  {
    public void Method1()
    {
      System.Console.WriteLine("Method1");
    }
  }
}
```

Output


file:///D:/OneDrive/Articles/O

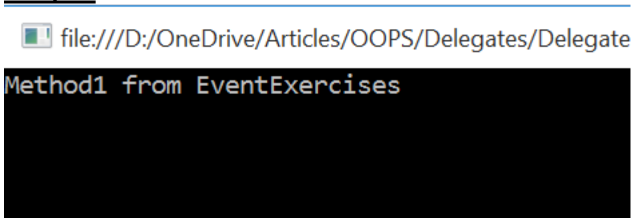
```
Method1
```

The example is just to show that the method that is in base class could also be passed to a delegate as a parameter without any error.

Lab12

```
using System;

namespace DelegatesAndEvents
{
  class Program : EventExercises
  {
    public delegate void MyDelegate();
    public void PQR()
    {
      MyDelegate del = new MyDelegate(base.Method1);
      del();
    }
    public static void Main()
    {
      Program programInstance = new Program();
      programInstance.PQR();
      Console.ReadLine();
    }
    public void Method1()
    {
      System.Console.WriteLine("Method1 from Program");
    }
  }
  class EventExercises
  {
    public void Method1()
    {
      System.Console.WriteLine("Method1 from EventExercises");
    }
  }
}
```

Output


file:///D:/OneDrive/Articles/OOPS/Delegates/Delegate

```
Method1 from EventExercises
```

We can also use the keyword base to call the function from the base class. If you remember, base calls code from the base class and not from the derived class. This is in spite of what the documentation says and we quote verbatim 'If the method group resulted from a base-access, an error occurs'. There is no way known to man that can change the method associated with a delegate once the delegate has been created. It remains the same for the entire lifetime of the delegate. The parameter to a delegate creation cannot be a constructor, indexer, property or obviously a user define operator even though they carry code. We are left with only one choice as a parameter, a method.

Lab13

```
using System;

namespace DelegatesAndEvents
{
  class Program
  {
    public delegate void MyDelegate();
    public static void Main()
    {
      MyDelegate d = new MyDelegate(Program);
    }
  }

}
```

Output

Compile time error: 'Program' is a type, which is not valid in the given context

This means that delegates cannot hold constructors as a parameter ☺

Some more interesting stuff

Lab14 : Event Properties

```
namespace DelegatesAndEvents
{
  public delegate void MyDelegate();
  public class Program
  {
    public event MyDelegate delegate1
    {
      add
      {
        return null;
      }
    }
  }
}
```

Output

Compile time error:
'Program.delegate1': event property must have both add and remove accessors
Since 'Program.delegate1.add' returns void, a return keyword must not be followed by an object expression

Like normal properties in C#, if we are creating the event property , we should keep in mind that it should have both add and remove accessors.

Lab15 : Event Data Type

```
namespace DelegatesAndEvents
{
  public class MyDelegate
  {

  }
  public class Program
  {
    public event MyDelegate delegate1
    {
      add { return null; }
      remove { }
    }
  }

}
```

Output

Compile time error:

'Program.d1': event must be of a delegate type
Since 'Program.delegate.add' returns void, a return keyword must not be followed by an object expression

This error means that event should always be the data type of a delegate and not of user defined.

Lab16

```
namespace DelegatesAndEvents
{
  delegate void MyDelegate();
  interface MyInterface
  {
    event MyDelegate del = new MyDelegate();
  }

}
```

Output

Compile time error:
'MyInterface.del': event in interface cannot have initializer
'MyDelegate' does not contain a constructor that takes 0 arguments

So this again proves that interfaces can only contain definitions and not implementation code.

Lab16 : Accessors in Interfaces?

```
namespace DelegatesAndEvents
{
  public delegate void MyDelegate();
  public interface IMyInterface
  {
    event MyDelegate del
    {
      remove { }
      add { return null; }
    }
  }
}
```

Output

Compile time error:
 Error   CS0069 An event in an interface cannot have add or remove accessors
 Error   CS0069 An event in an interface cannot have add or remove accessors

The above example demonstrates that in an interface one cannot have accessor code.

Lab17

```
namespace DelegatesAndEvents
{
  public delegate void MyDelegate();
  public class Program
  {
    public event MyDelegate del1;
    public static void Main()
    {
    }
  }
}
```

Output

Compile time warning:
 warning CS0067: The event 'Program. del1 is never used

So our code compiles here, but shows a warning. Compiler gives us a warning when an event is declared but not used.

Lab18 : Event in Interface

```
namespace DelegatesAndEvents
{
  public delegate void MyDelegate();

  interface IMyInterface
  {
    event MyDelegate myDelegate;
  }

  class Program : IMyInterface
  {
    event MyDelegate IMyInterface.myDelegate()
    {
    }
  }

}
```

Output

Compile time error:

- CS0071 An explicit interface implementation of an event must use event accessor syntax
- CS1520 Method must have a return type
- CS0535 'Program' does not implement interface member 'IMyInterface.myDelegate'
- CS0539 'Program.' in explicit interface declaration is not a member of interface
- CS0065 'Program.': event property must have both add and remove accessors

We got a lot of errors in above code.
The error says that we need to use the syntax of the property i.e. get and set, when one try to implement an event that is declared in the interface. Events and interfaces share this strange bonding ☺.

Lab19

```
namespace DelegatesAndEvents
{
  delegate void MyDelegate(int i);
  class Program
  {
    public static void Main()
    {
      MyDelegate myDelegate = new MyDelegate(500);
    }
  }

}
```

Output

Compile time error:
- CS0149 Method name expected

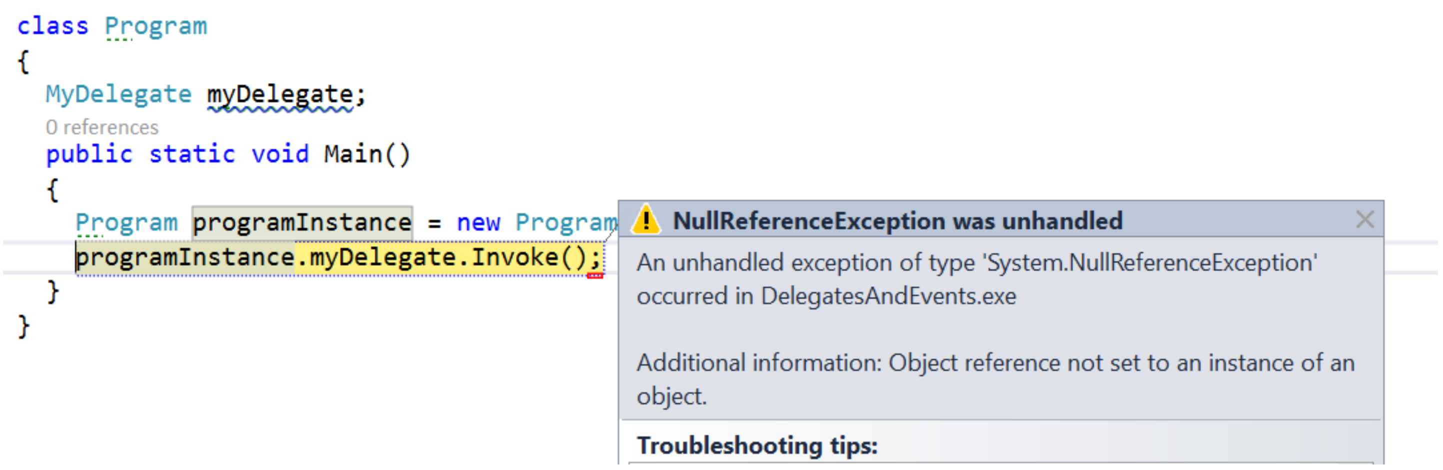
Like we mentioned delegates are like method pointers, which could only point to methods. So while creating delegate object, it requires the name of the method.In the above code, it gives the compiler error because we pass number instead of a method name.

Lab20

```
namespace DelegatesAndEvents
{
  public delegate void MyDelegate();
  class Program
  {
    MyDelegate myDelegate;
    public static void Main()
    {
      Program programInstance = new Program();
      programInstance.myDelegate.Invoke();
    }
  }
}
```

Output

We get a run time error.



```
class Program
{
  MyDelegate myDelegate;
  0 references
  public static void Main()
  {
    Program programInstance = new Program
    programInstance.myDelegate.Invoke();
  }
}
```

NullReferenceException was unhandled

An unhandled exception of type 'System.NullReferenceException' occurred in DelegatesAndEvents.exe

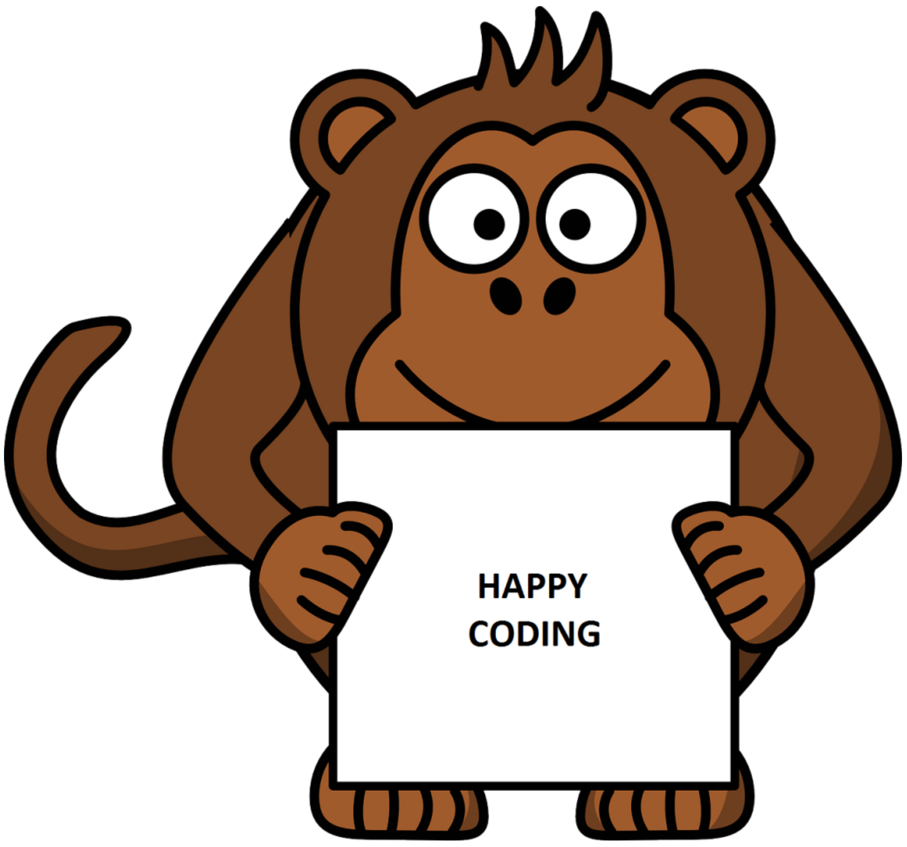
Additional information: Object reference not set to an instance of an object.

Troubleshooting tips:

So, there is only one defined way to use delegates as explained, and we cannot use Invoke method to directly call the delegate

Conclusion

This article covered the topic of Events and some parts of delegates in detail. Events are very crucial and fun to understand but are tricky to implement. I hope this post helped the readers to get an insight of events and delegates.



Other Series

My other series of articles:

- Learning MVC

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

Read more:

- C# and ASP.NET Questions (All in one)
- MVC Interview Questions
- C# and ASP.NET Interview Questions and Answers
- Web Services and Windows Services Interview Questions

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