Diving in OOP (Day 5) : All about access modifiers in C# (Public/Private/Protected/Internal/Sealed/Constants/Static/Readonly Fields)

Table of Contents

Introduction:

Thanks to my readers for their tremendous support which motivated me to continue this OOP series further. We have already covered almost all the aspects of Inheritance and Polymorphism in C#. My article will highlight almost all the aspects/scenarios of access modifiers in C#. We'll learn by doing hands on lab and not only by theory. We'll cover my favourite topic Constants in a very different manner by categorizing the sections in form of "Labs". My effort in this article will be to cover each and every concept to the related topic, so that at the end of the article we confidently say that we know "All about access modifiers in C#". Just dive into OOP.

Image credit: http://en.wikipedia.org/wiki/Colocation_centre

Pre-requisites:

I expect that my readers of this article should have very basic knowledge of C#. The reader should only know the definition of access modifiers. Last but not the least as I always wish that my readers should enjoy while reading this article.

Roadmap:

Let's recall our road map,

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 Abstract 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): All about Properties and Indexers in C#.

Access Modifiers :

Let us take the definition from Wikipedia this time,

"Access modifiers (or access specifiers) are keywords in object-oriented languages that set the accessibility of classes, methods, and other members. Access modifiers are a specific part of programming language syntax used to facilitate the encapsulation of components."

Like the definition says that we can control the accessibility of our class methods and members through access modifiers, let us understand this in detail by taking every access modifier one by one.

Public, Private, Protected at class level:

Whenever we create a class we always want to have the scope to decide who can access certain members of the class. In other words, we would sometimes need to restrict access to the class members. The one thumb rule is that members of a class can freely access each other. A method in one class can always access another method of the same class without any restrictions. When we talk about the default behavior, the same class is

allowed complete access but no else is provided access to the members of the class. The default access modifier is private for class members.

Point to remember: The default access modifier is private for class members.

Let's do some hands on lab.Just open your visual studio and add a console application in c# named **AccessModifiers,** You'll get a Program.cs class file by default.In the same file add a new class named Modifiers and add following code to it,

```
using System;

namespace AccessModifiers
{
   class Modifiers
   {
      static void AAA()
      {
         Console.WriteLine("Modifiers AAA");
      }

      public static void BBB()
      {
         Console.WriteLine("Modifiers BBB");
         AAA();
      }
   }

    class Program
            {
               static void Main(string[] args)
               {
                  Modifiers.BBB();
               }
            }

  }
```

So, your Program.cs file becomes like shown in above code snippet.We added a class Modifiers and two static methods AAA and BBB. Method BBB is marked as public.We call the method BBB from Main method.The method is called directly by the class name because it is marked static.
When we run the application, we get the output as follows,

Output:

```
Modifiers BBB
Modifiers AAA
```

BBB is marked public and so anyone is allowed to call and run it. Method AAA is not marked with any access modifier which automatically makes it private, that is the default. The private modifier has no effect on members of the same class and so method BBB is allowed to call method AAA. Now this concept is called member access.

Modify the Program class and try to access AAA as,

```
class Program
  {
    static void Main(string[] args)
    {
      Modifiers.AAA();
      Console.ReadKey();
    }
  }
```

Output:

'AccessModifiers.Modifiers.AAA()' is inaccessible due to its protection level

So , since method AAA is private therefore no one else can have access to it except Modifiers class.
Now mark the AAA method as protected, our class looks like,

Modifiers:

```
class Modifiers
  {
    protected static void AAA()
    {
      Console.WriteLine("Modifiers AAA");
    }

    public static void BBB()
    {
      Console.WriteLine("Modifiers BBB");
      AAA();
    }
  }
```

Program:

```
  class Program
  {
    static void Main(string[] args)
    {
      Modifiers.AAA();
      Console.ReadKey();
```

```
    }
}
```

Output:

'AccessModifiers.Modifiers.AAA()' is inaccessible due to its protection level

Again the same output. We cannot access the method AAA even after we introduced a new modifier named protected.But BBB can access AAA method because it lies in the same class.

Modifiers in Inheritance:

Let's add one more class and make a relation of base and derived class to our existing class and add one more method to our base class. So our class structure will look something like this,

Modifiers Base Class

```
class ModifiersBase
  {
    static void AAA()
    {
      Console.WriteLine("ModifiersBase AAA");
    }
    public static void BBB()
    {
      Console.WriteLine("ModifiersBase BBB");
    }
    protected static void CCC()
    {
      Console.WriteLine("ModifiersBase CCC");
    }
  }
```

Modifiers Derive Class

```
  class ModifiersDerived:ModifiersBase
  {
    public static void XXX()
    {
      AAA();
      BBB();
      CCC();
    }
  }
```

Program Class

```
class Program
{
    static void Main(string[] args)
    {
        ModifiersDerived.XXX();
        Console.ReadKey();
    }
}
```

Output :

'AccessModifiers.ModifiersBase.AAA()' is inaccessible due to its protection level

Now I this case we are dealing with derived class. Whenever we mark a method with the specifier, protected, we are actually telling C# that only derived classes can access that method and no one else can. Therefore in method XXX we can call CCC because it is marked protected, but it cannot be called from anywhere else including Main function. The method AAA is made private and can be called only from the class ModifiersBase. If we remove AAA from method XXX, the compiler will give no error.

Therefore now we are aware of three important concepts. Private means only the same class has access to the members, public means everybody has access and protected lies in between where only derived classes have access to the base class method.
All the methods for example reside in a class. The accessibility of that method is decided by the class in which it resides as well as the modifiers on the method. If we are allowed an access to a member, then we say that the member is accessible, else it is inaccessible.

Internal modifier at class level:

Let's take one another scenario. Create a class library with a name "AccessModifiersLibrary" in your visual studio. Add a class named ClassA in that class library and mark the class as internal, the code will be like as shown below,

AccessModifiersLibrary.ClassA:

```
namespace AccessModifiersLibrary
{
    internal class ClassA
    {
    }
}
```

Now compile the class, and leave it. Its dll will be generated in ~\AccessModifiersLibrary\bin\Debug folder.

Now in your console application "AccessModifiers" i.e. created earlier. Add the reference of AccessModifiersLibrary library by adding its compiled dll as a reference to AccessModifiers.
In Program.cs of AccessModifiers console application, modify the Program class like shown below,

AccessModifiers.Program:

```
using AccessModifiersLibrary;

namespace AccessModifiers
{
   class Program
   {
      static void Main(string[] args)
      {
         ClassA classA;
      }
   }

}
```

And compile the code,

Output:

Compile time error: 'AccessModifiersLibrary.ClassA' is inaccessible due to its protection level

We encountered this error because the access specifier internal means that we can only access ClassA from AccessModifiersLibrary.dll and not from any other file or code. Internal modifier means that access is limited to current program only. So try never to create a component and mark the class internal as no one would be able to use it.

And what if remove the field internal from ClassA, will the code compile? i.e.,

AccessModifiersLibrary.ClassA:

```
namespace AccessModifiersLibrary
{
   class ClassA
   {
   }
}
```

AccessModifiers.Program:

```
using AccessModifiersLibrary;

namespace AccessModifiers
```

```
{
   class Program
   {
      static void Main(string[] args)
      {
         ClassA classA;
      }
   }

}
```

Output:

Compile time error: 'AccessModifiersLibrary.ClassA' is inaccessible due to its protection level

We again got the same error. We should not forget that by default if no modifier is specified, the class is internal.So our class ClassA is internal by default even if we do not mark it with any access modifier, so the compiler results remains same.

Had the class ClassA marked public, everything would have gone smooth without any error.

Point to remember: A class marked as internal can only be have its access limited to the current assembly only.

Namespaces with modifiers:

Let's for fun, mark a namespace of AccessModifiers class library as public in Program class,

Program:

```
public namespace AccessModifiers
{
   class Program
   {
      static void Main(string[] args)
      {

      }
   }
}
```

Compile the application,

Output:

Compile time error: A namespace declaration cannot have modifiers or attributes

Point to remember: Namespaces as we see by default can have no accessibility specifiers at all. They are by default public and we cannot add any other access modifier including public again too.

Private Class:

Let's do one more experiment and mark the class Program as private, so our code becomes,

```
namespace AccessModifiers
{
   private class Program
   {
      static void Main(string[] args)
      {

      }
   }
}
```

Compile the code,

Output:

Compile time error: Elements defined in a namespace cannot be explicitly declared as private, protected, or protected internal

So, **Point to remember:** A class can only be public or internal. It cannot be marked as protected or private. The default is internal for the class.

Access modifiers for the members of the class:

Now here is a big statement, that the members of a class can have all the above explained access modifiers, but default modifier is private.

Point to remember: Members of a class can be marked with all the access modifiers, and the default access modifier is private.

What if we want to mark a method with two access modifiers?

```
namespace AccessModifiers
{
   public class Program
```

```
    {
        static void Main(string[] args)
        {
        }

        public private void Method1()
        {

        }
    }
}
```

Compile the code,

Output:

Compile time error: More than one protection modifier

Therefore we can't mark a member with more than one access modifier often. But there are such scenarios too, we'll cover them in next sections. Already defined types like int and object have no accessibility restrictions. They can be used anywhere and everywhere.

Internal class and public method:

Create a class library with a class named ClassA marked internal and have a public method MethodClassA(), as,

```
namespace AccessModifiersLibrary
{
    internal class ClassA
    {
        public void MethodClassA(){}
    }
}
```

Add the reference of class library to our console application. Now in Program.cs of console application, try to access that method MethodClassA of ClassA.

Program:

```
using AccessModifiersLibrary;

 namespace AccessModifiers
{
    public class Program
    {
        public static void Main(string[] args)
```

```
    {
        ClassA classA = new ClassA();
        classA.MethodClassA();
    }

  }
}
```

Output:

Compile time errors:

'AccessModifiersLibrary.ClassA' is inaccessible due to its protection level
The type 'AccessModifiersLibrary.ClassA' has no constructors defined
'AccessModifiersLibrary.ClassA' is inaccessible due to its protection level
'AccessModifiersLibrary.ClassA' does not contain a definition for 'MethodClassA' and no extension method 'MethodClassA' accepting a first argument of type 'AccessModifiersLibrary.ClassA' could be found (are you missing a using directive or an assembly reference?)

So many errors. The errors are self explanatory though. Even the method MethodClassA of ClassA is public, it could not be accesses in Program class due to protection level of ClassA i.e. internal. The type enclosing the method MethodClassA is internal, so no matter if the method is marked public, we can not access it in any other assembly.

Public class and private method:

Let's make the class ClassA as public and method as private,

AccessModifiersLibrary.ClassA:

```
namespace AccessModifiersLibrary
{
  public class ClassA
  {
    private void MethodClassA(){}
  }
}
```

Program:

```
using AccessModifiersLibrary;

 namespace AccessModifiers
{
```

```
    public class Program
    {
        public static void Main(string[] args)
        {
            ClassA classA = new ClassA();
            classA.MethodClassA();
        }


    }
}
```

Output on compilation:

'AccessModifiersLibrary.ClassA' does not contain a definition for 'MethodClassA' and no extension method 'MethodClassA' accepting a first argument of type 'AccessModifiersLibrary.ClassA' could be found (are you missing a using directive or an assembly reference?)

Now we marked our class Public, still can't access the private method. So for accessing a member of the class, the access modifier of class as well as method is very important.

Public class and internal method:

Make ClassA as public and MethodClassA as internal,

AccessModifiersLibrary.ClassA:

```
namespace AccessModifiersLibrary
{
    public class ClassA
    {
        Internal void MethodClassA(){}
    }
}
```

Program:

```
using AccessModifiersLibrary;

 namespace AccessModifiers
{
    public class Program
    {
        public static void Main(string[] args)
        {
```

```
        ClassA classA = new ClassA();
        classA.MethodClassA();
    }


  }
}
```

Output on compilation:

'AccessModifiersLibrary.ClassA' does not contain a definition for 'MethodClassA' and no extension method 'MethodClassA' accepting a first argument of type 'AccessModifiersLibrary.ClassA' could be found (are you missing a using directive or an assembly reference?)

So an internal marked member means that no one from outside that dll can access the member.

Protected internal:

In the class library make three classes ClassA, ClassB and ClassC, and place the code somewhat like this,

```
namespace AccessModifiersLibrary
{
  public class ClassA
  {
    protected internal void MethodClassA()
    {

    }
  }

  public class ClassB:ClassA
  {
    protected internal void MethodClassB()
    {
      MethodClassA();
    }
  }

  public class ClassC
  {
    public void MethodClassC()
    {
      ClassA classA=new ClassA();
      classA.MethodClassA();
```

```
    }
  }
}
```

And in Program class in our console application, call the MethodClassC of ClassC,

Program:

```
using AccessModifiersLibrary;

namespace AccessModifiers
{
  public class Program
  {
    public static void Main(string[] args)
    {
      ClassC classC=new ClassC();
      classC.MethodClassC();
    }
  }
}
```

Compiler output: The code successfully compiles with no error.

Protected internal modifier indicates two things, that either the derived class or the class in the same file can have access to that method, therefore in above mentioned scenario, the derived class ClassB and the class in the same file i.e. ClassC can access that method of ClassA marked as protected internal.

Point to remember: Protected internal means that the derived class and the class within the same source code file can have access.

Protected member:

In our Program.cs in console application, place the following code,

```
namespace AccessModifiers
{
  class AAA
  {
    protected int a;
    void MethodAAA(AAA aaa,BBB bbb)
    {
      aaa.a = 100;
```

```
        bbb.a = 200;
      }
   }
    class BBB:AAA
    {
       void MethodBBB(AAA aaa, BBB bbb)
       {
          aaa.a = 100;
          bbb.a = 200;
       }
    }
   public class Program
   {
      public static void Main(string[] args)
      {
      }
   }
}
```

Compiler Output:

Cannot access protected member 'AccessModifiers.AAA.a' via a qualifier of type 'AccessModifiers.AAA'; the qualifier must be of type 'AccessModifiers.BBB' (or derived from it)

Class AAA is containing a protected member i.e. a. But to the same class no modifiers make sense. However as a is protected, in the derived class method MethodBBB, we cannot access it through AAA as aaa.a gives us an error. However bbb which looks like BBB does not give an error. To check this out, comment out the line aaa.a=100 in MethodBBB (). This means that we can not access the protected members from an object of the base class, but from the objects of derived class only. This is in spite of the fact that a is a member of AAA i.e. the base class. Even so, we still cannot access it. Also we cannot access a from the method Main.

Accessibility Priority in inheritance:

Program:

```
 namespace AccessModifiers
{
   class AAA
   {

   }
   public class BBB:AAA
    {
```

```
    }
   public class Program
   {
      public static void Main(string[] args)
      {
      }
   }
}
```

Compiler Output:

Compile time error: Inconsistent accessibility: base class 'AccessModifiers.AAA' is less accessible than class 'AccessModifiers.BBB'

The error again gives us one more point to remember,

Point to remember: In between public and internal, public always allows greater access to its members.

The class AAA is by default marked internal and BBB that derives from AAA is made public explicitly. We got an error as the derived class BBB has to have an access modifier which allows greater access than the base class access modifier. Here internal seems to be more restrictive than public.

But if we reverse the modifiers to both the classes i.e. ClassA marked as public and ClassB internal or default, we get rid of the error.

Point to remember: The base class always allows more accessibility than the derived class.

Another scenario,

Program:

```
 namespace AccessModifiers
{
   class AAA
   {

   }
   public class BBB
    {
      public AAA MethodB()
      {
         AAA aaa= new AAA();
         return aaa;
      }
    }
   public class Program
   {
```

```
        public static void Main(string[] args)
        {
        }
    }
}
```

Compiler output: Inconsistent accessibility: return type 'AccessModifiers.AAA' is less accessible than method 'AccessModifiers.BBB.MethodB()'

Here the accessibility of AAA is internal which is more restrictive than public. The accessibility of method MethodB is public which is more than that of the typeAAA. Now the error occurred because return values of a method must have greater accessibility than that of the method itself, which is not true in this case.

Point to remember: The return values of a method must have greater accessibility than that of the method itself.

Program:

```
 namespace AccessModifiers
{
    class AAA
    {

    }
    public class BBB
    {
        public AAA aaa;
    }
    public class Program
    {
        public static void Main(string[] args)
        {
        }
    }
}
```

Compiler Output: Inconsistent accessibility: field type 'AccessModifiers.AAA' is less accessible than field 'AccessModifiers.BBB.aaa'

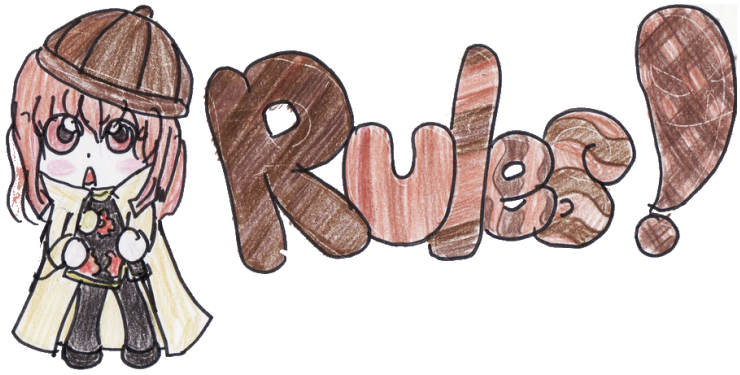


Image credit: http://moyamoyya.deviantart.com/art/Rules-147356501

Now rules are same for everyone. The class AAA or data type aaa is internal. aaa field is public which makes it more accessible than AAA which is internal. So we got the error.

Change the code to,

```
namespace AccessModifiers
{
   class AAA
   {

   }
   public class BBB
   {
      AAA a;
   }
   public class Program
   {
      public static void Main(string[] args)
      {
      }
   }
}
```

The output compilation results in no error.

We learnt a lot about these access modifiers like, public,private,protected,internal, protected internal.We also learnt about their priority of access and usage, let's summarize their details in a tabular format for revision.Later we'll move to other types as well.

Tables taken from : msdn

Declared accessibility	Meaning

public	Access is not restricted.
protected	Access is limited to the containing class or types derived from the containing class.
internal	Access is limited to the current assembly.
protected internal	Access is limited to the current assembly or types derived from the containing class.
private	Access is limited to the containing type.

"Only one access modifier is allowed for a member or type, except when you use the protected internal combination.
Access modifiers are not allowed on namespaces. Namespaces have no access restrictions.
Depending on the context in which a member declaration occurs, only certain declared accessibilities are permitted. If no access modifier is specified in a member declaration, a default accessibility is used.
Top-level types, which are not nested in other types, can only have internal or public accessibility. The default accessibility for these types is internal.
Nested types, which are members of other types, can have declared accessibilities as indicated in the following table."

Members of	Default member accessibility	Allowed declared accessibility of the member
enum	**Public**	None
class	**Private**	**public** **protected** **internal** **private** **protected internal**
interface	**Public**	None
struct	**Private**	**public** **internal** **private**

Sealed Classes:

"Sealed" is a special class of access modifier in C#.If a class is marked as sealed, no other class can derive from that sealed class . In other words a class marked as sealed can't act as a base class to any other class.

Program:

```
namespace AccessModifiers
{
  sealed class AAA
  {

  }
  class BBB:AAA
  {

  }
  public class Program
  {
    public static void Main(string[] args)
    {
    }
  }
}
```

Compiler Output: 'AccessModifiers.BBB': cannot derive from sealed type 'AccessModifiers.AAA'
Hence proved.

Point to remember: A class marked sealed can't act as a base class to any other class.

Image credit: https://www.flickr.com/photos/lwr/931211869/

Access the members of sealed class,

Program:

```
using System;

namespace AccessModifiers
{
   sealed class AAA
   {
      public int x = 100;
      public void MethodA()
      {
         Console.WriteLine("Method A in sealed class");
      }

   }
   public class Program
   {
      public static void Main(string[] args)
      {
         AAA aaa=new AAA();
         Console.WriteLine(aaa.x);
         aaa.MethodA();
         Console.ReadKey();
      }
   }
}
```

Compiler Output:

```
100
Method A in sealed class
```

So, as we discussed, the only difference between a sealed and a non sealed class is that the sealed class cannot be derived from.A sealed class can contain variables, methods, properties like a normal class do.

Point to remember: Since we cannot derive from sealed classes, the code from the sealed classes can not be overridden.

Constants:

Lab1:

Our Program class in the console application,

Program:

```
  public class Program
```

```
{
   private const int x = 100;
   public static void Main(string[] args)
   {
      Console.WriteLine(x);
      Console.ReadKey();
   }
}
```

Output: 100

We see, a constant maked variable or a const variable behaves like a member variable in c#. We can provide it an initial value and can use it anywhere we want.

Point to remember: We need to initialize the const variable at the time we create it. We are not allowed to initialize it later in our code or program.

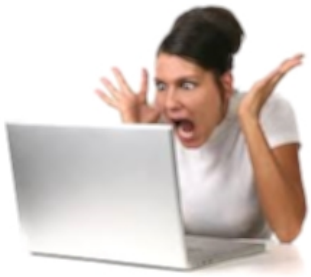
Lab2:

```
using System;

namespace AccessModifiers
{
   public class Program
   {
      private const int x = y + 100;
      private const int y = z - 10;
      private const int z = 300;

      public static void Main(string[] args)
      {
        System.Console.WriteLine("{0} {1} {2}",x,y,z);
         Console.ReadKey();
      }
   }
}
```

Can you guess the output? What ? Is it a compiler error?

Output:

390 290 300



Shocked? A constant field can no doubt depend upon another constant. C# is very smart to realize that to calculate the value of variable x marked const, it first needs to know the value of y variable. y's value depends upon another const variable z, whose value is set to 300. Thus C# first evaluates z to 300 then y becomes 290 i.e. z -1 and finally x takes on the value of y i.e. 290 + 100 resulting in 390.

Lab3:

Program:

```
using System;

namespace AccessModifiers
{
   public class Program
   {
      private const int x = y + 100;
      private const int y = z - 10;
      private const int z = x;

      public static void Main(string[] args)
      {
        System.Console.WriteLine("{0}{1}{2}",x,y,z);
         Console.ReadKey();
      }
   }
}
```

Output: The evaluation of the constant value for 'AccessModifiers.Program.x' involves a circular definition

We just assigned z=x from our previous code, and it resulted into error. The value of const x depends upon y , and y in turn depends upon value of z, but we see value z depends upon x as x it is assigned directly to z, it results in a circular dependency.

Point to remember: Like classes const variables cannot be circular i.e. they cannot depend on each other.

Lab4:

A const is a variable whose value once assigned cannot be modified, but its value is determined at compile time only.

```
using System;

namespace AccessModifiers
{
   public class Program
   {
      public const ClassA classA=new ClassA();
      public static void Main(string[] args)
      {
      }
   }

  public class ClassA
   {

   }
}
```

Output:

Compile time error: 'AccessModifiers.Program.classA' is of type 'AccessModifiers.ClassA'. A const field of a reference type other than string can only be initialized with null.

Point to remember: A const field of a reference type other than string can only be initialized with null.

If we assign the value to null in Program class,

```
using System;

namespace AccessModifiers
{
   public class Program
   {
      public const ClassA classA=null;
      public static void Main(string[] args)
      {
      }
   }

  public class ClassA
   {

   }
```

}

Then the error will vanish. The error disappears as we now initialize classA to an object which has a value that can be determined at compile time i.e null. We can never change the value of classA, so it will always be null. Normally we do not have consts as classA reference type as they have value only at runtime.

Point to remember: One can only initialize a const variable to a compile time value i.e. a value available to the compiler while it is executing.

new() actually gets executed at runtime and therefore does not get value at compile time. So this results in an error.

Lab5:

ClassA:

```
public class ClassA
  {
    public const int aaa = 10;
  }
```

Program:

```
public class Program
  {
    public static void Main(string[] args)
    {
      ClassA classA=new ClassA();
      Console.WriteLine(classA.aaa);
      Console.ReadKey();
    }
  }
```

Output:

Compile time error: Member 'AccessModifiers.ClassA.aaa' cannot be accessed with an instance reference; qualify it with a type name instead

Point to remember: A constant by default is static and we can't use the instance reference i.e. a name to reference a const. A const has to be static as no one will be allowed to make any changes to a const variable.

Just mark the const as static,

```
using System;
```

```
namespace AccessModifiers
{
   public class ClassA
   {
      public static const int aaa = 10;
   }

   public class Program
   {
      public static void Main(string[] args)
      {
         ClassA classA=new ClassA();
         Console.WriteLine(classA.aaa);
         Console.ReadKey();
      }
   }


}
```

Output:

Compile time error: The constant 'AccessModifiers.ClassA.aaa' cannot be marked static

C# tells us frankly that a field i.e. already static by default cannot be marked as static.

Point to remember: A const variable cannot be marked as static.

Lab6:

```
using System;

namespace AccessModifiers
{
   public class ClassA
   {
      public const int xxx = 10;
   }

   public class ClassB:ClassA
   {
      public const int xxx = 100;
   }

   public class Program
   {
```

```
        public static void Main(string[] args)
        {
            Console.WriteLine(ClassA.xxx);
            Console.WriteLine(ClassB.xxx);
            Console.ReadKey();
        }
    }


}
```

Output:

10
100
Compiler Warning: 'AccessModifiers.ClassB.xxx' hides inherited member 'AccessModifiers.ClassA.xxx'. Use the new keyword if hiding was intended.

We can always create a const with the same name in the derived class as another const in the base class. The const variable of class ClassB xxx will hide the const xxx in class ClassA for the class ClassB only.

Static Fields:

Point to remember: A variable in C# can never have an uninitialized value.

Let's discuss this in detail,

Lab1:

Program:

```
using System;

namespace AccessModifiers
{
    public class Program
    {
        private static int x;
        private static Boolean y;
        public static void Main(string[] args)
        {
            Console.WriteLine(x);
            Console.WriteLine(y);
```

```
        Console.ReadKey();
    }
  }


}
```

Output:

```
0
False
```

Point to remember: Static variables are always initialized when the class is loaded first. An int is given a default value of zero and a bool is given a default to False.

Lab2:

Program:

```
using System;

namespace AccessModifiers
{
  public class Program
  {
    private  int x;
    private  Boolean y;
    public static void Main(string[] args)
    {
      Program program=new Program();
      Console.WriteLine(program.x);
      Console.WriteLine(program.y);
      Console.ReadKey();
    }
  }


}
```

Output:

```
0
False
```

Point to remember: An instance variable is always initialized at the time of creation of its instance.

An instance variable is always initialized at the time of creation of its instance. The keyword new will create an instance of the class Program. It will allocate memory for each of the non static i.e. instance variables and then initialize each of them to their default values as well.

Lab3:

Program:

```
using System;

namespace AccessModifiers
{
   public class Program
   {
      private static int x = y + 10;
      private static int y = x + 5;
      public static void Main(string[] args)
      {
         Console.WriteLine(Program.x);
         Console.WriteLine(Program.y);
         Console.ReadKey();
      }
   }


}
```

Output:

```
10
15
```

Output is self explanatory. C# always initializes static variables to their initial value after creating them. Variables x and y are therefore given a default of zero value. C# now realizes that these variables declared need to be assigned some values. C# does not read all the lines at once but only one at a time. It will now read the first line and as the variable y has a value of 0, so x will get a value of 10. Then at the next line, y is the value of x + 5. The variable x has a value of 10 and so y now becomes 15. As C# does not see both lines at the same time, it does not notice the circularity of the above definition.

Lab4:

Program:

```
using System;
```

```
namespace AccessModifiers
{
   public class Program
   {
       int x = y + 10;
       int y = x + 5;
      public static void Main(string[] args)
      {

      }
   }


}
```

Output:
Compile time error:

A field initializer cannot reference the non-static field, method, or property 'AccessModifiers.Program.y'
A field initializer cannot reference the non-static field, method, or property 'AccessModifiers.Program.x'

The lab we did in Lab3 does not work for instance variables as the rules of an instance variable are quite different than that of static variables. The initializer of an instance variable has to be determined at the time of creation of the instance. The variable y does not have a value at this point in time. It can't refer to variables of the same object at the time of creation. So we can refer to no instance members to initialize an instance member.

Readonly Fields:

Readonly fields are one of the most interesting topics of OOP in c#.

Lab1:

Program:

```
using System;

namespace AccessModifiers
{
   public class Program
   {
      public static readonly int x = 100;

      public static void Main(string[] args)
```

```
    {
      Console.WriteLine(x);
      Console.ReadKey();
    }
  }


}
```

Output:

100

Wow, we get no error, but remember not to use a non static variable inside a static methos else we'll get an error.

Lab2:

Program:

```
using System;

namespace AccessModifiers
{
  public class Program
  {
    public static readonly int x = 100;

    public static void Main(string[] args)
    {
      x = 200;
      Console.WriteLine(x);
      Console.ReadKey();
    }
  }


}
```

Output:

Compile time error: **A static readonly field cannot be assigned to (except in a static constructor or a variable initializer).**

We cannot change the value of a readonly field except in a constructor.

Point to remember: A static readonly field cannot be assigned to (except in a static constructor or a variable initializer)

Lab3:

Program:

```
using System;

namespace AccessModifiers
{
   public class Program
   {
      public static readonly int x;

      public static void Main(string[] args)
      {
      }
   }
}
```

Here we find one difference between const and readonly, unlike const, readonly fields need not have to be initialized at the time of creation.

Lab4:

Program:

```
using System;

namespace AccessModifiers
{
   public class Program
   {
      public static readonly int x;

      static Program()
      {
         x = 100;
         Console.WriteLine("Inside Constructor");
      }

      public static void Main(string[] args)
      {
         Console.WriteLine(x);
         Console.ReadKey();
      }
```

```
    }
}
```

Output:

```
Inside Constructor
100
```

One more major difference between const and readonly is seen here. A static readonly variable can be initialized in the constructor as well , like we have seen in above mentioned example.

Lab5:

Program:

```
using System;

namespace AccessModifiers
{
    public class ClassA
    {

    }
    public class Program
    {

        public readonly ClassA classA=new ClassA();
        public static void Main(string[] args)
        {
        }
    }
}
```

We have already seen this example in const section. The same code gave an error with const does not give an error with readonly fields. So we can say that readonly is a more generic const and it makes our programs more readable as we refer to a name and not a number. Is 10 more intuitive or priceofcookie easier to understand? The compiler would for efficiency convert all const's and readonly fields to the actual values.

Lab6:

Program:

```
using System;

namespace AccessModifiers
{
    public class ClassA
```

```
    {
        public int readonly x= 100;
    }
    public class Program
    {
      public static void Main(string[] args)
        {
        }
    }
}
```

Output:

Compile time error:

Member modifier 'readonly' must precede the member type and name
Invalid token '=' in class, struct, or interface member declaration

Wherever we need to place multiple modifiers, remind yourself that there are rules that decide the order of access modifiers, which comes first. Now here the readonly modifier precedes the data type int, we already discussed in the very start of the article. This is just a rule that must always be remembered.

Lab7:

Program:

```
using System;

namespace AccessModifiers
{
    public class ClassA
    {
        public readonly int x= 100;

        void Method1(ref int y)
        {

        }

        void Method2()
        {
            Method1(ref x);
        }
    }
    public class Program
    {
```

```
        public static void Main(string[] args)
        {
        }
    }
}
```

Output:

Compile time error:

A readonly field cannot be passed ref or out (except in a constructor)

A readonly field can't be changed by anyone except a constructor. The method Method1 expects a ref parameter which if we have forgotten allows you to change the value of the original. Therefore C# does not permit a readonly as a parameter to a method that accepts a ref or an out parameters.

Summary:

Let's recall all the points that we have to remember,

1. The default access modifier is private for class members.
2. A class marked as internal can only be have its access limited to the current assembly only.
3. Namespaces as we see by default can have no accessibility specifiers at all. They are by default public and we cannot add any other access modifier including public again too.
4. A class can only be public or internal. It cannot be marked as protected or private. The default is internal for the class.
5. Members of a class can be marked with all the access modifiers, and the default access modifier is private.
6. Protected internal means that the derived class and the class within the same source code file can have access.
7. Between public and internal, public always allows greater access to its members.
8. Base class always allows more accessibility than the derived class.
9. The return values of a method must have greater accessibility than that of the method itself.
10. A class marked sealed can't act as a base class to any other class.
11. Since we cannot derive from sealed classes, the code from the sealed classes can not be overridden.
12. We need to initialize the const variable at the time we create it. We are not allowed to initialize it later in our code or program.
13. Like classes const variables cannot be circular i.e. they cannot depend on each other.
14. A const field of a reference type other than string can only be initialized with null.
15. One can only initialize a const variable to a compile time value i.e. a value available to the compiler while it is executing.
16. A constant by default is static and we can't use the instance reference i.e. a name to reference a const. A const has to be static as no one will be allowed to make any changes to a const variable.
17. A const variable cannot be marked as static.
18. A variable in C# can never have an uninitialized value.

19. Static variables are always initialized when the class is loaded first. An int is given a default value of zero and a bool is given a default to False.
20. An instance variable is always initialized at the time of creation of its instance.
21. A static readonly field cannot be assigned to (except in a static constructor or a variable initializer)

Conclusion:

With this article we completed almost all the scenarios of access modifiers. We did a lot of hands-on lab to clear our concepts. I hope my readers now know by heart about these basic concepts and will never forget them. In my upcoming article, i.e. the last article of this series, we'll be discussing about Properties and Indexers in C#.

Keep coding and enjoy reading 

Also do not forget to rate/comment/like my article if it helped you by any means, this helps me to get motivated and encourages me to write more and more.

Read more:

- Learning MVC series.
- 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 **A Practical Approach**.