Modifiers

.NET

Modifiers *are C# keywords used to modify declarations of types (class, struct, interface, enum) and type members (fields, properties, methods, indexers, etc).*

HTTPS://DOCS.MICROSOFT.COM/EN-US/DOTNET/CSHARP/LANGUAGE-REFERENCE/KEYWORDS/

Modifier vs. Access Modifier

Modifiers – Abstract

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract

abstract means the thing being modified has a missing or incomplete implementation.

- intended only to be a base class of other classes,
- NOT instantiated on their own.
- classes, methods, properties, indexers, and events can be ***abstract***
- Members marked as ***abstract*** must be implemented by non-***abstract*** classes that derive from the ***abstract*** class.

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Modifiers – Abstract

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract

Abstract CLASSES…

- cannot be instantiated.
- may contain ***abstract*** methods and accessors.
- must provide implementation for all implemented interface members.
- Cannot include the ***sealed*** modifier.

Abstract METHODS…

- An ***abstract*** method is implicitly a ***virtual*** method.
- are only permitted in ***abstract*** classes.
- do not have method body. ({})
- only have an implementation in a derived class methods using ***override*** keyword.
- Cannot have s***tatic*** or ***virtual*** modifiers

Abstract PROPERTIES…

- (All characteristics of methods are also true for properties.)
- Abstract properties are written with { get; set; } but still do NOT have an implementation

Modifiers – Abstract

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/abstract

DerivedClass is derived from the ***abstract*** class, BaseClass. The ***abstract*** class contains an ***abstract*** method, AbstractMethod(), and two ***abstract*** properties, X and Y.

An attempt to instantiate the ***abstract*** class by using this statement gets an error:

```
BaseClass bc = new BaseClass();   // Error
```

The compiler cannot create an instance of the ***abstract*** class 'BaseClass'.

```
abstract class BaseClass   // Abstract class
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod();   // Abstract method
    public abstract int X    { get; }
    public abstract int Y    { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
    {
        _x++;
        _y++;
    }

    public override int X   // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y   // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        var o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine($"x = {o.X}, y = {o.Y}");
    }
}
// Output: x = 111, y = 161
```

Modifiers – Virtual

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual

The ***virtual*** keyword is used to modify a method, property, indexer, or event declaration and allow for it to be ***overridden*** in a derived class.

The implementation of a ***virtual*** member can be changed by an overriding member in a derived class.

```
public virtual double Area()
{
    return x * y;
}
```

Modifiers – Virtual

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual

- By default, methods are non-***virtual***.
- You cannot ***override*** a non-***virtual*** method.
- You cannot use the ***virtual*** modifier with the ***static***, ***abstract***, ***private***, or ***override*** modifiers.
- A ***virtual*** inherited property can be overridden by using the ***override*** modifier.

```
class MyBaseClass
{
    // virtual auto-implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int num;
    public virtual int Number
    {
        get { return num; }
        set { num = value; }
    }
}

class MyDerivedClass : MyBaseClass
{
    private string name;

   // Override auto-implemented property with ordinary property
   // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }
}
```

Modifiers – Virtual

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/virtual

The Shape class contains the two coordinates x, y, and the Area() ***virtual*** method. Different shape classes such as Circle, Cylinder, and Sphere inherit the Shape class, and the surface area is calculated for each figure. Each derived class has its own ***override*** implementation of Area().

```
    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results.
        Console.WriteLine("Area of Circle   = {0:F2}", c.Area());
        Console.WriteLine("Area of Sphere   = {0:F2}", s.Area());
        Console.WriteLine("Area of Cylinder = {0:F2}", l.Area());
    }
}
/*
Output:
Area of Circle   = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
*/
```

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double x, y;

        public Shape()
        {
        }

        public Shape(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public virtual double Area()
        {
            return x * y;
        }
    }
```

```
public class Circle : Shape
{
    public Circle(double r) : base(r, 0)
    {
    }

    public override double Area()
    {
        return PI * x * x;
    }
}

class Sphere : Shape
{
    public Sphere(double r) : base(r, 0)
    {
    }

    public override double Area()
    {
        return 4 * PI * x * x;
    }
}

class Cylinder : Shape
{
    public Cylinder(double r, double h) : base(r, h)
    {
    }

    public override double Area()
    {
        return 2 * PI * x * x + 2 * PI * x * y;
    }
}
```

Modifiers – Sealed

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed

The ***sealed*** modifier prevents inheritance from a class.

The ***sealed*** modifier prevents an overriding method from being overridden by a more derived method.

In this example, Z inherits from Y but Z cannot ***override*** the ***virtual*** function F that is declared in X and ***sealed*** in Y.

```
class A {}
sealed class B : A {}
```

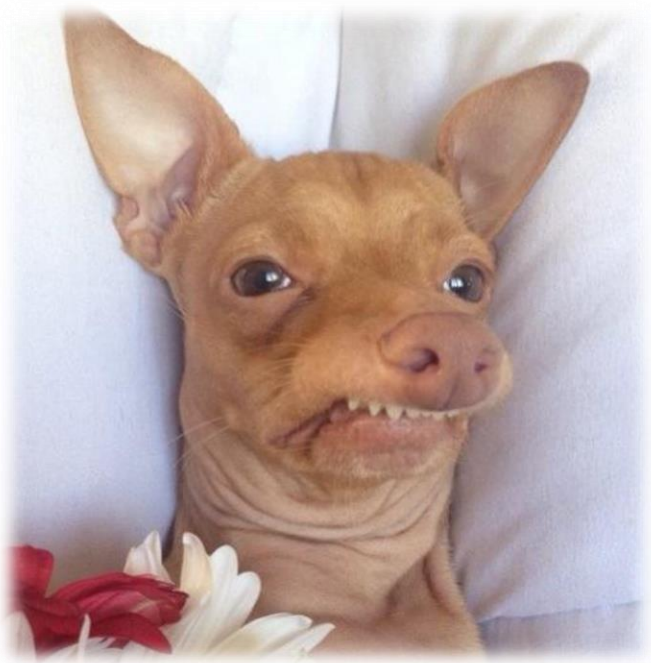
```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

Modifiers – Sealed

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed

When should you use *Sealed*? Consider…

- The potential benefits that deriving classes might gain through the ability to customize your class.
- The potential that deriving classes could modify your classes in such a way that they would no longer work correctly or as expected.



Modifiers – Static

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/static

Static CLASSES…

- Cannot be instantiated or extended.
- If a class is static, all It's members must be static.
- Essentially, just a container for static members.

ALL Static Members…

- Cannot use this to reference static methods or property accessors.
- Belongs to the class type itself rather than the specific object instance.
- A static member is referenced through the type name. (ex. Class.Struct.prop)

Modifiers – Const

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/const

- Const fields and locals aren't variables and may not be modified.
- Const fields can be numbers (int, double, long, etc), boolean, string, or null.
- The only ***reference*** types that can be const are string and a null reference.
- The static modifier is not allowed in a const declaration.

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

Modifiers – readonly

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/readonly

- The initialization can only occur as part of the declaration or in a constructor in the same class.
- Like const, but initialization can be deferred until it's constructor finishes.

Modifiers – Override

https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override

The ***override*** modifier is required to extend or modify the ***abstract*** or ***virtual*** implementation of an inherited method, property, indexer, or event.

ALL Override Members…

- Must provide a new implementation of an inherited method
- must have the same signature as the inherited method.
- Can only override a virtual, abstract, or override method.
- You cannot use the new, static, or virtual modifiers to modify an override method.

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    int side;

    public Square(int n) => side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => side * side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

Partial Classes, Structs, Interfaces

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/partial-classes-and-methods

---

You can split the definition of a ***class***, a ***struct***, an ***interface*** or a ***method*** over two or more source files. Each source file contains a section. All parts are combined on compilation.

When would you do this?

- When working with automatically generated source code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.
- Attributes, inherited classes, etc, are merged at compile-time.

```
public partial class Employee
{
    public void DoWork()
    {
    }
}

public partial class Employee
{
    public void GoToLunch()
    {
    }
}
```