CLASS IMPLEMENTATION AND HIERARCHIES

Static Class

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. For example, in the .NET Framework Class Library, the static System.Math class contains methods that perform mathematical operations.

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
```

Static constructor

A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed once only. It is called automatically before the first instance is created or any static members are referenced.

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

Inner Classes (Nested Classes)

In C# an inner (nested) class is called a class that is declared inside the body of another class. Accordingly, the class that encloses the inner class is called an outer class.
The main reason to declare one class into another are:

1. To better organize the code when working with objects in the real world, among which have a special relationship and one cannot exist without the other.

2. To hide a class in another class, so that the inner class cannot be used outside the class wrapped it.

```
public class OuterClass
{
        private string name;

        private OuterClass(string name)
        {
                this.name = name;
        }

        private class NestedClass
        {
                private string name;
                private OuterClass parent;

                public NestedClass(OuterClass parent, string name)
                {
                        this.parent = parent;
                        this.name = name;
                }

                public void PrintNames()
                {
                        Console.WriteLine("Nested name: " + this.name);
                        Console.WriteLine("Outer name: " + this.parent.name);
                }
        }

        static void Main()
        {
                OuterClass outerClass = new OuterClass("outer");
                NestedClass nestedClass = new
                        OuterClass.NestedClass(outerClass, "nested");
                nestedClass.PrintNames();
        }
}
```

Inheritance

Classes can inherit from another class.

```
public class A
{
    public A() { }
}

public class B : A
{
    public B() { }
}
```

A constructor can use the base keyword to call the constructor of a base class. For example:

```
public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

virtual and override

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. For example, this method can be overridden by any class that inherits it:

```
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 (value != String.Empty)
            {
                name = value;
            }
            else
            {
                name = "Unknown";
            }
        }
    }

}



class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

In the Main method, declare variables bc, dc, and bcdc.

- bc is of type BaseClass, and its value is of type BaseClass.
- dc is of type DerivedClass, and its value is of type DerivedClass.
- bcdc is of type BaseClass, and its value is of type DerivedClass. This is the variable to pay attention to.

Because bc and bcdc have type BaseClass, they can only directly access Method1, unless you use casting. Variable dc can access both Method1 and Method2. These relationships are shown in the following code.

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Next, add the following Method2 method to BaseClass. The signature of this method matches the signature of the Method2 method in DerivedClass.

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Because BaseClass now has a Method2 method, a second calling statement can be added for BaseClass variables bc and bcdc, as shown in the following code.

C#
```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

When you build the project, you see that the addition of the Method2 method in BaseClass causes a warning. The warning says that the Method2 method in DerivedClass hides the Method2 method in BaseClass. You are advised to use the new keyword in the Method2 definition if you intend to cause that result. Alternatively, you could rename one of the Method2 methods to resolve the warning, but that is not always practical.
Before adding new, run the program to see the output produced by the additional calling statements. The following results are displayed.

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

The new keyword preserves the relationships that produce that output, but it suppresses the warning. The variables that have type BaseClass continue to access the members of BaseClass, and the variable that has type DerivedClass continues to access members in DerivedClass first, and then to consider members inherited from BaseClass.
To suppress the warning, add the new modifier to the definition of Method2 in DerivedClass, as shown in the following code. The modifier can be added before or after public.

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Run the program again to verify that the output has not changed. Also verify that the warning no longer appears. By using new, you are asserting that you are aware that the member that it modifies hides a member that is inherited from the base class. For more information about name hiding through inheritance.
To contrast this behavior to the effects of using override, add the following method to DerivedClass. The override modifier can be added before or after public.

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Add the virtual modifier to the definition of Method1 in BaseClass. The virtual modifier can be added before or after public.

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Run the project again. Notice especially the last two lines of the following output.

C#
```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

The use of the override modifier enables bcdc to access the Method1 method that is defined in DerivedClass. Typically, that is the desired behavior in inheritance hierarchies. You want objects that have values that are created from the derived class to use the methods that are defined in the derived class. You achieve that behavior by using override to extend the base class method.

Assemblies

Assemblies take the form of an executable (.exe) file or dynamic link library (.dll) file, and are the building blocks of the .NET Framework. They provide the common language runtime with the information it needs to be aware of type implementations. You can think of an assembly as a collection of types and resources that form a logical unit of functionality and are built to work together.

Access modifier

The access modifiers, public, protected, internal, or private, is used to specify one of the following declared accessibility levels for members.

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.

The internal keyword is an access modifier for types and type members. Internal types or members are accessible only within files in the same assembly, as in this example:

```
public class BaseClass
{
    // Only accessible within the same assembly
    internal static int x = 0;
}
```

Types or members that have access modifier protected internal can be accessed from the current assembly or from types that are derived from the containing class.

A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code. For example, a framework for building graphical user interfaces could provide Control and Form classes that cooperate by using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

It is an error to reference a type or a member with internal access outside the assembly within which it was defined.

This example contains two files, Assembly1.cs and Assembly1_a.cs. The first file contains an internal base class, BaseClass. In the second file, an attempt to instantiate BaseClass will produce an error.

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
   public static int intM = 0;
}


// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
   static void Main()
   {
      BaseClass myBase = new BaseClass();   // CS0122
   }
}
```

In the following example, use the same files you used in example 1, and change the accessibility level of BaseClass to public. Also change the accessibility level of the member IntM to internal. In this case, you can instantiate the class, but you cannot access the internal member.

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
   internal static int intM = 0;
}


// Assembly2_a.cs
// Compile with: /reference:Assembly1.dll
public class TestAccess
{
   static void Main()
   {
      BaseClass myBase = new BaseClass();   // Ok.
      BaseClass.intM = 444;    // CS0117
   }
}
```

Partial Classes

There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time.
- When working with automatically generated source, code can be added to the class without having to recreate the source file.
- To split a class definition, use the partial keyword modifier, as shown here:

In the following example, the fields and the constructor of the class, CoOrds, are declared in one partial class definition, and the member, PrintCoOrds, is declared in another partial class definition.

```
public partial class CoOrds
{
    private int x;
    private int y;

    public CoOrds(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

public partial class CoOrds
{
    public void PrintCoOrds()
    {
        Console.WriteLine("CoOrds: {0},{1}", x, y);
    }
}
class TestCoOrds
{
    static void Main()
    {
        CoOrds myCoOrds = new CoOrds(10, 15);
        myCoOrds.PrintCoOrds();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Partial Methods

A partial class may contain a partial method. One part of the class contains the signature of the method. An optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

Partial methods are especially useful as a way to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method.

A partial method declaration consists of two parts: the definition, and the implementation. These may be in separate parts of a partial class, or in the same part. If there is no implementation declaration, then the compiler optimizes away both the defining declaration and all calls to the method.

```
// Definition in file1.cs
partial void onNameChanged();

// Implementation in file2.cs
partial void onNameChanged()
{
  // method body
}
```

Abstract class

The abstract modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes. Members marked as abstract, or included in an abstract class, must be implemented by classes that derive from the abstract class.

```
abstract class ShapesClass
{
    abstract public int Area();
}
class Square : ShapesClass
{
    int side = 0;

    public Square(int n)
    {
        side = n;
    }
    // Area method is required to avoid
    // a compile-time error.
    public override int Area()
    {
        return side * side;
    }

    static void Main()
    {
        Square sq = new Square(12);
        Console.WriteLine("Area of the square = {0}", sq.Area());
    }

// Output: Area of the square = 144
```

Abstract classes have the following features:

- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- It is not possible to modify an abstract class with the sealed modifier because the two modifers have opposite meanings. The sealed modifier prevents a class from being inherited and the abstract modifier requires a class to be inherited.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Use the abstract modifier in a method or property declaration to indicate that the method or property does not contain implementation.

- Abstract methods have the following features:
- An abstract method is implicitly a virtual method.
- Abstract method declarations are only permitted in abstract classes.

Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. For example:

```
public abstract void MyMethod();
```

- The implementation is provided by an overriding methodoverride (C# Reference), which is a member of a non-abstract class.
- It is an error to use the static or virtual modifiers in an abstract method declaration.
- Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.
- It is an error to use the abstract modifier on a static property.
- An abstract inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.
- An abstract class must provide implementation for all interface members.
- An abstract class that implements an interface might map the interface methods onto abstract methods. For example:

In this example, the class DerivedClass is derived from an abstract class BaseClass. The abstract class contains an abstract method, AbstractMethod, and two abstract properties, X and Y.

```
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()
    {
        DerivedClass o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine("x = {0}, y = {1}", o.X, o.Y);
    }
}
// Output: x = 111, y = 161
```

In the preceding example, if you attempt to instantiate the abstract class by using a statement like this:

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

You will get an error saying that the compiler cannot create an instance of the abstract class 'BaseClass'.

When an abstract class inherits a virtual method from a base class, the abstract class can override the virtual method with an abstract method. For example:

```
// compile with: /target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }

}
```

A class inheriting an abstract method cannot access the original implementation of the method—in the previous example, DoWork on class F cannot call DoWork on class D. In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

Sealed Classes and Methods

The sealed modifier is used to prevent derivation from a class. An error occurs if a sealed class is specified as the base class of another class. A sealed class cannot also be an abstract class.

The sealed modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations.

In C# structs are implicitly sealed; therefore, they cannot be inherited.

```
using System;
sealed class MyClass
{
public int x;
public int y;
}

class MainClass
{
public static void Main()
{
MyClass mC = new MyClass();
mC.x = 110;
mC.y = 150;
Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
}
}
```

In the preceding example, if you attempt to inherit from the sealed class by using a statement like this:

```
class MyDerivedC: MyClass {} // Error
```

You will get the error message:
'MyDerivedC' cannot inherit from sealed class 'MyBaseC'.

In C# a method can't be declared as sealed. However when we override a method in a derived class, we can declare the overrided method as sealed as shown below. By declaring it as sealed, we can avoid further overriding of this method.

```
using System;
class MyClass1
```

```
{
public int x;
public int y;
public virtual void Method()
{
Console.WriteLine("virtual method");
}
}
class MyClass : MyClass1
{
public override sealed void Method()
{
Console.WriteLine("sealed method");
}
}
class MainClass
{
public static void Main()
{
MyClass1 mC = new MyClass();
mC.x = 110;
mC.y = 150;
Console.WriteLine("x = {0}, y = {1}", mC.x, mC.y);
mC.Method();
}
}
```

Extension Method

Extension methods are used to "extend" functionality of an existing class. Extension method is defined separately from the class it is extending.
The method is defined separately from the class it is extending. It should be defined as static and be in non-generic static class that is not nested.
Extension method should have an "instance parameter" defined. This is the first parameter in Extension method and that is preceded by this modifier. The instance parameter cannot be a pointer type and cannot have parameter modifiers (ref, out, etc.)

Example

Lets assume you have a class Cat:

```
using System;

namespace  ExtensionMethodsSamples
{
   public class Cat
   {
        public void Eat(object food)
        {
            //TODO: eat implementation
        }

        public void Jump()
        {
            //TODO: jump implementation
        }

    }
}
```

After some time you want to extend it with Dance method. To create is as extension method you declare it in a separate class as a static method in a following way:

```
using System;
using ExtensionMethodsSamples;

namespace CatExtension
{
    public static class CatExtension
    {
        public static void Dance(this Cat cat)
        {
            for (int i=0; i<10; i++)
                cat.Jump(); // dumb implementation, forgive me, cat
        }
    }
}
```

Note this keyword in parameter declaration. Also note that method is static and that it is defined in namespace CatExtension(name doesn't matter, just note that it is different from the one where Cat is defined).

Now you should import namespace where extension method is defined and cat gains ability to "dance". You can call now Dance method of the Cat class just like the usual instance method.

```
using System;
using CatExtension;

namespace ExtensionMethodsSamples
{
    class Program
    {
        static void Main(string[] args)
        {
            Cat c = new Cat();
            c.Dance();

        }
    }
}
```

So, when and how to use Extension Methods?
The possible right uses of extension methods:
To provide extension to a library that cannot be extended in other way and on which you do not have access over.