Contents

[1. What is the basic class we use in .Net. 2](#_Toc446119293)

[2. What is the class inherited used by code behind file. 3](#_Toc446119294)

[3. Constructor Chaning 3](#_Toc446119295)

[4. Constructor behaviour in inheritance 3](#_Toc446119296)

[5. Static Constructors 5](#_Toc446119297)

[6. FAQ on Constructors 7](#_Toc446119298)

[7. Object declaration in Inheritance 8](#_Toc446119299)

[8. The is and as keywords 9](#_Toc446119300)

[9. Boxing and Unboxing 10](#_Toc446119301)

[10. Structures 10](#_Toc446119302)

[11. Ref and Out Keyword 10](#_Toc446119303)

[12. Advantages of a Partial Class 11](#_Toc446119304)

[13. C# - Constructors in C# with Example, Types of Constructor in C# with Example 12](#_Toc446119305)

[14. Const, Read only and Static Variable : 20](#_Toc446119306)

[const vs. readonly 20](#_Toc446119307)

[static 22](#_Toc446119308)

[15. ILDASM and Constants 23](#_Toc446119309)

[16. All about abstract classes. 23](#_Toc446119310)

# What is the basic class we use in .Net.

Ans – System.Objects

In C# (and the .NET framework) all the types (classes, structures and interfaces) are implicitly inherited from the Object class defined in the System namespace. This class provides the low-level services and general functionality to each and every class in the .NET framework. The Object class is extremely useful when it comes to polymorphism (which we are about to see), as a reference of type Object can hold any type of object. The Object class has following methods:

|  |  |
| --- | --- |
| Method Name | Description |
| Equals(Object) | Compares two objects for equality. The default implementation only supports reference equality, that is, it will return true if both references point to the same object. For value types, bitwise checking is performed. Derived classes should override this method to define  equality for their objects. |
| Static Equals(Object, Object) | Same as above except, this method is static. |
| GetHashCode() | Returns the hash code for the current object. |
| GetType() | Returns the Type object that represents the exact run-time type of the  current object. |
| static ReferenceEquals (Object,Object) | Returns true if both the references passed point to the same object, otherwise returns false. |
| ToString() | Returns the string representation of the object. Derived classes should override this method to provide the string representation of the current  object. |
| protected Finalize() | This protected method should be overridden by the derived classes to free any resources. This method is called by CLR before the current  object is reclaimed by Garbage Collector. |
| protected MemberwiseClone() | Provides a shallow copy of the current object. A shallow copy contains a copy of all the instance fields (state) of the current objects. |

C# also provides an object keyword which maps to the System.Object class in the .NET framework class library (FCL).

# What is the class inherited used by code behind file.

Ans- System.web.UI.page

This class can not be instanciated or reused.

# Constructor Chaning

class Student {

string \_studentType = "";

string \_id = "";

string \_fName = "";

string \_lName = "";

public Student(string id)

: this(id, "", "") {

}

public Student(string id, string fName)

: this(id, fName, "") {

}

public Student(string id, string fName, string lName) {

*//Validate logic.....*

\_studentType = "<student\_type>";

\_id = id;

\_fName = fName;

\_lName = lName;

}

}

# Constructor behaviour in inheritance

Let us first create the inherited class.

public class myBaseClass

{

public myBaseClass()

{

// Code for First Base class Constructor

}

public myBaseClass(int Age)

{

// Code for Second Base class Constructor

}

// Other class members goes here

}

public class myDerivedClass : myBaseClass

// Note that I am inheriting the class here.

{

public myDerivedClass()

{

// Code for the First myDerivedClass Constructor.

}

public myDerivedClass(int Age):base(Age)

{

// Code for the Second myDerivedClass Constructor.

}

// Other class members goes here

}

Now what will be the execution sequence here :

If I create the object of the Derived class as

myDerivedClass obj = new myDerivedClass()

Then the sequence of execution will be

1. public myBaseClass() method

2. and then public myDerivedClass() method.

Note: If we do not provide initializer referring to the base class constructor then it executes the no parameter constructor of the base class.

Note one thing here : We are not making any explicit call to the constructor of base class neither by initializer nor by the base() keyword, but it is still executing. This is the normal behavior of the constructor.

If I create the object of the Derived class as

myDerivedClass obj = new myDerivedClass(15)

Then the sequence of execution will be

1. public myBaseClass(int Age) method.

2. and then public myDerivedClass(int Age) method.

Here the new keyword base has come into picture. This refers to the base class of the current class. So, here it refers to the myBaseClass. And base(10) refers to the call to myBaseClass(int Age) method.

Also note the usage of Age variable in the syntax :

public myDerivedClass(int Age):base(Age).

Understanding it is left to the reader.

# ****Static Constructors****

This is a new concept introduced in C#. By new here I mean that it was not available for the C++ developers. This is a special constructor and gets called before the first object is created of the class. The time of execution cannot be determined, but it is definitely before the first object creation - could be at the time of loading the assembly.

The syntax of writing the static constructors is also damn simple. Here it is :

public class myClass

{

static myClass()

{

// Initialization code goes here.

// Can only access static members here.

}

// Other class methods goes here

}

Notes for Static Constructors :

1. There can be only one static constructor in the class.

2. The static constructor should be without parameters.

3. It can only access the static members of the class.

4. There should be no access modifier in static constructor definition.

Ok fine, all the above points are fine but why is it like that? Let us go step by step here.

Firstly, the call to the static method is made by the CLR and not by the object, so we do not need to have the access modifier to it.

Secondly, it is going to be called by CLR, who can pass the parameters to it, if required, No one, so we cannot have parameterized static constructor.

Thirdly, Non-static members in the class are specific to the object instance so static constructor, if allowed to work on non-static members, will reflect the changes in all the object instances, which is impractical. So static constructor can access only static members of the class.

Fourthly, Overloading needs the two methods to be different in terms to methods definition, which you cannot do with Static Constructors, so you can have at the most one static constructor in the class.

Now, one question raises here, can we have the two constructors as

public class myClass

{

static myClass()

{

// Initialization code goes here.

// Can only access static members here.

}

public myClass()

{

// Code for the First myDerivedClass Constructor.

}

// Other class methods goes here

}

This is perfectly valid, though doesn't seem to be in accordance with overloading concepts. But why? Because the time of execution of the two method are different. One is at the time of loading the assembly and one is at the time of object creation.

# FAQ on Constructors

1. Is the Constructor mandatory for the class ?

Yes, It is mandatory to have the constructor in the class and that too should be accessible for the object i.e., it should have a proper access modifier. Say for example we have the private constructor in the class then it is of no use as it cannot be accessed by the object, so practically it is no available for the object. In such conditions it will raise an error.

2. What if I do not write the constructor ?

In such case the compiler will try to supply the no parameter constructor for your class behind the scene. Compiler will attempt this only if you do not write the constructor for the class. If you provide any constructor ( with or without parameters), then compiler will not make any such attempt.

3. What if I have the constructor public myDerivedClass() but not the public myBaseClass() ?

It will raise an error. If either the no parameter constructor is absent or it is in-accessible ( say it is private ), it will raise an error. You will have to take the precaution here.

4. Can we access static members from the non-static ( normal ) constructors ?

Yes, We can. There is no such restriction on non-static constructors. But there is one on static constructors that it can access only static members.

# Object declaration in Inheritance

**Using the reference of the base type for referencing the objects of child types**

class A

{

public void MethodA()

{

...

}

}

class B : A

{

public void MethodB()

{

...

}

}

Then it is legal to write:

A a = new B();

Here a reference of type A is holding an object of type B. In this case we are treating the object of type B as an object of type A (which is quite possible as B is a sub-type of A). Now, it is possible to write:

a.MethodA();

But it is incorrect to write:

a.MethodB(); // error

Although we have an object of type B (contained in the reference of type A), we can not access any member of type B since the apparent type here is A and not B.

But if we mark the method in base class as virtual and we are overriding the method in child class then

A a = new B();

And a.MethodA will call MethodA which is overridden in the child class.

**However if we mark the methodA virtal in the base class and implemented methodA with the new keyword then**

A a = new B(); will call MethodA of base class.

If the methodA is marked as virtual then you have either mark the method as new or override in the child class.

Parent theParent = new Child(); --- called Up casting , its safe

Child theChild = new Parent(); --- called Down Casting which is unsafe

# The is and as keywords

To check the run-time type of an object, you can use either the is or the as keyword. is compares the type of the object with the given type and returns true if it is cast-able; otherwise, it returns false. For example,

Console.WriteLine(shapes[1] is Rectangle);

would print true on the Console Window, otherwise false, its best to use while downcasting

Alternatively, we can also use the as operator to check the run-time type of the object. The as operator returns null if the object is not cast-able otherwise, it casts the object to the specified type as

Shape [] shapes = { new Circle(), new Rectangle(), new Curve() };

Rectangle rect = shapes[1] as Rectangle;

if(rect != null)

Console.WriteLine("Cast successful");

else

Console.WriteLine("Cast unsuccessful");

# Boxing and Unboxing

Value types are stored on the stack and objects are stored in the heap. Boxing takes a copy of the value type from the stack to the heap while un-boxing takes the value type back to the stack. On the other hand, casting does not physically move or operate on the object. Casting merely changes the way objects are treated in a program by altering their reference type.

# Structures

* A struct is useful for creating types that are used to hold data like Point, Rectangle, Color types.
* A struct is of value type, contrary to classes which are of reference type. This means that structures are allocated on the stack and passed to methods by value, that is, by making their copies.
* A struct may contain constructors (except for the no-argument constructor), fields, methods and properties just like in classes.
* Like all value types, structs can neither inherit another class, nor can they be inherited. A struct can implement interfaces.
* Like every other type in C#, a struct is also implicitly inherited from the System.Object class. Instances of a struct can be created with and without using the new keyword.

# Ref and Out Keyword

* While we pass the variable value to the method the change in value of it inside the method is in scope of the method.

But if we want the variable value to be affected even inside the method and out of the method then we can use the ref and out keyword, Actually the reference of the variable is passed to the method that is why the changes resides,

In the case of ref keyword the variable value should be declared however for out keyword we can assign value inside the method as well. That is why out is used for output purpose.

* int a = 3;

DoWork(ref a);

Console.WriteLine("The value of a is " + a);

public static void DoWork(ref int i) // note ref

{

i++;

}

The value of a is 4

* int a;

DoWork(out a);

Console.WriteLine("The value of a is " + a);

public static void DoWork(out int i) // note out

{

i=4;

}

The value of a is 4

# Advantages of a Partial Class

Here is a list of some of the advantages of partial classes:

1. You can separate UI design code and business logic code so that it is easy to read and understand. For example, you are developing a web application using Visual Studio and add a new web form then there are two source files, "*aspx.cs*" and "*aspx.designer.cs*". These two files have the same class with the partialkeyword. The "*.aspx.cs*" class has the business logic code while "*aspx.designer.cs*" has user interface control definition.
2. When working with automatically generated source, the code can be added to the class without having to recreate the source file. For example, you are working with LINQ to SQL and create a DBML file. Now when you drag and drop a table, it creates a partial class in *designer.cs* and all table columns have properties in the class. You need more columns in this table to bind on the UI grid but you don't want to add a new column to the database table so you can create a separate source file for this class that has a new property for that column and it will be a partial class. So that does affect the mapping between database table and DBML entity but you can easily get an extra field. It means you can write the code on your own without messing with the system generated code.
3. More than one developer can simultaneously write the code for the class.
4. You can maintain your application better by compacting large classes. Suppose you have a class that has multiple interfaces so you can create multiple source files depending on interface implements. It is easy to understand and maintain an interface implemented on which the source file has a partial class. Let's see the following code snippet.

Hide   Copy Code

public interface IRegister

{

*//Register related function*

}

public interface ILogin

{

*//Login related function*

}

*//UserRegister.cs file*

public partial classUser : IRegister, ILogin

{

*//implements IRegister interface*

}

*//UserLogin.cs file*

public partial classUser

{

*//implements ILogin interface*

}

# [C# - Constructors in C# with Example, Types of Constructor in C# with Example](http://www.aspdotnet-suresh.com/2013/09/csharp-constructor-example-types-of-constructor-in-csharp.html)

Constructor is a special method of a class which will invoke automatically whenever instance or object of class is created. Constructors are responsible for object initialization and memory allocation of its class. If we create any class without constructor, the compiler will automatically create one default constructor for that class. There is always at least one constructor in every class.

Here you need to remember that a class can have any number of constructors and constructors don’t have any return type, not even void and within a class we can create only one static constructor.

Generally constructor name should be same as class name. If we want to create constructor in a class we need to create a constructor method name same as class name check below sample method for constructor

|  |
| --- |
| class SampleA  {  public SampleA()  {  Console.WriteLine("Sample A Test Method");  }  } |

**Types of Constructors**

Basically constructors are 5 types those are

Default Constructor

Parameterized Constructor

Copy Constructor

Static Constructor

Private Constructor

**Default Constructor**

A constructor without having any parameters called default constructor. In this constructor every instance of the class will be initialized without any parameter values like as shown below

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  class Sample  {  public string param1, param2;  public Sample() // Default Constructor  {  param1 = "Welcome";  param2 = "Aspdotnet-Suresh";  }  }  class Program  {  static void Main(string[] args)  {  Sample obj=new Sample(); // Once object of class created automatically constructor will be called  Console.WriteLine(obj.param1);  Console.WriteLine(obj.param2);  Console.ReadLine();  }  }  } |

When we run above program it will show output like as shown below

**Output**

|  |
| --- |
| Welcome  Aspdotnet-Suresh |

**Parameterized Constructors**

A constructor with at least one parameter is called as parameterized constructor. In parameterized constructor we can initialize each instance of the class to different values like as shown below

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  class Sample  {  public string param1, param2;  public Sample(string x, string y) // Declaring Parameterized constructor with Parameters  {  param1 = x;  param2 = y;  }  }  class Program  {  static void Main(string[] args)  {  Sample obj=new Sample("Welcome","Aspdotnet-Suresh"); // Parameterized Constructor Called  Console.WriteLine(obj.param1 +" to "+ obj.param2);  Console.ReadLine();  }  }  } |

When we run above program it will show output like as shown below

**Output**

|  |
| --- |
| Welcome to Aspdotnet-Suresh |

**Constructor Overloading**

In c# we can overload constructor by creating another constructor with same method name and different parameters like as shown below

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  class Sample  {  public string param1, param2;  public Sample() // Default Constructor  {  param1 = "Hi";  param2 = "I am Default Constructor";  }  public Sample(string x, string y) // Declaring Parameterized constructor with Parameters  {  param1 = x;  param2 = y;  }  }  class Program  {  static void Main(string[] args)  {  Sample obj = new Sample(); // Default Constructor will Called  Sample obj1=new Sample("Welcome","Aspdotnet-Suresh"); // Parameterized Constructor will Called  Console.WriteLine(obj.param1 + ", "+obj.param2);  Console.WriteLine(obj1.param1 +" to " + obj1.param2);  Console.ReadLine();  }  } |

When we run above program it will show output like as shown below

**Output**

|  |
| --- |
| Hi, I am Default Constructor  Welcome to Aspdotnet-Suresh |

**Copy Constructor**

A parameterized constructor that contains a parameter of same class type is called as copy constructor. Main purpose of copy constructor is to initialize new instance to the values of an existing instance. Check below example for this

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  class Sample  {  public string param1, param2;  public Sample(string x, string y)  {  param1 = x;  param2 = y;  }  public Sample(Sample obj) // Copy Constructor  {  param1 = obj.param1;  param2 = obj.param2;  }  }  class Program  {  static void Main(string[] args)  {  Sample obj = new Sample("Welcome", "Aspdotnet-Suresh"); // Create instance to class Sample  Sample obj1=new Sample(obj); // Here obj details will copied to obj1  Console.WriteLine(obj1.param1 +" to " + obj1.param2);  Console.ReadLine();  }  }  } |

When we run above program it will show output like as shown below

**Output**

|  |
| --- |
| Welcome to Aspdotnet-Suresh |

**Static Constructor**

When we declared constructor as static it will be invoked only once for any number of instances of the class and it’s during the creation of first instance of the class or the first reference to a static member in the class. Static constructor is used to initialize static fields of the class and to write the code that needs to be executed only once.

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  class Sample  {  public string param1, param2;  static Sample()  {  Console.WriteLine("Static Constructor");  }  public Sample()  {  param1 = "Sample";  param2 = "Instance Constructor";  }  }  class Program  {  static void Main(string[] args)  {  // Here Both Static and instance constructors are invoked for first instance  Sample obj=new Sample();  Console.WriteLine(obj.param1 + " " + obj.param2);  // Here only instance constructor will be invoked  Sample obj1 = new Sample();  Console.WriteLine(obj1.param1 +" " + obj1.param2);  Console.ReadLine();  }  }  } |

When we run above program we will get output like as shown below

**Output**

|  |
| --- |
| Static Constructor  Sample Instance Constructor  Sample Instance Constructor |

**Importance points of static constructor**

- Static constructor will not accept any parameters because it is automatically called by CLR.

- Static constructor will not have any access modifiers.

- Static constructor will execute automatically whenever we create first instance of class

- Only one static constructor will allowed.

**Private Constructor**

Private constructor is a special instance constructor used in a class that contains static member only. If a class has one or more private constructor and no public constructor then other classes is not allowed to create instance of this class this mean we can neither create the object of the class nor it can be inherit by other class. The main purpose of creating private constructor is used to restrict the class from being instantiated when it contains every member as static.

|  |
| --- |
| using System;  namespace ConsoleApplication3  {  public class Sample  {  public string param1, param2;  public Sample(string a,string b)  {  param1 = a;  param2 = b;  }  private Sample() // Private Constructor Declaration  {  Console.WriteLine("Private Constructor with no prameters");  }  }  class Program  {  static void Main(string[] args)  {  // Here we don't have chance to create instace for private constructor  Sample obj = new Sample("Welcome","to Aspdotnet-Suresh");  Console.WriteLine(obj.param1 +" " + obj.param2);  Console.ReadLine();  }  }  } |

**Output**

|  |
| --- |
| Welcome to Aspdotnet-Suresh |

In above method we can create object of class with parameters will work fine. If create object of class without parameters it will not allow us create.

|  |
| --- |
| // it will works fine  Sample obj = new Sample("Welcome","to Aspdotnet-Suresh");  // it will not work because of inaccessability  Sample obj=new Sample(); |

**Important points of private constructor**

One use of private construct is when we have only static member.

Once we provide a constructor that is either private or public or any, the compiler will not allow us to add public constructor without parameters to the class.

If we want to create object of class even if we have private constructors then we need to have public constructor along with private constructor

# Const, Read only and Static Variable :

Jump to: [navigation](http://tutorials.csharp-online.net/const,_static_and_readonly#column-one), [search](http://tutorials.csharp-online.net/const,_static_and_readonly#searchInput)

Within a class, const, static and readonly members are special in comparison to the other modifiers.

const vs. readonly

const and readonly perform a similar function on data members, but they have a few important differences.

const

A constant member is defined at compile time and cannot be changed at runtime. Constants are declared as a field, using the const keyword and must be initialized as they are declared. For example;

public class MyClass

{

public const double PI = 3.14159;

}

* PI cannot be changed in the application anywhere else in the code as this will cause a compiler error.
* Constants must be a value type (sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, or bool), an enumeration, a string literal, or a reference to null.
* Since classes or structures are initialized at run time with the new keyword, and not at compile time, you can't set a constant to a class or structure.
* Constants can be marked as public, private, protected, internal, or protected internal.
* Constants are accessed as if they were static fields, although they cannot use the static keyword.
* To use a constant outside of the class that it is declared in, you must fully qualify it using the class name.

readonly

A read only member is like a constant in that it represents an unchanging value. The difference is that a readonly member can be initialized at runtime, in a constructor as well being able to be initialized as they are declared. For example:

public class MyClass

{

public readonly double PI = 3.14159;

}

or

public class MyClass

{

public readonly double PI;

public MyClass()

{

PI = 3.14159;

}

}

* Because a readonly field can be initialized either at the declaration or in a constructor, readonly fields can have different values depending on the constructor used. A readonly field can also be used for runtime constants as in the following example:

public static readonly uint l1 = (uint)DateTime.Now.Ticks;

**Notes**

* readonly members are not implicitly static, and therefore the static keyword can be applied to a readonly field explicitly if required.
* A readonly member can hold a complex object by using the new keyword at initialization.

static

* Use of the static modifier to declare a static member, means that the member is no longer tied to a specific object. This means that the member can be accessed without creating an instance of the class. Only one copy of static fields and events exists, and static methods and properties can only access static fields and static events. For example:

public class Car

{

public static int NumberOfWheels = 4;

}

The static modifier can be used with classes, fields, methods, properties, operators, events and constructors, but cannot be used with indexers, destructors, or types other than classes.

static members are initialized before the static member is accessed for the first time, and before the static constructor, if any is called. To access a static class member, use the name of the class instead of a variable name to specify the location of the member. For example:

int i = Car.NumberOfWheels;

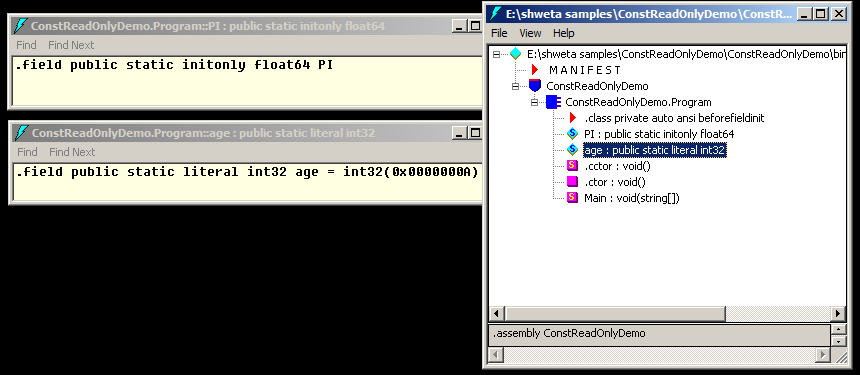
#### When to use what?

If the value is going to fix throughout the program and is never going to change in any circumstances, then one should choose const.

But on the other hand, if assignment of initial value depends on some parameter/conditions and value needs to be decided at run-time, then one can opt for readonly and based on that, initial value of a readonly variable can be set. But please note, once the value is assigned, further modification is not at all possible till the lifetime of the application.

# ILDASM and Constants

Now, let's jump quickly on ildasm to prove the value assignment for both of these.



As I mentioned earlier, const are compile time constants and are assigned at the time of declaration itself. So, the same can be proved via ildasm using IL code. In ildasm, one can see the value of const variable in hexa but for readonly variable, there is no such value assigned in PI variable in ildasm.

Hope the above tip was useful.

# All about abstract classes.

abstract class absClass

{

}

An example of an abstract method:

abstract class absClass

{

public abstract void abstractMethod();

}

abstract class absClass

{

public void NonAbstractMethod()

{

Console.WriteLine("NonAbstract Method");

}

}

A sample program that explains abstract classes:

using System;

namespace abstractSample

{

*//Creating an Abstract Class*

abstract class absClass

{

*//A Non abstract method*

public int AddTwoNumbers(int Num1, int Num2)

{

return Num1 + Num2;

}

*//An abstract method, to be*

*//overridden in derived class*

public abstract int MultiplyTwoNumbers(int Num1, int Num2);

}

*//A Child Class of absClass*

class absDerived:absClass

{

[STAThread]

static void Main(string[] args)

{

*//You can create an*

*//instance of the derived class*

absDerived calculate = new absDerived();

int added = calculate.AddTwoNumbers(10,20);

int multiplied = calculate.MultiplyTwoNumbers(10,20);

Console.WriteLine("Added : {0},

Multiplied : {1}", added, multiplied);

}

*//using override keyword,*

*//implementing the abstract method*

*//MultiplyTwoNumbers*

public override int MultiplyTwoNumbers(int Num1, int Num2)

{

return Num1 \* Num2;

}

}

}

*/Abstract Class1*

abstract class absClass1

{

public abstract int AddTwoNumbers(int Num1, int Num2);

public abstract int MultiplyTwoNumbers(int Num1, int Num2);

}

*//Abstract Class2*

abstract class absClass2:absClass1

{

*//Implementing AddTwoNumbers*

public override int AddTwoNumbers(int Num1, int Num2)

{

return Num1+Num2;

}

}

*//Derived class from absClass2*

class absDerived:absClass2

{

*//Implementing MultiplyTwoNumbers*

public override int MultiplyTwoNumbers(int Num1, int Num2)

{

return Num1\*Num2;

}

}

Prepoperties:

*/Abstract Class with abstract properties*

abstract class absClass

{

protected int myNumber;

public abstract int numbers

{

get;

set;

}

}

class absDerived:absClass

{

*//Implementing abstract properties*

public override int numbers

{

get

{

return myNumber;

}

set

{

myNumber = value;

}

}

}

*//Incorrect*

abstract sealed class absClass

{

}

An abstract method cannot be private.

*//Incorrect*

private abstract int MultiplyTwoNumbers();

Hide   Copy Code

*//Incorrect*

public abstract virtual int MultiplyTwoNumbers();

An abstract member cannot be static.

Hide   Copy Code

*//Incorrect*

publpublic abstract static int MultiplyTwoNumbers();