public abstract class Car  
{  
    public abstract void StartCar();  
}  
  
public class Audi : Car  
{  
    #region Overrides of Car  
  
    public override void StartCar()  
    {  
        UseCarKey();  
    }  
  
    #endregion  
}  
  
public class BMW : Car  
{  
    #region Overrides of Car  
  
    public override void StartCar()  
    {  
        UseCarChipcard();  
    }  
  
    #endregion  
}

**OOPS Features**

\* The object oriented programming (OOP) is a programming model   
  
where Programs are organized around object and data rather than   
  
action and logic.   
  
  
\* OOP allow decomposition of a problem into a number of entities called  
  
Object and then builds data and function around these objects.

* The Program is divided into number of small units called Object. The data and function are build around these objects.
* The data of the objects can be accessed only by the functions associated with that object.
* The functions of one object can access the functions of other object.

OOP has the following important features.

**Class:**

A class is the core of any modern Object Oriented Programming language such as C#.

In OOP languages it is must to create a class for representing data.

Class is a blueprint of an object that contains variables for storing data and functions to performing operations on these data.

Class will not occupy any memory space and hence it is only logical   
  
representation of data.

To create a class, you simply use the keyword "class" followed by the class name:

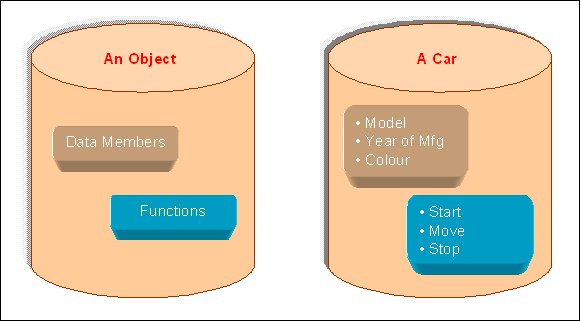
class Employee

{

}

**Object:**  
  
Objects are the basic run-time entities in an object oriented system.They may represent a person,a place or any item that the program has to handle.   
  
"Object is a Software bundle of related variable and methods. "

“Object is an instance of a class”

[](http://2.bp.blogspot.com/-hsy3T6ySv2w/T2tZ67pWJEI/AAAAAAAAB8I/Sg70P8dgqeU/s1600/what+is+Object.jpg)

Class will not occupy any memory space. Hence to work with thedata represented by the class you must create a variable for the class, which is called as an object.

When an object is created by using the keyword **new**, then memory will be allocated for the class in heap memory area, which is called as an instance and its starting address will be stored in the object in stack memory area.

 When an object is created without the keyword new, then memory will not be allocated in heap I.e. instance will not be created and object in the stack contains the value **null.**

When an object contains null, then it is not possible to access the members of the class using that object.

class Employee

{

}

Syntax to create an object of class Employee:-

Employee objEmp = new Employee();

All the programming languages supporting object oriented Programming will be supporting these three main concepts:

1. Encapsulation
2. Inheritance
3. Polymorphism

**Abstraction:**

Abstraction is used to create a common set of methods that might have different specific implementations by subclasses. Abstract class cannot be instantiated and consists of abstract methods without any implementations. Classes inheriting from abstract class must implement all the methods in abstract class.

Abstraction allows us to represent complex real world in simplest manner. It is process of identifying the relevant qualities and behaviors an object should possess, in other word represent the necessary feature without representing the back ground details. Abstraction is a process of hiding work style of an object and showing only those information which are required to understand the object. Abstraction means putting all the variables and methods in a class which are necessary.

Abstraction is "To represent the essential feature without representing the back ground details."

Abstraction lets you focus on what the object does instead of how it does it.

Abstraction provides you a generalized view of your classes or object by providing relevant information.

Abstraction is the process of hiding the working style of an object, and showing the information of an object in understandable manner.

**Real world Example of Abstraction:** -

Suppose you have an object Mobile Phone.

Suppose you have 3 mobile phones as following:-

Nokia 1400 (Features:- Calling, SMS)

Nokia 2700 (Features:- Calling, SM**S**, FM Radio, MP3, Camera)

Black Berry (Features:-Calling, SMS, FM Radio, MP3, Camera, Video Recording, Reading E-mails)

Abstract information (Necessary and Common Information) for the object "Mobile Phone" is make a call to any number and can send SMS."

so that, for mobile phone object you will have abstract class like following:-

    abstract class MobilePhone

    {

        public void Calling();

        public void SendSMS();

    }

    public class Nokia1400 : MobilePhone

    {

    }

    public class Nokia2700 : MobilePhone

    {

        public void FMRadio();

        public void MP3();

        public void Camera();

    }

    public class BlackBerry : MobilePhone

    {

        public void FMRadio();

        public void MP3();

        public void Camera();

        public void Recording();

        public void ReadAndSendEmails();

    }

Abstraction means putting all the variables and methods in a class which are necessary.

For example: - Abstract class and abstract method.

Abstraction is the common thing.

example:

If somebody in your collage tell you to fill application form, you will fill your details like name, address, data of birth, which semester, percentage you have got etc.

If some doctor gives you an application to fill the details, you will fill the details like name, address, date of birth, blood group, height and weight.

See in the above example what is the common thing?

Age, name, address so you can create the class which consist of common thing that is called abstract class.

That class is not complete and it can inherit by other class.

There are some important points about Abstract Base Class :

1. An Abstract Base class can not be instantiated; it means the object of that class can not be created.
2. Class having abstract keyword and having, abstract keyword with some of its methods (not all) is known as an Abstract Base Class.
3. Class having Abstract keyword and having abstract keyword with all of its methods is known as pure Abstract Base Class.
4. The method of abstract class that has no implementation is known as "operation". It can be defined as abstract void method ();
5. An abstract class holds the methods but the actual implementation of those methods is made in derived class.

 class program

{

abstract class animal

{

public abstract void eat();

public void sound()

{

Console.WriteLine("dog can sound");

}

}

class dog : animal

{

public override void eat() { Console.WriteLine("dog can eat"); }

}

static void Main(string[] args)

{

dog mydog = new dog();

animal thePet = mydog;

thePet.eat();

mydog.sound();

}

}

1. Need to create multiple versions of your component since versioning is not a problem with abstract class. You can add properties or methods to an abstract class without breaking the code and all inheriting classes are automatically updated with the change.
2. Need to to provide default behaviors as well as common behaviors that multiple derived classes can share and override.

**Encapsulation:**

Encapsulate means to hide. Encapsulation is also called data hiding.You can think Encapsulation like a capsule (medicine tablet) which hides medicine inside it. Encapsulation is wrapping, just hiding properties and methods. Encapsulation is used for hide the code and data in a single unit to protect the data from the outside the world. Class is the best example of encapsulation.

Class Encapsulation

{

    private int marks;

    public int Marks

   {

      get { return marks; }

      set { marks = value;}

    }

}

Wrapping up data member and method together into a single unit (i.e. Class) is called Encapsulation.

Encapsulation is like enclosing in a capsule. That is enclosing the related operations and data related to an object into that object.

Encapsulation is like your bag in which you can keep your pen, book etc. It means this is the property of encapsulating members and functions.

    class Bag

    {

        book;

        pen;

        ReadBook();

    }

Encapsulation means hiding the internal details of an object, i.e. how an object does something.

Encapsulation prevents clients from seeing its inside view, where the behaviour of the abstraction is implemented.

Encapsulation is a technique used to protect the information in an object from the other object.

Hide the data for security such as making the variables as private, and expose the property to access the private data which would be public.

So, when you access the property you can validate the data and set it.

Example:

class Demo

{

   private int \_mark;

   public int Mark

   {

     get { return \_mark; }

     set { if (\_mark > 0) \_mark = value; else \_mark = 0; }

   }

 }

**Real world Example of Encapsulation:-**

Let's take example of Mobile Phone and Mobile Phone Manufacturer

Suppose you are a Mobile Phone Manufacturer and you designed and developed a Mobile Phone design(class), now by using machinery you are manufacturing a Mobile Phone(object) for selling, when you sell your Mobile Phone the user only learn how to use the Mobile Phone but not that how this Mobile Phone works.

This means that you are creating the class with function and by making object (capsule) of it you are making availability of the functionality of you class by that object and without the interference in the original class.

**Example-2:**

TV operation

It is encapsulated with cover and we can operate with remote and no need to open TV and change the channel.

Here everything is in private except remote so that anyone can access not to operate and change the things in TV.

**Inheritance:**

When a class acquire the property of another class is known as inheritance.

Inheritance is process of object reusability.

For example, A Child acquire property of Parents.

public class ParentClass

    {

        public ParentClass()

        {

            Console.WriteLine("Parent Constructor.");

        }

        public void print()

        {

            Console.WriteLine("I'm a Parent Class.");

        }

    }

    public class ChildClass : ParentClass

    {

        public ChildClass()

        {

            Console.WriteLine("Child Constructor.");

        }

        public static void Main()

        {

            ChildClass child = new ChildClass();

            child.print();

        }

    }

**Output:**

**Parent Constructor.  
    Child Constructor.  
    I'm a Parent Class.**

In Object Oriented Programing concept there are 3 types of inheritences.

1. Single Inheritence,

2. Multiple Inheritence

3. Multilevel Inheritence

Single Inheritence:

public class A

{

}

public class B:A

{

}

Multiple Inheritence:

public class A

{

}

public class B

{

}

public class C:A,B

{

}

Multilevel Inheritence:

public class A

{

}

public class B:A

{

}

public class C:B

{

}

In the above three types C# dont proved Multiple Inheritence. As there is conflict of multiple override metods in base classes (say A, B in above example)

As in Place C# give another feature called Interfaces

using interfaces you can achive multiple Inheritence feature.

Ex:

public interface AX

{

}

public interface AY

{

}

public class LM:AX,AY

{

}

-------------------------------------------------------------

Single Inheritance, Multilevel Inheritance, Multiple Inheritance, Hierarchical Inheritance, Hybrid Inheritance

In this post i would like to explain about Inheritance types with basic examples on each inheritance.

In C# inheritance are five types Single,Multilevel,Multiple,Hierarchical and Hybrid Inheritance

1) Single Inheritance : Single class is derived from single base class

Example

public class BaseClass

{

public BaseClass()

{

// Default Constructor

}

}

public class SubClass : BaseClass

{

public SubClass()

{

// Default Constructor

}

}

2) Multilevel : When a Class is derived from a class which is derived from Base class

Example

public class BaseClass

{

public BaseClass()

{

// Default Constructor

}

}

public class SubClass1 : BaseClass

{

public SubClass1()

{

// Default Constructor

}

}

public class SubClass2 : SubClass1

{

public SubClass2()

{

// Default Constructor

}

}

In above example SubClass2 is derived from SubClass1 which is derived from BaseClass

3) Multiple Inheritance : When a class is derived from more than one Base-class.

C# does not support Multiple inheritance due to ambiguity problem. By using Interfaces we can achieve Multiple inheritance.

Example

interface interface1

{

}

interface interface2

{

}

public class Subclass : interface1, interface2

{

}

Subclass is derived from interface1 and interface2, like this we can derive a class from one or more interfaces by just separating interface names with comma

4) Hierarchical inheritance : When two or more classes are derived from a single base class

Example

public class BaseClass

{

public BaseClass()

{

// Default Constructor

}

}

public class SubClass1 : BaseClass

{

public SubClass1()

{

// Default Constructor

}

}

public class SubClass2 : BaseClass

{

public SubClass2()

{

// Default Constructor

}

}

In above example SubClass1,SubClass2 are derived from single BaseClass

5) Hybrid Inheritance : It is the combination of Single inheritance, Multilevel inheritance and Hierarchical inheritance.

Example

public class Class1

{

}

public class Class2 : Class1

{

}

public class DerSubClass1 : Class2

{

}

public class DerSubClass2 : Class2

{

}

In above example Class2 is derived from Single base class Class1 implies Single Inheritance;

DerSubClass1 is derived from Class2 Which is derived from Class1 implies Multilevel inheritance.

DerSubClass1 and DerSubClass2 are derived from Class2 implies Hierarchical inheritance.

**Polymorphism:**

Polymorphism means **one name many forms**. Polymorphism means one object behaving as multiple forms.

One function behaves different forms.

In other words, "Many forms of a single object is called Polymorphism."

**Real World Example of Polymorphism:**

**Example-1:**

A Teacher behaves to student.

A Teacher behaves to his/her seniors.

Here teacher is an object but attitude is different in different situation.

**Example-2:**

Person behaves SON in house at the same time that person behaves EMPLOYEE in office.

**Example-3:**

Your mobile phone, one name but many forms

* As phone
* As camera
* As mp3 player
* As radio

To Read Polmorphism in Detail click following link:-

What are types of Polymorphism

1. There are basically two types of polymorphism in c# i.e.  
     
   Static or compile time polymorphism

Method overloading

Dynamic or runtime polymorphism

Method overriding

Static Polymorphism

The mechanism of linking a function with an object during compile time is called early binding. It is also called static binding. C# provides two techniques to implement static polymorphism. These are:

* Function overloading
* Operator overloading

We will discuss function overloading in the next section and operator overloading will be dealt with in next chapter.

Function Overloading

You can have multiple definitions for the same function name in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Following is the example where same function **print()** is being used to print different data types:

using System;

namespace PolymorphismApplication

{

class Printdata

{

void print(int i)

{

Console.WriteLine("Printing int: {0}", i );

}

void print(double f)

{

Console.WriteLine("Printing float: {0}" , f);

}

void print(string s)

{

Console.WriteLine("Printing string: {0}", s);

}

static void Main(string[] args)

{

Printdata p = new Printdata();

// Call print to print integer

p.print(5);

// Call print to print float

p.print(500.263);

// Call print to print string

p.print("Hello C++");

Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result:

Printing int: 5

Printing float: 500.263

Printing string: Hello C++

Dynamic Polymorphism

C# allows you to create abstract classes that are used to provide partial class implementation of an interface. Implementation is completed when a derived class inherits from it. **Abstract** classes contain abstract methods, which are implemented by the derived class. The derived classes have more specialized functionality.

Please note the following rules about abstract classes:

* You cannot create an instance of an abstract class
* You cannot declare an abstract method outside an abstract class
* When a class is declared **sealed**, it cannot be inherited, abstract classes cannot be declared sealed.

The following program demonstrates an abstract class:

using System;

namespace PolymorphismApplication

{

abstract class Shape

{

public abstract int area();

}

class Rectangle: Shape

{

private int length;

private int width;

public Rectangle( int a=0, int b=0)

{

length = a;

width = b;

}

public override int area ()

{

Console.WriteLine("Rectangle class area :");

return (width \* length);

}

}

class RectangleTester

{

static void Main(string[] args)

{

Rectangle r = new Rectangle(10, 7);

double a = r.area();

Console.WriteLine("Area: {0}",a);

Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result:

Rectangle class area :

Area: 70

When you have a function defined in a class that you want to be implemented in an inherited class(es), you use **virtual** functions. The virtual functions could be implemented differently in different inherited class and the call to these functions will be decided at runtime.

Dynamic polymorphism is implemented by **abstract classes** and **virtual functions**.

The following program demonstrates this:

using System;

namespace PolymorphismApplication

{

class Shape

{

protected int width, height;

public Shape( int a=0, int b=0)

{

width = a;

height = b;

}

public virtual int area()

{

Console.WriteLine("Parent class area :");

return 0;

}

}

class Rectangle: Shape

{

public Rectangle( int a=0, int b=0): base(a, b)

{

}

public override int area ()

{

Console.WriteLine("Rectangle class area :");

return (width \* height);

}

}

class Triangle: Shape

{

public Triangle(int a = 0, int b = 0): base(a, b)

{

}

public override int area()

{

Console.WriteLine("Triangle class area :");

return (width \* height / 2);

}

}

class Caller

{

public void CallArea(Shape sh)

{

int a;

a = sh.area();

Console.WriteLine("Area: {0}", a);

}

}

class Tester

{

static void Main(string[] args)

{

Caller c = new Caller();

Rectangle r = new Rectangle(10, 7);

Triangle t = new Triangle(10, 5);

c.CallArea(r);

c.CallArea(t);

Console.ReadKey();

}

}

}

When the above code is compiled and executed, it produces the following result:

Rectangle class area:

Area: 70

Triangle class area:

Area: 25

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/1445/Introduction-to-inheritance-polymorphism-in-C)

class Animal

{

public Animal()

{

Console.WriteLine("Animal constructor");

}

public void Greet()

{

Console.WriteLine("Animal says Hello");

}

public void Talk()

{

Console.WriteLine("Animal talk");

}

public virtual void Sing()

{

Console.WriteLine("Animal song");

}

};

Now see how we derive another class from this base class.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/1445/Introduction-to-inheritance-polymorphism-in-C)

class Dog : Animal

{

public Dog()

{

Console.WriteLine("Dog constructor");

}

public new void Talk()

{

Console.WriteLine("Dog talk");

}

public override void Sing()

{

Console.WriteLine("Dog song");

}

};

Now try this code out.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/1445/Introduction-to-inheritance-polymorphism-in-C)

Animal a1 = new Animal();

a1.Talk();

a1.Sing();

a1.Greet();

*//Output*

Animal constructor

Animal talk

Animal song

Animal says Hello

Okay, that came out just as expected. Now try this code out.

http://www.codeproject.com/images/minus.gif Collapse | [Copy Code](http://www.codeproject.com/Articles/1445/Introduction-to-inheritance-polymorphism-in-C)

Animal a2 = new Dog();

a2.Talk();

a2.Sing();

a2.Greet();

*//Output*

Animal constructor

Dog constructor

Animal talk

Dog song

Animal says Hello

**Polymorphism**

Polymorphism means many forms (ability to take more than one form). In Polymorphism poly means “multiple” and morph means “forms” so polymorphism means many forms.

In polymorphism we will declare methods with same name and different parameters in same class or methods with same name and same parameters in different classes. Polymorphism has ability to provide different implementation of methods that are implemented with same name.

In Polymorphism we have 2 different types those are

        -   **Compile Time Polymorphism**(Called as Early Binding or Overloading or static binding)

        -   **Run Time Polymorphism**(Called as Late Binding or Overriding or dynamic binding)

**Compile Time Polymorphism**

Compile time polymorphism means we will declare methods with same name but different signatures because of this we will perform different tasks with same method name. This compile time polymorphism also called as **early binding** or **method overloading**.

Method Overloading or compile time polymorphism means same method names with different signatures (different parameters)

**Example**

|  |
| --- |
| public class Class1  {  public void NumbersAdd(int a, int b)  {  Console.WriteLine(a + b);  }  public void NumbersAdd(int a, int b, int c)  {  Console.WriteLine(a + b + c);  }  } |

In above class we have two methods with same name but having different input parameters this is called method overloading or compile time polymorphism or early binding.

**Run Time Polymorphism**

Run time polymorphism also called as **late binding** or **method overriding** or **dynamic polymorphism**. Run time polymorphism or method overriding means same method names with same signatures.

In this run time polymorphism or method overriding we can override a method in base class by creating similar function in derived class this can be achieved by using inheritance principle and using “**virtual** &**override**” keywords.

In base class if we declare methods with **virtual** keyword then only we can override those methods in derived class using **override** keyword

**Example**

|  |
| --- |
| //Base Class  public class Bclass  {  public virtual void Sample1()  {  Console.WriteLine("Base Class");  }  }  // Derived Class  public class DClass : Bclass  {  public override void Sample1()  {  Console.WriteLine("Derived Class");  }  }  // Using base and derived class  class Program  {  static void Main(string[] args)  {  // calling the overriden method  DClass objDc = new DClass();  objDc.Sample1();  // calling the base class method  Bclass objBc = new DClass();  objBc.Sample1();  }  } |

If we run above code we will get output like as shown below

**Output**

|  |
| --- |
| ----------------------------------  Derived Class  Derived Class |

using System;

**Example**

class A

{

public virtual void Test()

{

Console.WriteLine("A.Test");

}

}

class B : A

{

public override void Test()

{

Console.WriteLine("B.Test");

}

}

class Program

{

static void Main()

{

// Compile-time type is A.

// Runtime type is A as well.

A ref1 = new A();

ref1.Test();

// Compile-time type is A.

// Runtime type is B.

A ref2 = new B();

ref2.Test();

}

}

Output

A.Test

B.Test

**Example**

class BC

{

public void Display()

{

System.Console.WriteLine("BC::Display");

}

}

class DC : BC

{

new public void Display()

{

System.Console.WriteLine("DC::Display");

}

}

class Demo

{

public static void Main()

{

BC b;

b = new BC();

b.Display();

b = new DC();

b.Display();

}

}

Output

BC::Display

BC::Display

**Live Example Overriding**

class Animal

{

public virtual void eat()

{

Console.Write("Animal eating");

}

}

class Dog : Animal

{

public override void eat()

{

Console.Write("Dog eating");

}

}

class Program

{

public void Main()

{

Animal dog = new Dog();

Animal generic = new Animal();

dog.eat();

generic.eat();

}

}

public class Shape

{

// A few example members

public int X { get; private set; }

public int Y { get; private set; }

public int Height { get; set; }

public int Width { get; set; }

// Virtual method

public virtual void Draw()

{

Console.WriteLine("Performing base class drawing tasks");

}

}

class Circle : Shape

{

public override void Draw()

{

// Code to draw a circle...

Console.WriteLine("Drawing a circle");

base.Draw();

}

}

class Rectangle : Shape

{

public override void Draw()

{

// Code to draw a rectangle...

Console.WriteLine("Drawing a rectangle");

base.Draw();

}

}

class Triangle : Shape

{

public override void Draw()

{

// Code to draw a triangle...

Console.WriteLine("Drawing a triangle");

base.Draw();

}

}

class Program

{

static void Main(string[] args)

{

// Polymorphism at work #1: a Rectangle, Triangle and Circle

// can all be used whereever a Shape is expected. No cast is

// required because an implicit conversion exists from a derived

// class to its base class.

System.Collections.Generic.List<Shape> shapes = new System.Collections.Generic.List<Shape>();

shapes.Add(new Rectangle());

shapes.Add(new Triangle());

shapes.Add(new Circle());

// Polymorphism at work #2: the virtual method Draw is

// invoked on each of the derived classes, not the base class.

foreach (Shape s in shapes)

{

s.Draw();

}

// Keep the console open in debug mode.

Console.WriteLine("Press any key to exit.");

Console.ReadKey();

}

}

/\* Output:

Drawing a rectangle

Performing base class drawing tasks

Drawing a triangle

Performing base class drawing tasks

Drawing a circle

Performing base class drawing tasks

\*/

Interface

An interface contains only the signatures of [methods](http://msdn.microsoft.com/en-us/library/ms173114.aspx), [properties](http://msdn.microsoft.com/en-us/library/x9fsa0sw.aspx), [events](http://msdn.microsoft.com/en-us/library/awbftdfh.aspx) or [indexers](http://msdn.microsoft.com/en-us/library/6x16t2tx.aspx). A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition. In the following example, class ImplementationClass must implement a method named SampleMethod that has no parameters and returns **void**.

An interface can inherit from one or more base interfaces.

interface IFlyable

{

void Fly();

}

class Bird : IFlyable

{

public void Fly() { }

}

class Plane : IFlyable

{

public void Fly() { }

}

List<IFlyable> things = GetBirdInstancesAndPlaneInstancesMixed();

foreach(IFlyable item in things)

{

item.Fly();

}

interface IMyInterface  
{  
            void MethodToImplement();//Abstract Method signature.  
}

class InterfaceImplementer : IMyInterface  
{  
         static void Main()  
         {  
                     InterfaceImplementer  obj = new InterfaceImplementer();  
                     obj.MethodToImplement();  
         }  
  
         public void MethodToImplement()  
         {  
                //Abstract Method Implementation  
         }  
}

**HOW to User TWO interface’s same method (name) in Inherite same class**

public interface ITest {

void Test();

}

public interface ITest2 {

void Test();

}

public class Dual : ITest, ITest2

{

void ITest.Test() {

Console.WriteLine("ITest.Test");

}

void ITest2.Test() {

Console.WriteLine("ITest2.Test");

}

}

public interface IFoo1

{

void DoStuff();

}

public interface IFoo2

{

void DoStuff();

}

You can implement both like this:

public class Foo : IFoo1, IFoo2

{

public void IFoo1.DoStuff() { }

public void IFoo2.DoStuff() { }

}

using System;

interface A

{

    void Hello();

}

interface B

{

    void Hello();

}

class Test : A, B

{

    void A.Hello()

    {

        Console.WriteLine("Hello to all-A");

    }

    void B.Hello()

    {

        Console.WriteLine("Hello to all-B");

    }

}

public class interfacetest

{

    public static void Main()

    {

        A Obj1 = new Test();

        Obj1.Hello();

        B Obj2 = new Test();

        Obj2.Hello();

    }

}

Difference bet. Abstract class and interface

**What is an Abstract Class?**

An abstract class is a special kind of class that cannot be instantiated. So the question is why we need a class that cannot be instantiated? An abstract class is only to be sub-classed (inherited from). In other words, it only allows other classes to inherit from it but cannot be instantiated. The advantage is that it enforces certain hierarchies for all the subclasses. In simple words, it is a kind of contract that forces all the subclasses to carry on the same hierarchies or standards.

**What is an Interface**

An interface is not a class. It is an entity that is defined by the word Interface. An interface has no implementation; it only has the signature or in other words, just the definition of the methods without the body. As one of the similarities to Abstract class, it is a contract that is used to define hierarchies for all subclasses or it defines specific set of methods and their arguments. The main difference between them is that a class can implement more than one interface but can only inherit from one abstract class. Since C# doesn't support multiple inheritance, interfaces are used to implement multiple inheritance.

When we create an interface, we are basically creating a set of methods without any implementation that must be overridden by the implemented classes. The advantage is that it provides a way for a class to be a part of two classes: one from inheritance hierarchy and one from the interface.

When we create an abstract class, we are creating a base class that might have one or more completed methods but at least one or more methods are left uncompleted and declared abstract. If all the methods of an abstract class are uncompleted then it is same as an interface. The purpose of an abstract class is to provide a base class definition for how a set of derived classes will work and then allow the programmers to fill the implementation in the derived classes.

Interface : should be used if you want to imply a rule on the components which may or may not be

related to each other

Abstract Class : should be used where you want to have some basic or default behavior or

implementation for components related to each other

Pros:

Interface:

Allows multiple inheritance

provides abstraction by not exposing what exact kind of object is being used in the context

provides consistency by a specific signature of the contract

Abstract Class:

faster then interface

has flexibility in the implementation (you can implement it fully or partially)

can be easily changed without breaking the derived classes

Cons:

Interface :

Must implement all the contracts defined

cannot have variables or delegates

once defined cannot be changed without breaking all the classes

Abstract Class :

cannot be instantiated

does not support multiple inheritance

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Feature** | **Interface** | **Abstract class** | | Multiple inheritance | A class may inherit several interfaces. | A class may inherit only one abstract class. | | Default implementation | An interface cannot provide any code, just the signature. | An abstract class can provide complete, default code and/or just the details that have to be overridden. | | Access Modfiers | An interface cannot have access modifiers for the subs, functions, properties etc everything is assumed as public | An abstract class can contain access modifiers for the subs, functions, properties | | Core VS Peripheral | Interfaces are used to define the peripheral abilities of a class. In other words both Human and Vehicle can inherit from a IMovable interface. | An abstract class defines the core identity of a class and there it is used for objects of the same type. | | Homogeneity | If various implementations only share method signatures then it is better to use Interfaces. | If various implementations are of the same kind and use common behaviour or status then abstract class is better to use. | | Speed | Requires more time to find the actual method in the corresponding classes. | Fast | | Adding functionality (Versioning) | If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method. | If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly. | | Fields and Constants | No fields can be defined in interfaces | An abstract class can have fields and constrants defined | |

**Difference between Abstraction andEncapsulation :-**

|  |  |
| --- | --- |
| **Abstraction** | **Encapsulation** |
| 1. Abstraction solves the problem in the design level. | 1. Encapsulation solves the problem in the implementation level. |
| 2. Abstraction is used for hiding the unwanted data and giving relevant data. | 2. Encapsulation means hiding the code and data into a single unit to protect the data from outside world. |
| 3. Abstraction lets you focus on what the object does instead of how it does it | 3. Encapsulation means hiding the internal details or mechanics of how an object does something. |
| 4. **Abstraction**- Outer layout, used in terms of design.  For Example:-   Outer Look of a Mobile Phone, like it has a display screen and keypad buttons to dial a number. | 4. **Encapsulation**- Inner layout, used in terms of implementation.  For Example:- Inner Implementation detail of a Mobile Phone, how keypad button and Display Screen are connect with each other using circuits. |

The **easier way to understand Abstraction and encapsulation** is as follows:-

**Real World Example**:-

**Take an example of Mobile Phone**:-

You have a Mobile Phone, you can dial a number using keypad buttons. Even you don't know how these are working internally. This is called Abstraction. You have the only information that is needed to dial a number. But not its internal working of mobile.

**Multi-Level Inheritance**  
class Father  
{

    //Father Relations--Members  
}

class Mother: Father

{

    //Mother Relations--Members

}

class Child: Mother //Multi- Level Inheritance

{

    //Accessing both Relations class members.  
}  
  
**Multiple Inheritance**class Father

{

    //Father Relations--Members

}

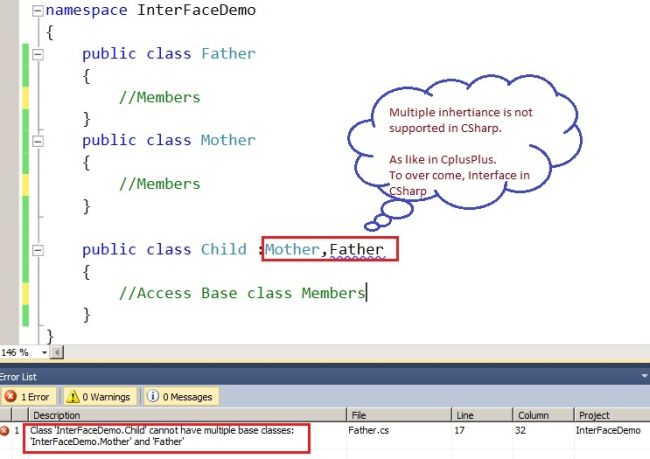
class Mother  
{

    //Mother Relations--Members

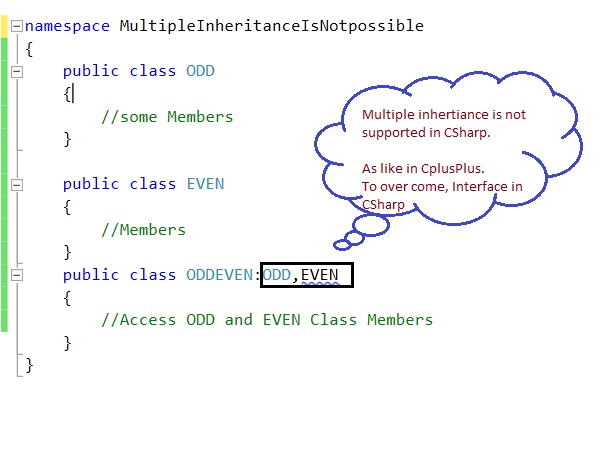
}

class Child: Mother,Father //Multiple Inheritance

{

    //Accessing both Relations class members.  
}  
  
This type of multiple inheritance is possible in C++ but it's not possible in C#.  
  
  
  
**What is Interface?**An interface looks like a class, but has no implementation. The only thing it contains are declarations of events, indexers, methods and/or properties. The reason interfaces only provide declarations is because they are inherited by structs and classes, that must provide an implementation for each interface member declared.  
  
So, what are interfaces good for if they don't implement functionality? They're great for putting together plug-n-play like architectures where components can be interchanged at will. Since all interchangeable components implement the same interface, they can be used without any extra programming. The interface forces each component to expose specific public members that will be used in a certain way.  
  
Because interfaces must be implemented by derived structs and classes, they define a contract.

interface IMyInterface  
{  
            void MethodToImplement();//Abstract Method signature.  
}

class InterfaceImplementer : IMyInterface  
{  
         static void Main()  
         {  
                     InterfaceImplementer  obj = new InterfaceImplementer();  
                     obj.MethodToImplement();  
         }  
  
         public void MethodToImplement()  
         {  
                //Abstract Method Implementation  
         }  
}  
  
Here, we advised to use "I" as the prefix for the interface to understand that the interface is an interface.  
  
**Best example for Interface**  
  
Here I have the best example for understanding interfaces.  
  
Not Possible:  
  
  
  
1. Open a Console Application and give "InterFaceDemo" as the project name, then add a new class item and rename it "ODDEVEN.CS".  
  
**ODDEVEN.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace InterFaceDemo

{

    interface IOne

    {

        void ONE();//Pure Abstract Method Signature

    }

    interface ITwo

    {

        void TWO();

    }

    interface IThree:IOne

    {

        void THREE();

    }

    interface IFour

    {

        void FOUR();

    }

    interface IFive:IThree

    {

        void FIVE();

    }

    interface IEVEN:ITwo,IFour

    {

    }

    class ODDEVEN:IEVEN,IFive//Must Implement all the abstract method, in Derived class.

    {

        public void ONE()//Implementation of Abstract Method.

        {

            Console.WriteLine("This is ONE");

        }

        public void TWO()

        {

            Console.WriteLine("This is TWO");

        }

        public void THREE()

        {

            Console.WriteLine("This is THERE");

        }

        public void FOUR()

        {

            Console.WriteLine("This is FOUR");

        }

        public void FIVE()

        {

            Console.WriteLine("This is FIVE");

        }

    }

}  
  
**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace InterFaceDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            Console.WriteLine("This is ODD");

            IFive obj1 = new ODDEVEN();

            obj1.ONE();

            obj1.THREE();

            obj1.FIVE();

            Console.WriteLine("\n\nThis is EVEN");

            IEVEN obj2 = new ODDEVEN();

            obj2.TWO();

            obj2.FOUR();

            Console.ReadLine();

        }

    }

}

The following is the output:  
  
  
  
Enjoy Coding.

**Define Constructors?**

A constructor is a member function in a class that has the same name as its class. The constructor is automatically invoked whenever an object class is created. It constructs the values of data members while initializing the class.

|  |
| --- |
| **Define destructors?** |
|  | A destructor is called for a class object when that object passes out of scope or is explicitly deleted.A destructors as the name implies is used to destroy the objects that have been created by a constructors.Like a constructor , the destructor is a member function whose name is the same as the class name but is precided by a tilde. |

# Static Class :

May 11, 2011

A static class is you can say same as the non-static class, but there is one difference them is a static class can't be instantiated. In other words, you cannot use the "new" keyword to create a instance variable of the class type. As there is no instance variable, you can access the members of a static class by using the name of the class itself. Static classes and it's class members are used to create data and functions that can be accessed without creating an instance of the class. Static class members can be used to separate data and behavior that is independent of any object identity: the data and functions do not change regardless of what happens to the object. Static classes can be used when data or behavior is not present in the class that depends on object identity. Static classes are loaded automatically by the .[NET Framework](javascript:void(0);) CLR when that program or namespace containing the class is loaded.

# The main features of a static class are:

* static classes can only contain static members.
* static classes can not be instantiated.
* static classes are sealed and therefore cannot be inherited.
* static classes can not contain Instance Constructors (C# Programming Guide).

public static class Settings

{

static int i;

public static string GetName()

{

return "MyName";

}

}

class Program

{

static void Main(string[] args)

{

string str=Settings.GetName();

Console.Write(str);

Console.Read();

}

}

# Static Fields:

Static fields can be declared by using the keyword static.

class MySettings

{

public static int height;

public static int width = 20;

}

When we declare a static field inside a class, it can be initialized as shown above in the example with a value. All un-initialized static fields get automatically initialized to their default values when the containing class is loaded for the first time.

# For example

class MySettings

{

public static int height = 20;

public static int width;

public static int length = 25;

public MySettings(int i)

{

height = i;

width = i;

length = i;

}

}

class AllSettings

{

public static void Main()

{

Console.WriteLine("{0},{1},{2}", MySettings.height, MySettings.width, MySettings.length);

MySettings mc = new MySettings(25);

Console.WriteLine("{0},{1},{2}", MySettings.height, MySettings.width, MySettings.length);

}

}

# Static Method:

Static methods can be declared using Static keyword befor method name. The static methods can by accessed directly from the class. Static methods are normally faster to invoke on the call stack than instance methods

class MySettings

{

private static int height = 100;

private static int width = 150;

public static void MyMethod()

{

Console.WriteLine("{0},{1}", height, width);

}

}

class AllSettings

{

public static void Main()

{

MyClass.MyMethod();

}

# }

**This Keyword**

The **this** keyword refers to the current instance of the class and is also used as a modifier of the first parameter of an extension method.

[Example](javascript:void(0))

In this example, **this** is used to qualify the Employee class members, name and alias, which are hidden by similar names. It is also used to pass an object to the methodCalcTax, which belongs to another class.

C#

class Employee

{

private string name;

private string alias;

private decimal salary = 3000.00m;

// Constructor:

public Employee(string name, string alias)

{

// Use this to qualify the fields, name and alias:

this.name = name;

this.alias = alias;

}

// Printing method:

public void printEmployee()

{

Console.WriteLine("Name: {0}\nAlias: {1}", name, alias);

// Passing the object to the CalcTax method by using this:

Console.WriteLine("Taxes: {0:C}", Tax.CalcTax(this));

}

public decimal Salary

{

get { return salary; }

}

}

class Tax

{

public static decimal CalcTax(Employee E)

{

return 0.08m \* E.Salary;

}

}

class MainClass

{

static void Main()

{

// Create objects:

Employee E1 = new Employee("Mingda Pan", "mpan");

// Display results:

E1.printEmployee();

}

}

/\*

Output:

Name: Mingda Pan

Alias: mpan

Taxes: $240.00

\*/

public class Person

{

string name = ""; *//Field "name" in class Person*

*//Constructor of the Person class, takes the name of the Person*

*//as argument*

public Person (string name)

{

*//Assign the value of the constructor argument "name" to the field "name"*

this.name = name;

*//If you'd miss out the "this" here (name = name;) you would just assign the*

*//constructor argument to itself and the field "name" of the*

*//Person class would keep its value "".*

}

}

**Classes Only:**

* Can support inheritance
* Are reference (pointer) types
* The reference can be null
* Have memory overhead per new instance
* Heap

**Structs Only:**

* Cannot support inheritance
* Are value types
* Are passed by value (like integers)
* Cannot have a null reference (unless Nullable is used)
* Do not have a memory overhead per new instance - unless 'boxed'
* Stack

1. Structs cannot have explicit parameterless constructor where as a class can  
2. Structs cannot have destructors, where as a class can  
3. Struct can't inherit from another class where as a class can, Both structs and classes can inherit from an interface.

**14.   What are value types and reference types?**

Value types are stored in the Stack whereas reference types stored on heap.  
Value types:

[csharp] int, enum , byte, decimal, double, float, long[/csharp]

Reference Types:

[csharp] string , class, interface, object.[/csharp]

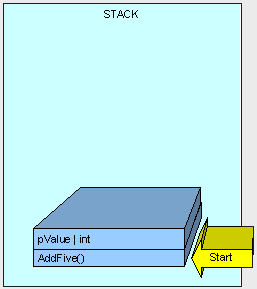
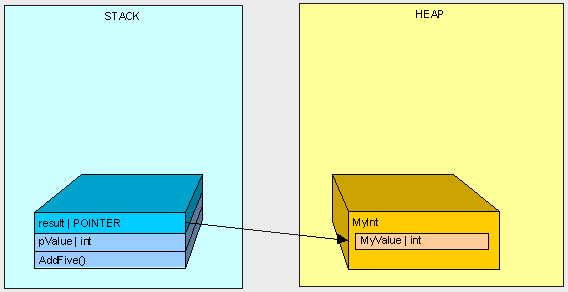
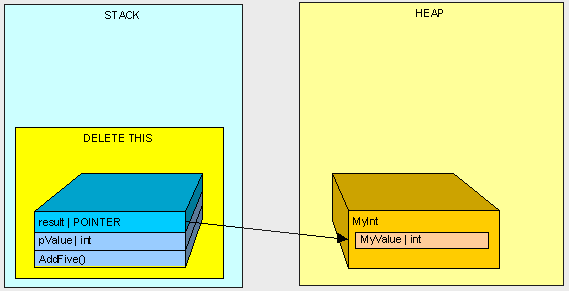
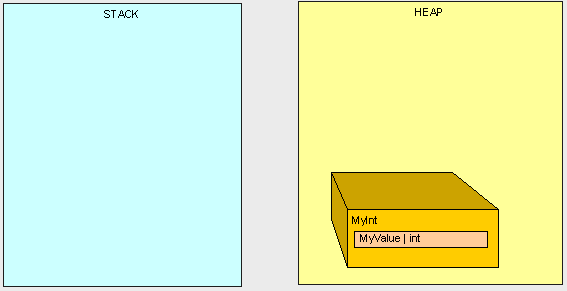
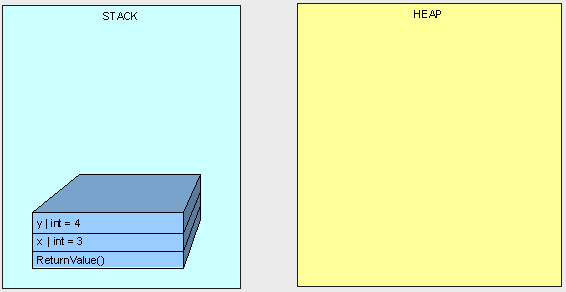
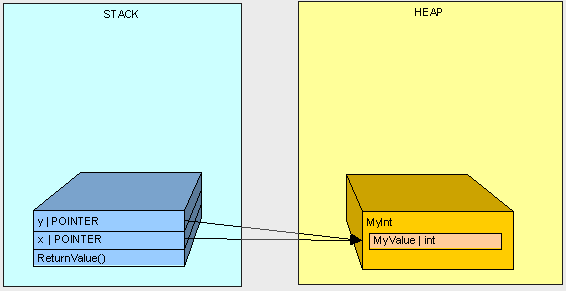
**Value Types:**

In C#, all the "things" declared with the following list of type declarations are Value types (because they are from System.ValueType):

* bool
* byte
* char
* decimal
* double
* enum
* float
* int
* long
* sbyte
* short
* struct
* uint
* ulong
* ushort

**Reference Types:**

All the "things" declared with the types in this list are Reference types (and inherit from System.Object, except, of course, for object which is the System.Object object):

* class
* interface
* delegate
* object
* string
* If we have the following MyInt class (which is a Reference Type because it is a class):
* public class MyInt  
            {            
               public int MyValue;  
            }
* and the following method is executing:
* public MyInt AddFive(int pValue)  
            {  
                  MyInt result = new MyInt();  
                  result.MyValue = pValue + 5;  
                  return result;  
            }
* Then just as before, the thread starts executing the method and its parameters are placed on sthe thread's stack.
* 
* Now is when it gets interesting.
* Because MyInt is a Reference Type, it is placed on the Heap and referenced by a Pointer on the Stack.
* 
* After AddFive() is finished executing (like in the first example), and we are cleaning up...
* 
* we're left with an orphaned MyInt in the Heap (there is no longer anyone in the Stack standing around pointing to MyInt)!
* 
* f we execute the following method:
* public int ReturnValue()  
            {  
                  int x = new int();  
                  x = 3;  
                  int y = new int();  
                  y = x;        
                  y = 4;            
                  return x;  
            }
* We'll get the value 3.  Simple enough, right?
* However, if we are using the MyInt class from before:
* public class MyInt  
            {  
                  public int MyValue;  
            }
* and we are executing the following method:
* public int ReturnValue2()  
            {  
                  MyInt x = new MyInt();  
                  x.MyValue = 3;  
                  MyInt y = new MyInt();  
                  y = x;                   
                  y.MyValue = 4;                
                  return x.MyValue;  
            }
* What do we get?    4!
* Why?...  How does x.MyValue get to be 4? Take a look at what we're doing and see if it makes sense:
* In the first example everything goes as planned:
* public int ReturnValue()  
            {  
                  int x = 3;  
                  int y = x;      
                  y = 4;  
                  return x;  
            }
* 
* In the next example, we don't get "3" because both variables "x" and "y" point to the same object in the Heap.
* public int ReturnValue2()  
            {  
                  MyInt x;  
                  x.MyValue = 3;  
                  MyInt y;  
                  y = x;                  
                  y.MyValue = 4;  
                  return x.MyValue;  
            }
* 
* Hopefully this gives you a better understanding of a basic difference between Value Type and Reference Type variables in C# and a basic understanding of what a Pointer is and when it is used.  In the next part of this series, we'll get further into memory management and specifically talk about method parameters.