INHERITANCE

Inheritance means parent-child relationship. By using Inheritance methodology. we can create a new class by using existing class code (i.e. reuse existing methods, properties, etc.). It also referred to as reusability of the code so by using Inheritance we can reuse the code again and again.

What We Call

In Inheritance main existing class is called as generalized class, base class, super class and parent class and the new class created from existing class is called as specialized class, sub class, child class and derived class. We normally talk in terms of base class and derived class.

Syntax of Inheritance

```
class ParentClass{

...parent class code

}

class ChildClass : ParentClass{

...child class code

}
```

Special Character “:” in Inheritance

Inheritance uses special character called ":" colon to make a relationship between parent and child as you can see in above syntax of code.

When to Implement Inheritance

When we create a new class and we want to reuse some of the methods or properties from existing class then that is an ideal time to implement inheritance.

Advantage of Inheritance

Reusability of the code.

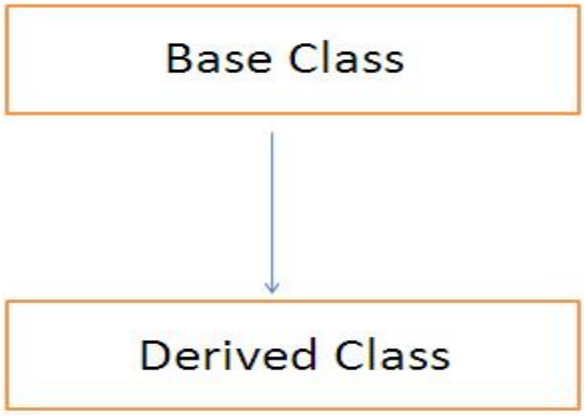
Types of inheritance in c#

There are 5 types of inheritance as shown below:

1.Single Inheritance
2.Multilevel Inheritance
3.Multiple Inheritance
4.Hierarchical Inheritance
5.Hybrid Inheritance

Single Inheritance

Single Inheritance means when a single base is been implemented to single derived class is called as Single Inheritance means we have only one parent class and one child class.



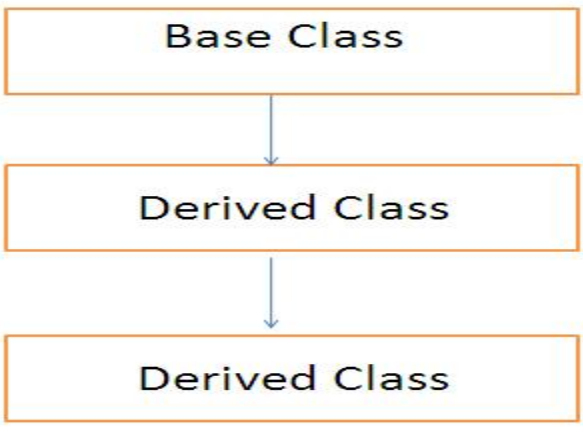
Example of Single Inheritance

```
class Company{

 public void CompanyName(){

  Console.WriteLine("Name of the Company");

 }

 public void CompanyAddress(){
```

```
    Console.WriteLine("Address of the Company");
  }
}
class Employee : Company {
  public void NameofEmployee(){
    Console.WriteLine("Name of the Employee");
  }
  public void Salary(){
    Console.WriteLine("Salary of the Employee");
  }
}
```

Multilevel Inheritance

When a derived class is created from another derived class or let me put it in this way that a class is created by using another derived class and this type of implementation is called as multilevel Inheritance



Example of Multilevel Inheritance

```
class HeadOffice{

  public void HeadOfficeAddress(){

   Console.WriteLine("Head Office Address");

  }
}
```

Now let's assume that class "HeadOffice" is our Parent class or Base class. Now in my next step I will create one derived class.

```
class BranchOffice : HeadOffice{

  public void BranchOfficeAddress(){

   Console.WriteLine("Branch Office Address");

  }

}
```

So as you can see that I have created a derived class "BranchOffice" by implementing the class "HeadOffice" to it.

It means now we have one parent class and one derived class. In the next step I will create another derived class by implementing our existing derived class "BranchOffice" to achieve multilevel Inheritance.

```
class Employee : BranchOffice {

   public void NameofEmployee(){

      Console.WriteLine("Name of the Employee");

   }

   public void Salary(){
```

```
    Console.WriteLine("Salary of the Employee");

  }
}
```

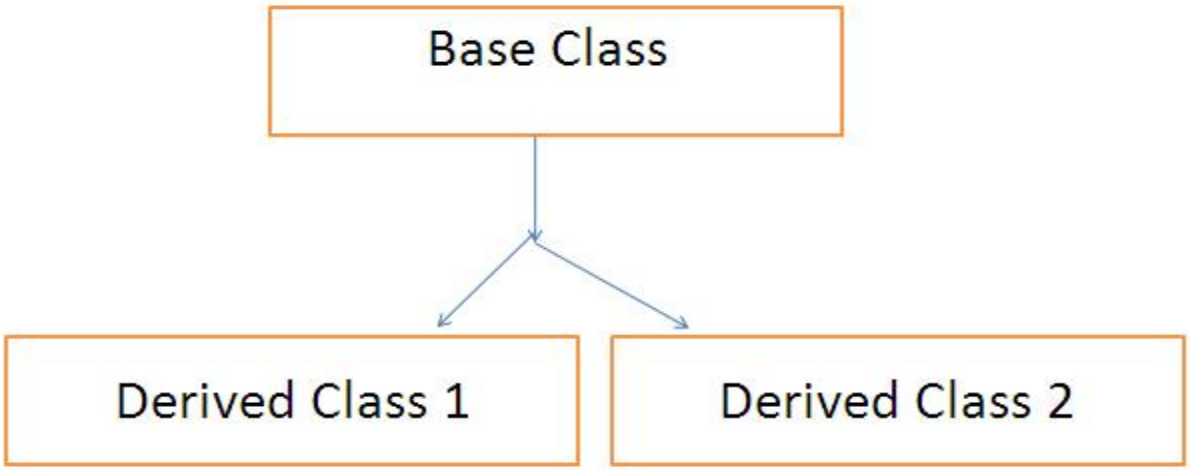
From the above source code you can see that we have achieved multilevel Inheritance by implementing one derived class to another derived class. Now the class "**Employee**" will have the access of all the properties and methods of class "**BranchOffice**" and class "**HeadOffice**".

Multiple Inheritance

Due to the complexity of a code multiple inheritance is not been supported in C# or in DOT.NET but DOT.NET or C# supports multiple interfaces.

Hierarchical Inheritance

When more than one derived classes are implemented from a same parent class or base class then that type of implementation is known as hierarchical inheritance.



In short it means single base class having more than one derived classes.

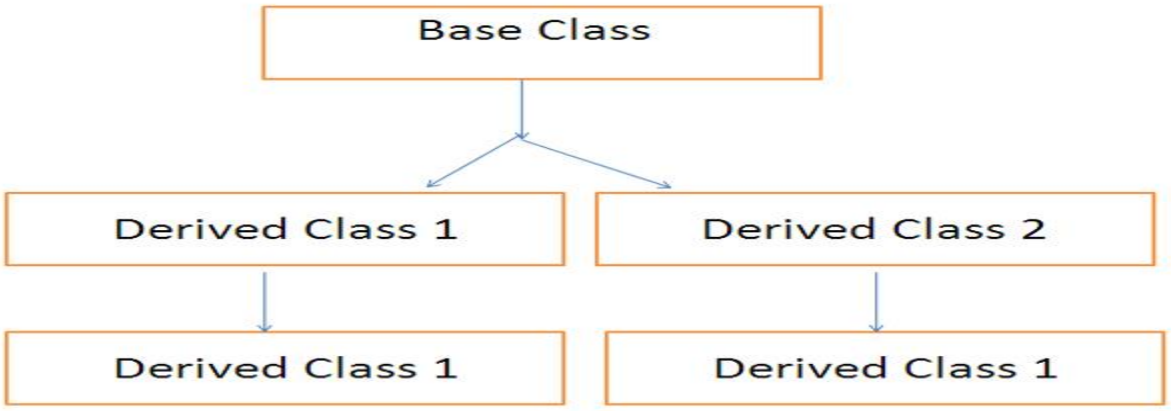
```
class HeadOffice{

 public void HeadOfficeAddress(){

  Console.WriteLine("Head Office Address");
```

```
  }
}

class BranchOffice1 : HeadOffice{

  public void BranchOfficeAddress(){

   Console.WriteLine("Branch Office Address");

  }

}

class BranchOffice2 : HeadOffice{

  public void BranchOfficeAddress(){

   Console.WriteLine("Branch Office Address");

  }

}
```

As you can see from above the code that we have one base class "**HeadOffice**" and two derived classes "**BranchOffice1**" and "**BranchOffice2**" which are implemented from same base class i.e. "**HeadOffice**".

Hybrid Inheritance

This is a special type of inheritance and can be achieved from any combination of single, hierarchical and multi-level inheritance known as hybrid inheritance.



In the below code example I have combined hierarchical inheritance and multi-level inheritance together.

```
//This part of code is related to hierarchical inheritance
class HeadOffice{

  public void HeadOfficeAddress(){

   Console.WriteLine("Head Office Address");

  }
}

class BranchOffice1 : HeadOffice{

  public void BranchOfficeAddress(){

   Console.WriteLine("Branch Office Address");

  }
}

class BranchOffice2 : HeadOffice{

  public void BranchOfficeAddress(){

   Console.WriteLine("Branch Office Address");

  }
}

//This part of code is related to combination of hierarchical inheritance and multi-level inheritance
class Employee : BranchOffice2  {
```

```
  public void NameofEmployee(){

    Console.WriteLine("Name of the Employee");

  }

  public void Salary(){

    Console.WriteLine("Salary of the Employee");

  }
}
```

So that you have understood the inheritance and their 5 types. Now let's a simple example of inheritance using c#.

Example of Inheritance using C#

```
public class Car{
    private string rearViewMirror;
    private string gear;
    private string clutch;
    private string steering;
    private int num = 4;

    public string RearViewMirror
    {
        get { return rearViewMirror; }
        set { rearViewMirror = value; }
    }

    public string Gear
    {
        get { return gear; }
        set { gear = value; }
    }

    public string Clutch
    {
        get { return clutch; }
        set { clutch = value; }
    }
```

```
    public string Steering
    {
       get { return steering; }
       set { steering = value; }
    }

    public virtual void CarPrice()
    {
       Console.WriteLine("Car Prize is : 0");
    }

    public virtual void NameofCar()
    {
       Console.WriteLine("Company Name of this Car is -- ");
    }
 }
```

Above I have created a simple class "**Car**" with some methods and some properties. Now next step I will create two classes "**SentroCar**" and "**BCWCar**" and implement it with a single base class "**Car**".

```
//Derived Class 1 implemented with base class Car
public class SentroCar:Car
 {

    public override void CarPrice()
    {
       Console.WriteLine("Car Prize is : 2.5L INR");
    }
    public override void NameofCar()
    {
       Console.WriteLine("Company Name of this Car is Sentro");
    }
 }

//Derived Class 2 implemented with base class Car
public class BCWCar : Car
{

   public override void CarPrice()
   {
      Console.WriteLine("Car Prize is :  4.5L INR");
   }
   public override void NameofCar()
   {
      Console.WriteLine("Company Name of this Car is BCW ");
   }
```

```
}
```

As you see that we have two derived classes implemented from same base class. This type of implementation can also be called as Hierarchical Inheritance.

Output by Using Console Application

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

        try{

         Console.WriteLine(num);    //Raise Error. It can't be accessed and non static

        }catch(Exception ex){

        Console.WriteLine("Exception caught: {0}", ex);

        }finally{

      Car obj = new SentroCar();
      obj.CarPrice();
      obj.NameofCar();

      Car obj = new BCWCar();
      obj.CarPrice();
      obj.NameofCar();

      }

    }
  }
```

Output

```
Car Prize is : 2.5L INR
Company Name of this Car is Sentro

Car Prize is : 4.5L INR
```

Company Name of this Car is BCW.

So this all about the inheritance. if you guys have any doubts or any query kindly let me and drop your valuable feedbacks.

Real World Illustration of Inheritance

Real time example of Single level inheritance –

In *Single level inheritance*, there is single base class & a single derived class

Maruti --> Swift

Here maruti is base class and Swift is derived Class

Real time example of Multi level inheritance –

In *Multilevel inheritance*, there is more than one single level of derived class.

Maruti --> Swift --> SwiftDzire

Here maruti is base class and Swift, SwiftDzire are derived Classes.

Real time example of Hierarchical inheritance –

Here multiple derived class would be extended from base class.

```
          |-->> Swift
          |
Maruti ---
          |
          |-->> Alto
```

Real time example of Hybrid inheritance -

Single, Multilevel, & hierarchical inheritance all together construct a hybrid inheritance.

```
          |-->> Swift -->> SwiftDzire
```

```
          |
Maruti ---
          |
          |-->> Alto
```

Which Type of Member can be accessed by a Child Class or Derived class?

A child class or derived class can access all the public, protected, internal and protected internal member. Private member cannot be accessed by child class however it is inherited and still present in child class and can be accessed using public property (GET SET modifier). There are two examples that demonstrate all the concept of member access clearly. First example will show which type of member can be accessed in child class and another example will show how to access private member in child class using GET SET modifier.

Example

```
using System;

namespace Member_Access
{
   class Program
   {
      static void Main(string[] args)
      {
         Childclass child = new Childclass ();
         child.checkmember();
         Console.ReadKey();
      }
   }
   class Baseclass
   {
      public void public_member()
      {
         Console.WriteLine("I am Public Method");
      }
      protected void protected_member()
      {
         Console.WriteLine("I am Protected Method");
```

```
    }
    internal void internal_member()
    {
       Console.WriteLine("I am Internal Method");
    }
    protected internal void protected_internal_member()
    {
       Console.WriteLine("I am protected internal method");
    }
    private void private_member()
    {
       Console.WriteLine("I am private method");
    }
  }
  class Childclass: Baseclass
  {
    public void checkmember()
    {
       public_member();
       protected_member();
       protected_internal_member();
       internal_member();
        try{

                  private_member();  //Raise Error. It can't be accessed

        }catch(Exception ex){

        Console.WriteLine("Exception caught: {0}", ex);

        }
    }
  }
}
```

Output

```
I am Public Method
I am Protected Method
I am protected internal method
I am Internal Method
```

**If you uncomment private_member() then you will get compile time error 'Member_Access.baseclass.private_member()' is inaccessible due to its protection level"

Access Private Member in Child class

This example shows how you can access private member in a child class.

Example

```
using System;

namespace Member_Variable
{
   class Program
   {
      static void Main(string[] args)
      {
         Childclass ch = new Childclass ();
         ch.Check();
         Console.ReadKey();
      }
   }
   class Baseclass
   {
      public int pub_var = 5;
      protected int pro_var = 6;
      internal int inter_var= 7;
      private int pri_var = 8;
      private int pri_var2 = 4;
      public int Private_variable
      {
         get
         {
            return pri_var;
         }
         set
         {
            pri_var = value;
         }
      }
   }
   class Childclass : Baseclass
   {
      public void Check ()
      {
```

```
        int sum = pub_var + pro_var + inter_var + pri_var;

         try{

          Console.WriteLine(pri_var);   //Raise Error. It can't be accessed

         }catch(Exception ex){

         Console.WriteLine("Exception caught: {0}", ex);

         }finally{

         Console.WriteLine("Total : " + sum.ToString());
         }

     }
   }
}
```

Output

```
Total : 26
```

Multiple inheritance in C#

Multiple inheritance allows programmers to create classes that combine aspects of multiple classes and their corresponding hierarchies. For ex. the C++ allows you to inherit from more than one class

In C#, the classes are only allowed to inherit from a single parent class, which is called single inheritance. But you can use interfaces or a combination of one class and interface(s), where interface(s) should be followed by class name in the signature.

Ex:

```
Class FirstClass { }
Class SecondClass { }


        interface X { }
```

```
interface Y { }
```

You can inherit like the following:

```
class NewClass : X, Y { }
```

In the above code, the class "NewClass" is created from multiple interfaces.

```
class NewClass : FirstClass, X { }
```

In the above code, the class "NewClass" is created from interface X and class "FirstClass".

WRONG:

```
class NewClass : FirstClass, SecondClass  { }
```

The above code is wrong in C# because the new class is created from class "FirstClass" and class "SecondClass". The reason is that C# does not allows multiple inheritance.

.Net Interface

An interface looks like a class, it contain only the declaration of the members but has no implementation. It provides a contract specifying how to create an Object, without caring about the specifics of how they do the things. An Interface is a reference type and it included only the signatures of methods, properties, events or indexers. In order to implement an interface member, the corresponding member of the implementing class should be public, non-static, and have the same name and signature as the interface member.

Keywords In C#

Base

The base keyword is used to access members of the base class from within a derived class:

- Call a method on the base class that has been overridden by another method.
- Specify which base-class constructor should be called when creating instances of the derived class.

A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

It is an error to use the base keyword from within a static method.

The base class that is accessed is the base class specified in the class declaration. For example, if you specify class ClassB : ClassA, the members of ClassA are accessed from ClassB, regardless of the base class of ClassA.

Ex:

In this example, both the base class, Person, and the derived class, Employee, have a method named Getinfo. By using the base keyword, it is possible to call the Getinfo method on the base class, from within the derived class.

```
public class Person
{
   protected string ssn = "444-55-6666";
   protected string name = "John L. Malgraine";

   public virtual void GetInfo()
   {
      Console.WriteLine("Name: {0}", name);
      Console.WriteLine("SSN: {0}", ssn);
   }
}
class Employee : Person
{
   public string id = "ABC567EFG";
   public override void GetInfo()
   {
      // Calling the base class GetInfo method:
      base.GetInfo();
      Console.WriteLine("Employee ID: {0}", id);
   }
}

class TestClass
{
   static void Main()
   {
      Employee E = new Employee();
```

```
        E.GetInfo();
    }
}
```

Output

```
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
```

Override

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

Ex:

In this example, the Square class must provide an overridden implementation of Area because Area is inherited from the abstract ShapesClass:

```
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());
    }

    interface I
    {
        void M();
    }
    abstract class C : I
    {
        public abstract void M();
    }

}
```

Output

```
Area of the square = 144
```

An override method provides a new implementation of a member that is inherited from a base class. The method that is overridden by an override declaration is known as the overridden base method. The overridden base method must have the same signature as the override method. For information about inheritance, see Inheritance.

You cannot override a non-virtual or static method. The overridden base method must be virtual, abstract, or override.

An override declaration cannot change the accessibility of the virtual method. Both the override method and the virtual method must have the same access level modifier.

You cannot use the new, static, or virtual modifiers to modify an override method.

An overriding property declaration must specify exactly the same access modifier, type, and name as the inherited property, and the overridden property must be virtual, abstract, or override.

Virtual

Virtual keyword is used for generating a virtual path for its derived classes on implementing method overriding. Virtual keyword is used within a set with override keyword. It is used as:

```
// Base Class
    class A
    {
        public virtual void Show()
```

```
    {
        Console.WriteLine("Hello: Base Class!");
        Console.ReadLine();
    }
}
```

Sealed

When applied to a class, the sealed modifier prevents other classes from inheriting from it. In the following example, class B inherits from class A, but no class can inherit from class B.

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

You can also use the sealed modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.

Ex:

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

```
class X
{
   protected virtual void F() { Console.WriteLine("X.F"); }
   protected virtual void F2() { Console.WriteLine("X.F2"); }
}
class Y : X
{
   sealed protected override void F() { Console.WriteLine("Y.F"); }
   protected override void F2() { Console.WriteLine("Y.F2"); }
}
class Z : Y
{
   // Attempting to override F causes compiler error CS0239.
   // protected override void F() { Console.WriteLine("C.F"); }

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

When you define new methods or properties in a class, you can prevent deriving classes from overriding them by not declaring them as virtual.

It is an error to use the abstract modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

When applied to a method or property, the sealed modifier must always be used with override.

Because structs are implicitly sealed, they cannot be inherited.

References

- https://www.completecsharptutorial.com/basic/
- http://www.dotnetfunda.com/articles/show/3502/understanding-the-csharp-concepts-through-real-time-examples
- https://social.msdn.microsoft.com/Forums/en-US/5c43ce21-38fb-425f-918f-a386d3d0366c/why-in-net-multiple-inheritance-is-not-allowed?forum=csharplanguage