C# Inheritance and interfaces

In C#, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which is defined in other class.

Important terminology:

Super Class: The class whose features are inherited is known as super class(or a base class or a parent class).
Sub Class: The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance

The symbol used for inheritance is **:**.

Syntax:

```
<access modifier> class derived-class : base-class
{
  // methods and fields
  .
  .
}
```

```
// C# program to illustrate the inheritance
using System;

namespace test
{

  public class Program
  {
    // Main Method
    static public void Main()
    {
      DerivedClass obj = new DerivedClass();
      obj.displayBC();
      obj.displayDC();
      Console.ReadKey();
    }
  }

  public class BaseClass
  {
    public void displayBC()
    {
      Console.WriteLine("I'm from base class");
    }
  }

  public class DerivedClass : BaseClass
  {
    public void displayDC()
```

```
    {
        Console.WriteLine("I'm from derived class");
    }
  }
}
```

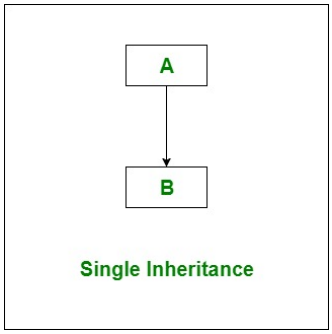
Output:
I'm from base class
I'm from derived class

Types of Inheritance in C#

Below are the different types of inheritance which is supported by C# in different combinations.

1. Single Inheritance:

In single inheritance, subclasses inherit the features of one superclass. In image below, the class A serves as a base class for the derived class B.

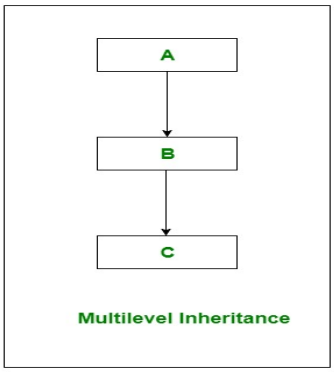



```
using System;
namespace Single
{
            public class Animal
             {
               public void eat() { Console.WriteLine("Eating..."); }
             }
            public class Dog: Animal
            {
               public void bark() { Console.WriteLine("Barking..."); }
            }

            class TestInheritance2
            {
               public static void Main(string[] args)
                {
                   Dog d1 = new Dog();
                   d1.eat();
                   d1.bark();
                 }
          }
}
```

Output:
Eating….
Barking…

2. Multilevel Inheritance:

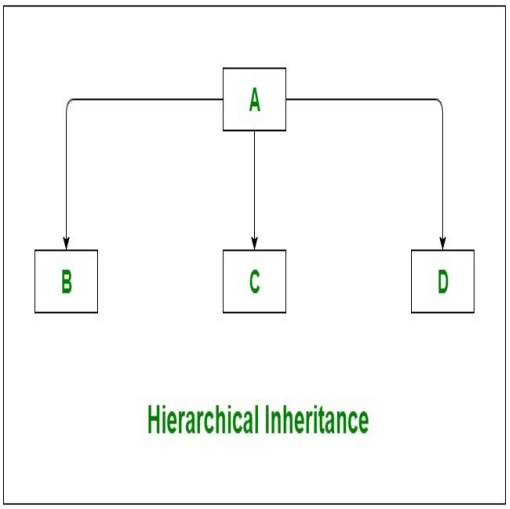
In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class. In below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.




```
 using System;
namespace MulitpleInt
{
        public class Animal
        {
           public void eat() { Console.WriteLine("Eating..."); }
        }
        public class Dog: Animal
        {
           public void bark() { Console.WriteLine("Barking..."); }
        }
        public class BabyDog : Dog
        {
           public void weep() { Console.WriteLine("Weeping..."); }
        }
        class TestInheritance2{
           public static void Main(string[] args)
            {
               BabyDog d1 = new BabyDog();
               d1.eat();
               d1.bark();
               d1.weep();
            }
         }
}
```

Output:
Eating...
Barking...
Weeping...

3. Hierarchical Inheritance:
In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In below image, class A serves as a base class for the derived class B, C, and D.



program to illustrate the hierachical inheritance

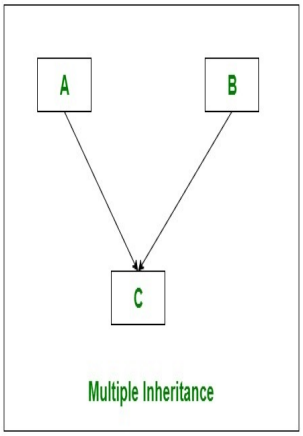
```
using System;
namespace test
{
   public class Animal
   {
      public void eat()
      {
         Console.WriteLine("Animal eating");
      }
   }

   public class Cat : Animal
   {
      public void meows()
      {
         Console.WriteLine("cat meows");
      }
   }

   public class Dog : Animal
   {
      public void barks()
      {
         Console.WriteLine("Dog barks");
      }
   }

   public class Program
   {
      static public void Main()
      {
         Cat objC = new Cat();
         objC.eat();
         objC.meows();

         Dog objD = new Dog();
         objD.eat();
```

```
            objD.barks();
            Console.ReadKey();
        }
    }
}
```

OutPut:
Animal eating
cat meows
Animal eating
Dog barks

4. Multiple Inheritance(Through Interfaces):

In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes. Please note that **C# does not support multiple inheritance** with classes. In C#, we can achieve multiple inheritance only through Interfaces. In the image below, Class C is derived from interface A and B.



// C# program to illustrate the multiple inheritance using interface

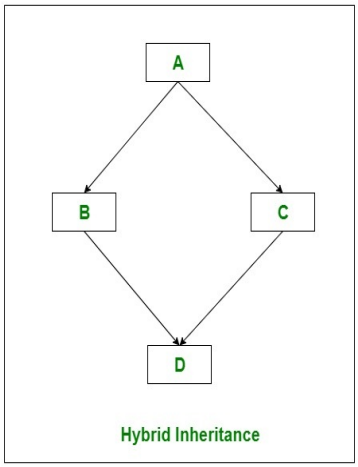
```
using System;
namespace test
{
   interface IArithmatic
   {
       int add(int a, int b);
   }

   interface IRelational
   {
      Boolean IsEqual(int a, int b);
   }

   class Mathematics:IArithmatic,IRelational
   {
      public int add(int a, int b)
      {
         return a + b;
      }

      public Boolean IsEqual(int a, int b)
```

```
    {
        if (a == b)
            return true;
        else
            return false;;
    }
}
public class Program
{
    static public void Main()
    {
        Mathematics obj = new Mathematics();
        Console.WriteLine("sum = {0}",obj.add(10,20));
        Console.WriteLine("both are equal? : {0}",obj.IsEqual(10,20));
        Console.ReadKey();
    }
 }
}
```

Output:
sum = 30
both are equal? : False

5. Hybrid Inheritance(Through Interfaces):

It is a mix of two or more of the above types of inheritance. Since C# doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In C#, we can achieve hybrid inheritance only through Interfaces.



// C# program to illustrate the hybrid inheritance using interfaces

```
using System;
namespace test
{
    interface IAnimals
    {
        void eat();
    }
```

```
    interface ICat:IAnimals
    {
        void meows();
    }

    interface IDog : IAnimals
    {
        void bark();
    }

    class Animals:IDog,ICat
    {
        public void meows()
        {
            Console.WriteLine("cat meows");
        }

        public void bark()
        {
            Console.WriteLine("dog barks");
        }

        public void eat()
        {
            Console.WriteLine("Animal eats");
        }
    }
    public class Program
    {
        static public void Main()
        {
            Animals obj = new Animals();
            obj.eat();
            obj.meows();
            obj.bark();
            Console.ReadKey();
        }
     }
}
```

Output:
Animal eats
cat meows
dog barks

Important facts about inheritance in C#

- **Default Superclass**: Except Object class, which has no superclass, every class has one and only one direct superclass(single inheritance). In the absence of any other explicit superclass, every class is implicitly a subclass of Object class.
- **Superclass can only be one:** A superclass can have any number of subclasses. But a subclass can have only **one** superclass. This is because C# does not support multiple inheritance with classes. Although with interfaces, multiple inheritance is supported by C#.
- **Inheriting Constructors:** A subclass inherits all the members (fields, methods) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.
- **Private member inheritance:** A subclass does not inherit the private members of its parent class. However, if the superclass has properties(get and set methods) for accessing its private fields, then a subclass can inherit.

Virtual method:

- A virtual method is a method that can be redefined in derived classes.
- A virtual method has an implementation in a base class as well as derived the class.
- A virtual method is created in the base class that can be overriden in the derived class.

It is used when a method's basic functionality is the same but sometimes more functionality is needed in the derived class. We create a virtual method in the base class using the virtual keyword and that method is overriden in the derived class using the override keyword.

Syntax:

```
<Access_specifer> virtual returntype methodName(parameterlist)
{
        Statements;
}
```

Ex.

```
Public virtual void getName()
{
        Console.WriteLine("abc");
}
```

Method overriding:

- If derived class defines same method as defined in its base class, it is known as method overriding in C#.
- It is used to achieve runtime polymorphism.
- It enables you to provide specific implementation of the method which is already provided by its base class.
- To perform method overriding in C#, you need to use **virtual** keyword with base class method and **override** keyword with derived class method

Syntax:

```
<Access_specifer> override returntype methodName(parameterlist)
{
        Statements;
}
```

Ex.

```
Public override void getName()
{
        Console.WriteLine("abc");
}
```

program to demonstrate virtual and override

```
using System;
namespace VirtualOvrd
{
```

```
class Program
{
    static void Main(string[] args)
    {
        // Base class, method draw is virtual
        Shape objShape = new Shape();
        objShape.draw();

        // Derived class, method draw is overidden
        Square objSquare = new Square();
        objSquare.draw();

        Console.ReadKey();
    }
}

// Base class
class Shape
{
    public virtual void  draw()
    {
        Console.WriteLine("From base class : Shape");
    }
}

// Derived class
class Square : Shape
{
    public override void draw()
    {
        Console.WriteLine("From derived class, Method(draw) is overidden : Square");
    }
}
}
```

Output:
From base class : Shape
From derived class, Method(draw) is overidden : Square

Interface:

Like a class, ***Interface*** can have methods, properties, events, and indexers as its members. But interfaces will contain only the declaration of the members. The implementation of interface's members will be given by class who implements the interface implicitly or explicitly.

- An interface cannot include private, protected, or internal members. All the members are public by default.
- By default all the members of Interface are public and abstract.
- The interface will always defined with the help of keyword '***interface***'.
- Interface cannot contain fields because they represent a particular implementation of data.
- *Multiple inheritance* is possible with the help of Interfaces but not with classes.

Syntax for Interface Declaration:

```
interface  <interface_name >
{
    // declare methods
    // declare properties
    // declare Events
    // declare indexers
```

```
}
```

Syntax for Implementing Interface:

```
class class_name : interface_name
{
        //Implementation of interface members
        // other class members
}
```

// C# program to demonstrate interface

```
using System;

// A simple interface
interface Inter1
{
        // method having only declaration not definition
        void display();
}

// A class that implements interface.
class testClass :  Inter1
{

        // providing the body part of function
        public void display()
        {
                Console.WriteLine("tumkur");
        }

        // Main Method
        public static void Main (String []args)
        {

                // Creating object
                testClass t = new testClass();

                // calling method
                t.display();
        }
}
```

Output:

Tumkur

Properties:

- A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field.
- Properties can be used as if they are public data members,
- They are actually special methods called *accessors*.
- This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Syntax:

```
<access_modifier> <return_type> <property_name>
{
   get
   {
      // return property value
   }
   set
   {
      // set a new value
   }
}
```

Ex.

```
public class Student
{
   private string name;

// Property for Name
   public string Name
   {
      get
      {
         return name;
      }
      set
      {
         name = val;
      }
   }
}

class Tes {
  public static void Main(string[] args)
   {
     Student std = new Student ();
      std.Name = " Karlos Ray ";
      Console.WriteLine(" name is: " +std.Name);
   }
}
```

Behaviour of properties

- Read and Write Properties: If your C# property holds both the get and the set methods.
- Read-Only Properties: If your C# property holds only get method defined in it.
- Write Only Properties: If your C# property holds only set method defined in it.

C# Exception Handling

- Exception Handling is *a process to handle runtime errors*.
- Performing exception handling so that normal flow of the application can be maintained even after runtime errors.
- In C#, exception is an event or object which is thrown at runtime.
- All exceptions the derived from ***System.Exception*** class. It is a runtime error which can be handled.
- If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It *maintains the normal flow* of the application. In such case, rest of the code is executed event after exception.

C# Exception Handling Keywords

In C#, we use 4 keywords to perform exception handling:

- try
- catch
- finally, and
- throw

Syntax:

```
try
{
  // statements causing exception
}

catch( ExceptionName e1 )
{
  // error handling code
}

 catch( ExceptionName e2 )
{
  // error handling code
}

 catch( ExceptionName eN )
 {
  // error handling code
}

 finally
{
  // statements to be executed
}
```

- **try** − A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.

- **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.
- **finally** − The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.
- **throw** − A program throws an exception when a problem shows up. This is done using a throw keyword.

Ex.

```
using System;
    public class ExExample
    {
       public static void Main(string[] args)
       {
          int a = 10, b = 0,c;
             try
             {
                c = a / b;
             }
             catch (Exception e)
             {
                Console.WriteLine(e);
             }
             finally
             {
                Console.WriteLine("finally : exception raises or not but i will be executed");
             }
             Console.WriteLine("out of exception scope");
             Console.ReadKey();
    }
```

Output:

System.DivideByZeroException: Attempted to divide by zero.
Rest of the code

There are two types exception:

1. Built in
2. User defines.

Built in exception.

Exception already defined in the library to use by the programmer/user is known as built in exception.

Some of the following built in exception.

Exception	Description
System.DivideByZeroException :	handles the error generated by dividing a number with zero.
System.NullReferenceException :	handles the error generated by referencing the null object.
System.InvalidCastException	handles the error generated by invalid typecasting.
System.IO.IOException	handles the Input Output errors.

System.FieldAccessException handles the error generated by invalid private or protected field access.

Ex. Refer above program i.e divide by zero (It throws **system.DivideByZeroException**)

User defined Exception.

Exception are defined by the user to meet his requirement is known as user defined exception.

C# allows us to create user-defined or custom exception. It is used to make the meaningful exception. To do this, we need to inherit **Exception** class.

Example :

```
using System;

//User defined exception
public class InvalidAgeException : Exception
{
   public InvalidAgeException(String message)
      : base(message)
   {

   }
}

public class TestUserDefinedException
{
   static void validate(int age)
   {
      if (age < 18)
      {
         throw new InvalidAgeException("Sorry, Age must be greater than 18");
      }
   }
   public static void Main(string[] args)
   {
      try
      {
         validate(12);
      }
      catch (InvalidAgeException e)
      {
            Console.WriteLine(e);
       }
      Console.WriteLine("Rest of the code");
   }
}
```

Output:
InvalidAgeException: Sorry, Age must be greater than 18
Rest of the code

Multithreading

- **Multi-threading** is a process which contains multiple threads within a single process. Here each thread performs different activities.
- Multitasking is the simultaneous execution of multiple tasks or processes over a certain time interval.
- A thread is a lightweight process, or in other words, a thread is a unit which executes the code under the program.
- Every program by default carries one thread to executes the logic of the program and the thread is known as the *Main Thread*, so every program or application is by default single threaded model.
- This single-threaded model has a drawback. The single thread runs all the process present in the program in synchronizing manner, means one after another. So, the second process waits until the first process completes its execution, it consumes more time in processing.
- To use multithreading concept we have to include **System.Threading** namespace in the program.

Windows operating system is an example of multitasking because it is capable of running more than one process at a time like running Google Chrome, Notepad, VLC player etc. at the same time. The operating system uses a term known as an *process* to execute all these applications at the same time.

Example : program to illustrate the concept of single threaded model

```
using System;
using System.Threading;

public class TestThread {

	// static method one
	public static void method1()
	{

		// It prints numbers from 0 to 10
		for (int I = 0; I <= 5; I++) {

			Console.WriteLine("Method1 is : {0}", I);

			// When the value of I is equal to 3 then this method sleeps for
			// 6 seconds and after 6 seconds it resumes its working
			if (I == 3) {
				Thread.Sleep(6000);
			}
		}
	}

	// static method two
	public static void method2()
	{

		// It prints numbers from 0 to 10
		for (int J = 0; J <= 5; J++) {
```

```
                        Console.WriteLine("Method2 is : {0}", J);
                }
        }
}

// Driver Class
public class GFG {

        // Main Method
        static public void Main()
        {

                // Calling static methods
                TestThread.method1();
                TestThread.method2();
        }
}
```

Output:

```
Method1 is : 0
Method1 is : 1
Method1 is : 2
Method1 is : 3
Method1 is : 4
Method1 is : 5
Method2 is : 0
Method2 is : 1
Method2 is : 2
Method2 is : 3
Method2 is : 4
Method2 is : 5
```

Explanation:

Here, first of all, *method1* executes. In *method1* , *for* loop starts from 0 when the value of i is equal to 3 then the method goes into sleep for 6 seconds and after 6 seconds it resumes its process and print remaining value. Until *method2* is in the waiting state. *method2* start its working when *method1* complete its assigned task. So to overcome the drawback of single threaded model multithreading is introduced.

Here we have a class and this call contains two different methods, now using multithreading each method is executed by a separate thread. So the major advantage of multithreading is it works simultaneously, means multiple tasks executes at the same time. And also maximizing the utilization of the CPU because multithreading works on time-sharing concept mean each thread takes its own time for execution and does not affect the execution of another thread, this time interval is given by the operating system.

Example : program to illustrate the concept of multithreading

```
using System;
using System.Threading;

public class TetstMthread {

        // static method one
```

```
	public static void method1()
	{

		// It prints numbers from 0 to 5
		for (int I = 0; I <= 5; I++) {
			Console.WriteLine("Method1 is : {0}", I);

			// When the value of I is equal to 3 then this method sleeps for 6 seconds
			if (I == 3) {
				Thread.Sleep(6000);
			}
		}
	}

	// static method two
	public static void method2()
	{
		// It prints numbers from 0 to 5
		for (int J = 0; J <= 5; J++) {
			Console.WriteLine("Method2 is : {0}", J);
		}
	}

	// Main Method
	static public void Main()
	{

		// Creating and initializing threads
		Thread thr1 = new Thread(method1);
		Thread thr2 = new Thread(method2);
		thr1.Start();
		thr2.Start();
	}
}
```

Output :

```
Method1 is : 0
Method1 is : 1
Method1 is : 2
Method1 is : 3
Method2 is : 0
Method2 is : 1
Method2 is : 2
Method2 is : 3
Method2 is : 4
Method2 is : 5
Method1 is : 4
Method1 is : 5
```

Explanation: Here, we create and initialize two threads, i.e *thr1* and *thr2* using Thread class. Now using thr1.Start(); and thr2.Start(); we start the execution of both the threads. Now both thread runs simultaneously and the processing of *thr2* does not depend upon the processing of *thr1* like in the single threaded model.

Note: Output may vary due to context switching.

Advantages of Multithreading:

- It executes multiple process simultaneously.
- Maximize the utilization of CPU resources.
- Time sharing between multiple process.