Inheritance

Creating a new class from existing class is called as inheritance. When a new class needs same members as an existing class then instead of creating those members again in new class, the new class can be created from existing class, which is called as inheritance. Main advantage of inheritance is reusability of the code. During inheritance, the class that is inherited is called as base class and the class that does the inheritance is called as derived class and every non private member in base class will become the member of derived class. To perform inheritance, use the following syntax.

[Access Modifier] class ClassName : baseclassname

{

}

Example : The following example creates a class with the name Base that contains the logic for accepting and printing two integers and another class Derived is created by inheriting the class Base as it requires the logic for accepting and printing three integers.


```
namespace Inheritance
{
    class Base
    {
        int A, B;
        public void GetAB()
        {
            Console.Write("Enter Value For A : ");
            A = int.Parse(Console.ReadLine());
            Console.Write("Enter Value For B : ");
            B = int.Parse(Console.ReadLine());
        }
        public void PrintAB()
        {
            Console.Write("A = {0}\tB = {1}\t", A, B);
        }
    }
    class Derived : Base
    {
        int C;
        public void Get()
        {
            GetAB();
            Console.Write("Enter Value For C : ");
            C = int.Parse(Console.ReadLine());
        }
        public void Print()
        {
            PrintAB();
            Console.WriteLine("C = {0}", C);
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            Derived D = new Derived();
            D.Get();
            D.Print();
        }
    }
}
```

Constructors, Destructor And Inheritance

During inheritance, base class may also contain constructor and destructor. In this case if you create an instance for the derived class then base class constructor will also be invoked and when derived instance is destroyed then base destructor will also be invoked and the order of execution of constructors will be in the same order as their derivation and order of execution of destructors will be in reverse order of their derivation.

Example : The following example creates a class with the name Base1 and another class with the name Derived1 inherited from Base1. Base1 contains default constructor and a destructor and Derived1 also contains a default constructor and a destructor.

```
namespace Inheritance
{
    class Base1
    {
        public Base1()
        {
            Console.WriteLine("Base Class Constructor");
        }
        ~Base1()
        {
            Console.WriteLine("Base Class Destructor");
        }
    }
    class Derived1 : Base1
    {
        public Derived1()
        {
            Console.WriteLine("Derived Class Constructor");
        }
        ~Derived1()
        {
            Console.WriteLine("Derived Class Destructor");
        }
    }
    class ConstructorInheritance
    {
        static void Main()
        {
            Derived1 D = new Derived1();
        }
    }
}
```

During inheritance, if the base class contains only parameterized constructor, then derived class must contain a parameterized constructor even it doesn't need one. In this case while creating parameterized constructor in derived class, you must specify the parameters required for derived class along with the parameters required for base class constructor and to pass arguments to base class constructor, use the keyword **"base"** at the end of parameterized constructor declaration in derived class preceded with **":"**.

Example : The following example creates two classes Base2 and Derived2 where Derived2 is inherited from Base2 and Base2 contains a parameterized constructor. Hence derived2 must also contain a parameterized constructor even it doesn't need one.

```
namespace Inheritance
{
    class Base2
    {
        int A, B;
        public Base2(int X, int Y)
        {
            A = X;
            B = Y;
        }
        public void PrintAB()
        {
            Console.Write("A = {0}\tB = {1}\t", A, B);
        }
    }
    class Derived2 : Base2
    {
        int C;
        public Derived2(int X, int Y, int Z)
            : base(X, Y)
        {
            C = Z;
        }
        public void Print()
        {
            PrintAB();
            Console.WriteLine("C = {0}", C);
        }
    }
    class BaseParamConstructor
    {
        static void Main()
        {
            Derived2 D = new Derived2(10, 20, 30);
            D.Print();
        }
    }
}
```

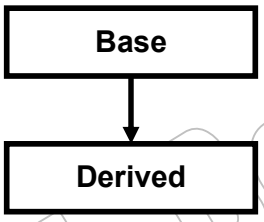
Example : The following example creates two classes Account and Interestaccount where IntersetAccount is inherited from Account class.

```
namespace Inheritance
{
    class Account
    {
        protected double Balance;
        public Account(double Amount)
        {
            Balance = Amount;
        }
        public Account()
        {
            Balance = 0.0;
        }
        public void Deposit(double Amount)
        {
            Balance += Amount;
        }
        public void Withdraw(double Amount)
        {
            if (Amount <= Balance)
            {
                Balance -= Amount;
            }
            else
            {
                Console.WriteLine("Sufficient Amount Not Available");
            }
        }
        public double GetBalance()
        {
            return Balance;
        }
    }
    class InterestAccount : Account
    {
        double Default_InterestRate = 5.5;
        double InterestRate;
        public InterestAccount(double Amount, Double Interest) :
                                                            base(Amount)
        {
            InterestRate = Interest;
        }
        public InterestAccount(double Amount)
            : base(Amount)
        {
            InterestRate = Default_InterestRate;
        }
        public InterestAccount()
        {
            InterestRate = Default_InterestRate;
        }
        public void Add_Monthly_Interest()
        {
            Balance = Balance + (Balance * InterestRate / 100) / 12;
        }
    }
```

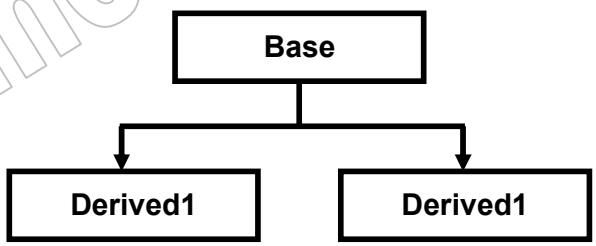
```
    class BankAccount
    {
        static void Main()
        {
        InterestAccount A = new InterestAccount(10000);
        A.Deposit(500.00);
        Console.WriteLine("Balance After Deposit : {0}", A.GetBalance());
        A.Withdraw(1000.00);
        Console.WriteLine("Balance After WithDraw : {0}", A.GetBalance());
        A.Add_Monthly_Interest();
        Console.WriteLine("Balance After Adding Interest : {0}",
                                                    A.GetBalance());
        }
    }
}
```

Types Of Inheritance

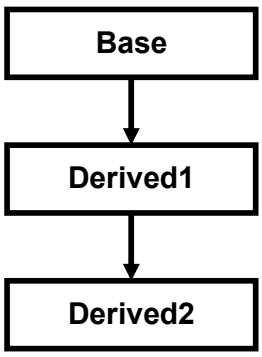
Single Inheritance : when a single derived class is created from a single base class then the inheritance is called as single inheritance.



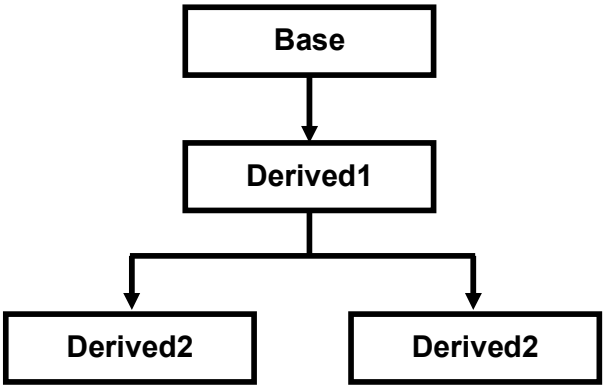
Hierarchical Inheritance : when more than one derived class are created from a single base class, then that inheritance is called as hierarchical inheritance.



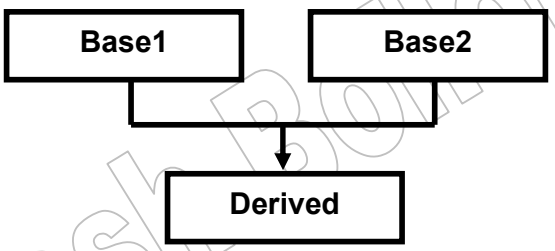
Multi Level Inheritance : when a derived class is created from another derived class, then that inheritance is called as multi level inheritance.



Hybrid Inheritance : Any combination of single, hierarchical and multi level inheritances is called as hybrid inheritance.



Multiple Inheritance : when a derived class is created from more than one base class then that inheritance is called as multiple inheritance. But multiple inheritance is not supported by .net using classes and can be done using interfaces.



Handling the complexity that causes due to multiple inheritance is very complex. Hence it was not supported in dotnet with class and it can be done with interfaces.

Method Overloading

The process of creating more than one method in a class with same name or creating a method in derived class with same name as a method in base class is called as method overloading. In VB.net when you are overloading a method of the base class in derived class, then you must use the keyword **"Overloads".** But in C# no need to use any keyword while overloading a method either in same class or in derived class. While overloading methods, a rule to follow is the overloaded methods must differ either in number of arguments they take or the data type of at least one argument.

Example : The following example creates two classes Class1 and Class2 where Class2 is inherited from Class1 and it overloads the method Sum available in Class1.

```
namespace Inheritance
{
    class Class1
    {
        public int Sum(int A, int B)
        {
            return A + B;
        }
    }
    class Class2 : Class1
    {
        public int Sum(int A, int B, int C)
        {
            return A + B + C;
        }
    }
    class MethodOL
    {
        static void Main()
        {
            Class2 C = new Class2();
            Console.WriteLine(C.Sum(10, 20));
            Console.WriteLine(C.Sum(10, 20, 30));
        }
    }
}
```

Example : The following example creates two classes with the name Maths1 and Maths2 where Maths2 is inherited from Maths1 and overloads a method Sum of Maths1.


```
namespace Inheritance
{
    class Maths1
    {
        public int Sum(params int[] A)
        {
            int S = 0;
            foreach (int n in A)
            {
                S += n;
            }
            return S;
        }
    }
    class Maths2 : Maths1
    {
        public float Sum(params float[] A)
        {
            float S = 0;
            foreach (float n in A)
            {
                S += n;
            }
            return S;
        }
    }
    class MethodOL1
    {
```

```
        static void Main()
        {
            Maths2 M = new Maths2();
            int[] A = { 23, 44, 56, 71, 90 };
            float[] F = { 34.23F, 17.23F, 19.99F, 14.76F, 21.21F };
            Console.WriteLine(M.Sum(A));
            Console.WriteLine(M.Sum(F));
        }
    }
}
```

Method overloading provides more than one form for a method. Hence it is an example for polymorphism. In case of method overloading, compiler identifies which overloaded method to execute based on number of arguments and their data types during compilation it self. Hence method overloading is an example for compile time polymorphism.

Method Overriding

Creating a method in derived class with same signature as a method in base class is called as method overriding. Same signature means methods must have same name, same number of arguments and same type of arguments. Method overriding is possible only in derived classes, but not within the same class. When derived class needs a method with same signature as in base class, but wants to execute different code than provided by base class then method overriding will be used. To allow the derived class to override a method of the base class, C# provides two options, **virtual methods** and **abstract methods.**

Virtual Methods

When you want to allow a derived class to override a method of the base class, within the base class method must be created as **virtual** method and within the derived class method must be created using the keyword **override.** When a method declared as virtual in base class, then that method can be defined in base class and it is optional for the derived class to override that method. When it needs same definition as base class, then no need to override the method and if it needs different definition than provided by base class then it must override the method.

Example : The following example creates three classes shape, circle and rectangle where circle and rectangle are inherited from the class shape and overrides the methods Area() and Circumference() that are declared as virtual in Shape class.

```
namespace Inheritance
{
    class Shape
```

```
    {
        protected float R, L, B;
        public virtual float Area()
        {
            return 3.14F * R * R;
        }
        public virtual float Circumference()
        {
            return 2 * 3.14F * R;
        }
    }
    class Rectangle : Shape
    {
        public void GetLB()
        {
            Console.Write("Enter Length : ");
            L = float.Parse(Console.ReadLine());
            Console.Write("Enter Breadth : ");
            B = float.Parse(Console.ReadLine());
        }
        public override float Area()
        {
            return L * B;
        }
        public override float Circumference()
        {
            return 2 * (L + B);
        }
    }
    class Circle : Shape
    {
        public void GetRadius()
        {
            Console.Write("Enter Radius : ");
            R = float.Parse(Console.ReadLine());
        }
    }
    class VirtualMethods
    {
        static void Main()
        {
            Rectangle R = new Rectangle();
            R.GetLB();
            Console.WriteLine("Area : {0}", R.Area());
            Console.WriteLine("Circumference : {0}", R.Circumference());

            Console.WriteLine();

            Circle C = new Circle();
            C.GetRadius();
            Console.WriteLine("Area : {0}", C.Area());
            Console.WriteLine("Circumference : {0}", C.Circumference());
        }
    }
}
```

Method overriding also provides more than one form for a method. Hence it is also an example for polymorphism. When overridden methods are directly called using derived object as in previous example, then it will be an example for compile time polymorphism. Because based on the type of derived object, during compilation it self compiler identifies which overridden method it has to execute.

Abstract Methods And Abstract Classes

Another way for method overriding is abstract methods. There may be a situation where it is not possible to define a method in base class and every derived class must override that method. In this situation **abstract methods** are used. When a method is declared as abstract in the base class then it is not possible to define it in the base class and every derived class of that class must provide its own definition for that method. When a class contains at least one abstract method, then the class must be declared as **abstract class.** When a class is declared as abstract class, then it is not possible to create an instance for that class. But it can be used as a parameter in a method.

Example : The following example creates three classes shape, circle and rectangle where circle and rectangle are inherited from the class shape and overrides the methods Area() and Circumference() that are declared as abstract in Shape class and as Shape class contains abstract methods it is declared as abstract class.


```
namespace Inheritance
{
    abstract class Shape1
    {
        protected float R, L, B;
        public abstract float Area();
        public abstract float Circumference();
    }
    class Rectangle1 : Shape1
    {
        public void GetLB()
        {
            Console.Write("Enter Length : ");
            L = float.Parse(Console.ReadLine());
            Console.Write("Enter Breadth : ");
            B = float.Parse(Console.ReadLine());
        }
        public override float Area()
        {
            return L * B;
        }
        public override float Circumference()
        {
            return 2 * (L + B);
        }
    }
```

```
class Circle1 : Shape1
{
    public void GetRadius()
    {
        Console.Write("Enter Radius : ");
        R = float.Parse(Console.ReadLine());
    }
    public override float Area()
    {
        return 3.14F * R * R;
    }
    public override float Circumference()
    {
        return 2 * 3.14F * R;
    }
}
class AbstractMethods
{
    public static void Calculate(Shape1 S)
    {
        Console.WriteLine("Area : {0}", S.Area());
        Console.WriteLine("Circumference : {0}", S.Circumference());
    }
    static void Main()
    {
        Rectangle1 R = new Rectangle1();
        R.GetLB();
        Calculate(R);

        Console.WriteLine();

        Circle1 C = new Circle1();
        C.GetRadius();
        Calculate(C);
    }
}
```


In the above example method calculate takes a parameter of type Shape1 from which rectangle1 and circle1 classes are inherited. A base class type parameter can take derived class object as an argument. Hence the calculate method can take either rectangle1 or circle1 object as argument and the actual argument in the parameter S will be determined only at runtime and hence this example is an example for runtime polymorphism.

Polymorphism

Polymorphism means having more than one form. Overloading and overriding are used to implement polymorphism. Polymorphism is classified into **compile time polymorphism or early binding or static binding** and **Runtime polymorphism or late binding or dynamic binding.**

The polymorphism in which compiler identifies which polymorphic form it has to execute at compile time it self is called as **compile time polymorphism or early binding**. Advantage of early binding is execution will be fast. Because every thing about the method is known to compiler during compilation it self and disadvantage is lack of flexibility. Examples of early binding are overloaded methods, overloaded operators and overridden methods that are called directly by using derived objects.

The polymorphism in which compiler identifies which polymorphic form to execute at runtime but not at compile time is called as **runtime polymorphism or late binding.** Advantage of late binding is flexibility and disadvantage is execution will be slow as compiler has to get the information about the method to execute at runtime. Example of late binding is overridden methods that are called using base class object.

Sealed Methods

The virtual nature of a method persists for any number of levels of inheritance. For example there is a class **"A"** that contains a virtual method **"M1".** Another class **"B"** is inherited from **"A"** and another class **"C"** is inherited from **"B". I**n this situation class **"C"** can override the method **"M1"** regardless of whether class **"B"** overrides the method **"M1".** At any level of inheritance, if you want to restrict next level of derived classes from overriding a virtual method, then create the method using the keyword **sealed** along with the keyword **override.**

```
[Access Modifier] sealed override returntype methodname([Params])
{

}
```

Sealed Classes

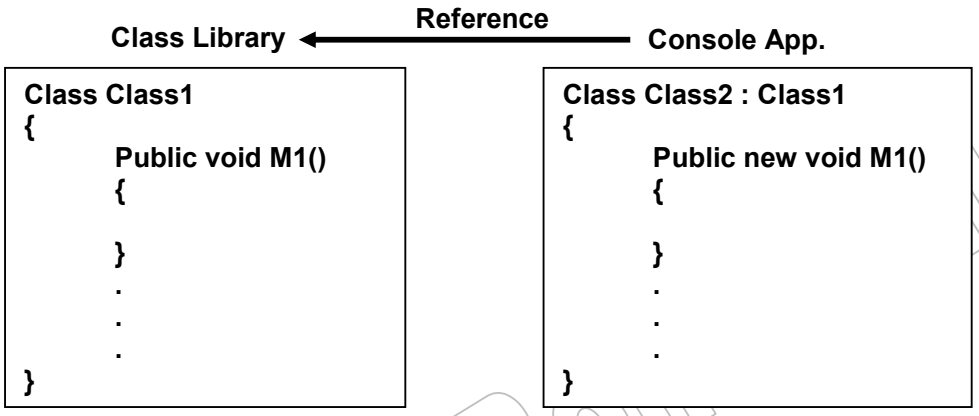
When you want to restrict your classes from being inherited by others you can create the class as sealed class. To create a class as sealed class, create the class using the keyword **sealed.**

```
[Access Modifier] sealed class classname
{

}
```

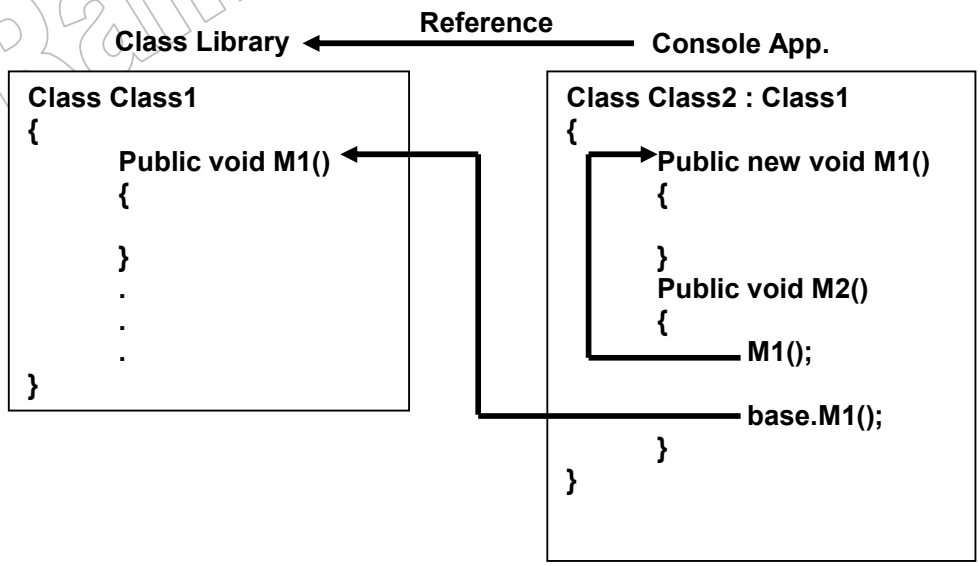
New

There may be a situation where you are creating a class in your application that is inherited from a class available in a class library referred by your application and you want to override a method of the base class in derived class and that method was not declared as virtual or abstract in the base class. In this situation you can hide the method of base class by creating a method in derived class with same signature as in base class by using the keyword **new**.



base

When you want to invoke a method of the base class that is overridden or hidden by the derived class then use the keyword **base.**



Comparison Between C#.Net And VB.Net in Keywords Related to Inheritance And Polymorphism

C#.Net	VB.Net
Virtual Methods	Overridable Methods
Abstract Methods	MustOverride Methods
Abstract Classes	MustInherit Classes
Override	Overrides
Sealed Methods	NotOverridable Methods
Sealed Classes	NotInheritable Classes
New	Shadows
Base	MyBase

Interfaces

An interface is a user defined type that contains only declarations of properties, methods and events and not implementation. Main purpose of interfaces is to provide support for multiple inheritance, which is not possible using classes. A class can implement more than one interface and when a class implements an interface then it must provide definition for each and every member of the interface. An interface is created using the keyword **interface** and has the following syntax.

[Access Modifier] interface interfacename
{

}

An interface has the following restrictions.

1. It can not contain fields.
2. It can contain only member declarations and not definition.
3. It can not contain static members.
4. It is not possible to specify access modifier for the members and by default members are public.

To implement an interface in a class, syntax is same as inheriting a class and is as follows.

[Access Modifier] class classname : interface1, interface2,…
{

}

Example : The following example creates two interfaces interface1 and interface2 that implemented in a class with the name Test

```
namespace Inheritance
{
    interface interface1
    {
        void M1();
    }
    interface interface2
    {
        void M2();
    }
```

```
    class Test : interface1, interface2
    {
        public void M1()
        {
            Console.WriteLine("M1 Of Interface1");
        }
        public void M2()
        {
            Console.WriteLine("M2 Of Interface2");
        }
    }
    class interfaces
    {
        static void Main()
        {
            Test T = new Test();
            T.M1();
            T.M2();
        }
    }
}
```

Members With Same Signature

There may be a situation where two interfaces contain a member with same signature and those two interfaces are implemented in the same class. In this situation to differentiate the definition of those members in class, you must qualify the member name with interface name and in this case no need to specify access modifier for this member even in class. To call these members first you have to create an instance for the class and then type cast the instance to interface, whose member you want to call.

Example : The following example creates two interfaces interface1 and interface2 where both contain a method M1 with same signature and are implemented in a class with the name Test.

```
namespace Inheritance
{
    interface interface3
    {
        void M1();
    }
    interface interface4
    {
        void M1();
    }
    class Test1 : interface3, interface4
    {
        void interface3.M1()
        {
            Console.WriteLine("M1 Of interface3");
        }
```

```
        void interface4.M1()
         {
             Console.WriteLine("M1 Of interface4");
         }
    }
    class interfacesamesign
    {
        static void Main()
        {
            Test1 T = new Test1();
            interface3 i3 = (interface3)T;
            i3.M1();

            interface4 i4 = (interface4)T;
            i4.M1();
        }
    }
}
```

Inheriting a Class And Implementing Interfaces

There may be a situation where you have to create a class by inheriting another class and at the same time implementing one or more interfaces. In this situation you must specify the base class name first and then interface names.

[Access Modifier] class ClassName : BaseClassName, interface1, interface2, …

{

}

Differences Between Abstract Class And Interface

Abstract Class	Interface
It can contain Fields	It can not contain fields
It can contain members with definition when they are not declared as abstract.	It can not contain definition for any member
You can specify access modifier for the members.	It is not possible to specify an access modifier for the members.
By default members are private.	By default members are public.
It can contain static members	It can not contain static members.
It can contain constructors and destructor	It can not contain constructors and destructor.
It doesn't support multiple inheritance.	It supports multiple inheritance.

Operator overloading

The operators available in C# will work only on standard data types like int, float and string. But not on user defined types like class and structure. Making an operator work on user defined type like class is called as operator overloading. To overload an operator, you have to create an operator method in the class. Operator methods are created using the keyword operator and they must be public and static.

Public static returntype operator #([Parameters])

{

}

Example : The following example creates a class Complex that represents a complex number and overload the + operator to perform addition of two complex objects using +.

```
namespace Inheritance
{
    class Complex
    {
        float R, I;
        public void Read()
        {
            Console.Write("Enter Imaginary : ");
            I = float.Parse(Console.ReadLine());
            Console.Write("Enter Real : ");
            R = float.Parse(Console.ReadLine());
        }
        public void Print()
        {
            Console.WriteLine("{0}i + {1}", I, R);
        }
        public static Complex operator +(Complex C1, Complex C2)
        {
            Complex T = new Complex();
            T.I = C1.I + C2.I;
            T.R = C1.R + C2.R;
            return T;
        }
    }
    class PlusOL
    {
        static void Main()
        {
            Complex C1 = new Complex();
            Complex C2 = new Complex();
            Complex C3;
            C1.Read();
            C2.Read();
```

```
            C3 = C1 + C2;
            Console.Clear();
            C1.Print();
            C2.Print();
            Console.WriteLine("----------------");
            C3.Print();
        }
    }
}
```

Object Comparison

In .net when you want to compare two objects for equality, then objects may be **value equivalent** or **reference equivalent.** Two objects are said to be value equivalent, if and only if they both are of same type and contain same values. To compare two objects for value equivalent, **Equals()** method of **object** class is used. By default every class in .net is inherited from **object** class. Equals() method is a virtual method in object class and you have to override that method in your class to use it for value equivalent comparison of objects of your class. When you want to compare two objects **obj1** and **obj2** for value equivalent, then syntax will be as follows.

Obj1.Equals(Obj2)

Two objects are said to be reference equivalent, if and only if they both are of same type and contain same reference. When two objects are reference equivalent, then they are automatically reference equivalent. To compare two objects for reference equivalent, **ReferenceEquals()** method of **object** class is used. ReferenceEquals() method is a static method in object class. To compare two objects **obj1** and **obj2** for reference equivalent, syntax is as follows.

Object.ReferenceEquals(obj1,obj2)

Example : The following example creates a class Test that contains a float field and overloads the operator > and < used for comparison of objects of the class Test.

```
namespace Inheritance
{
    class Test2
    {
        float F;
        public void Read()
```

```
        {
            Console.Write("Enter Value For F : ");
            F = float.Parse(Console.ReadLine());
        }
        public override bool Equals(object obj)
        {
            Test2 T = (Test2)obj;
            if (this.F == T.F)
                return true;
            else
                return false;
        }
        public override int GetHashCode()
        {
            return base.GetHashCode();
        }
        public static bool operator ==(Test2 T1, Test2 T2)
        {
            if (object.ReferenceEquals(T1, T2))
            {
                return true;
            }
            if (T1.Equals(T2))
            {
                return true;
            }
            return false;
        }
        public static bool operator !=(Test2 T1, Test2 T2)
        {
            return !(T1 == T2);
        }
    }
    class GTOL
    {
        static void Main()
        {
            Test2 T1 = new Test2();
            Test2 T2 = new Test2();
            Test2 T3 = T2;
            T1.Read();
            T2.Read();

            if (T1 == T2)
                Console.WriteLine("T1 And T2 Are Equal");
            else
                Console.WriteLine("T1 And T2 Are Not Equal");

            if (T2 == T3)
                Console.WriteLine("T2 And T3 Are Equal");
        }
    }
}
```

The following is the list of operators that can be overloaded and that can't be overloaded.

Operators	Overloadability
+, -, *, /, %, &, |, <<, >>	All C# binary operators can be overloaded.
+, -, !, ~, ++, --, true, false	All C# unary operators can be overloaded.
==, !=, <, >, <= , >=	All relational operators can be overloaded, but only as pairs.
&&, ||	They can't Be Overloaded
() , (Conversion operator)	They can't Be Overloaded
+=, -=, *=, /=, %=	These compound assignment operators can be overloaded. But in C#, these operators are automatically overloaded when the respective binary operator is overloaded.
=, . , ?:, ->, new, is, as, sizeof	They can't Be Overloaded