# Interfaces, Generics and Operator Overloading in C#

## What is an interface in C# ?

In C#, an interface is similar to abstract class. However, unlike abstract classes, all methods of an interface are fully abstract (method without body).

The interface is a fully abstract class and we cannot create objects for the same.

We use the interface keyword to create an interface. For example,

```
interface Shape
{
    void calculateArea();
    void calculatePerimeter();
}
```

Here,

- Shape is the name of the interface.
- The methods calculateArea() and calculatePerimeter() are abstract methods which does not have method implementations.
- All the implementing classes from the above interface are required to provide the implementation of those methods.
- We cannot use access modifiers inside an interface.
- All members of an interface are public by default.
- An interface doesn't allow fields.

## Implementing an Interface

1. We cannot create objects of an interface.
2. To use an interface, other classes must implement it.
3. Same as in C# Inheritance, we use : symbol to implement an interface.

For example,

```
class Rectangle : Shape
{
    public void calculateArea()
    {
        //Logic
    }
    public void calculatePerimeter()
    {
        //Logic
    }
}
```

Here, the Rectangle class implements Shape. And, provides the implementation of the calculateArea() and calculatePerimeter() methods.

**Note:** We must provide the implementation of all the methods of interface inside the class that implements it.

**Accessing Interface Methods**

We can use the reference variable of an interface to access the interface methods which enable the Run Time Polymorphism.

For example,

```
  class Example
    {
        public static void Main(string[] args)
        {
            Shape obj = null;
            obj = new Rectangle();
            obj.calculateArea();
            obj.calculatePerimeter();
        }
    }
```

Notice, we have used the reference variable **obj** of interface Shape. It points to the class Rectangle that implements it.

Though we cannot create objects of an interface, we can still use the reference variable of the interface that points to its implemented class.

**Practical Example of Interface**

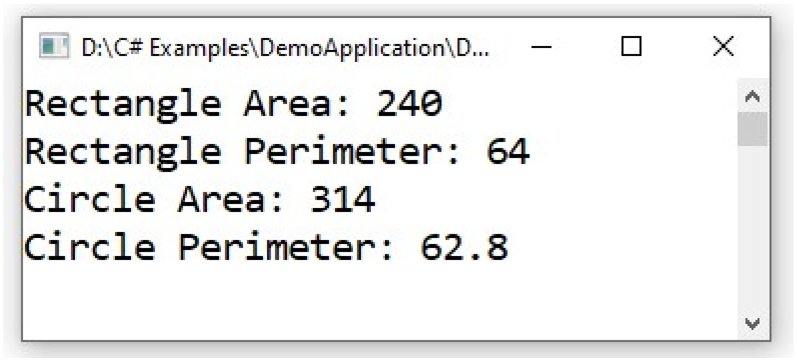
```
using System;
namespace DemoApplication
{
    interface Shape
    {
        void calculateArea();
        void calculatePerimeter();
    }
    class Rectangle : Shape
    {
        int length, breadth;
        public Rectangle(int length, int breadth)
        {
            this.length = length;
            this.breadth = breadth;
        }
        public void calculateArea()
        {
            int area = length * breadth;
            Console.WriteLine("Rectangle Area: " + area);
        }
        public void calculatePerimeter()
        {

            int perimeter = 2 * (length + breadth);
            Console.WriteLine("Rectangle Perimeter: " + perimeter);
        }
    }
    class Circle : Shape
    {
        int radius;
        public Circle(int radius)
        {
           this.radius = radius;
        }
        public void calculateArea()
        {
            double area = 3.14 * radius * radius ;
            Console.WriteLine("Circle Area: " + area);
        }
        public void calculatePerimeter()
        {

            double perimeter = 2 * 3.14 * radius;
            Console.WriteLine("Circle Perimeter: " + perimeter);
        }
    }
    class Example
    {
        public static void Main(string[] args)
        {
            Shape obj = null;

            obj = new Rectangle(12,20);
            obj.calculateArea();
            obj.calculatePerimeter();

            obj = new Circle(10);
            obj.calculateArea();
            obj.calculatePerimeter();
            Console.ReadKey();
        }
    }
}
```



```
Rectangle Area: 240
Rectangle Perimeter: 64
Circle Area: 314
Circle Perimeter: 62.8
```

Here, the reference variable of the Shape interface "obj" act as polymorphic reference. This reference variable calls the method such as **calculateArea()** and calculatePerimeter() based on the object that it points to.

**Implementing Multiple Interfaces (Multiple Inheritance)**

Unlike inheritance, a class can implement multiple interfaces. For example,

```
using System;
namespace DemoApplication
{
    interface Shape
    {
        void calculateArea();
        void calculatePerimeter();
    }
    interface Drawable
    {
        void drawShape();
    }
    class Rectangle : Shape , Drawable
    {
        public void calculateArea()
        {
           // Logic for Area Calculation
        }
        public void calculatePerimeter()
        {
           // Logic for Perimeter calculation
        }
        public void drawShape()
        {
            // Logic for drawing Rectangle Shape
        }
    }
}
```

---

## Generics in C#

Generics introduces the concept of type parameters to C# and .NET, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code.

C# Generics allow us to create a single class or method that can be used with different types of data. This helps us to reuse our code.

Here, we will learn to create generics class and method in C#.

### The Generic Class

A generics class is used to create an instance of any data type. To define a generics class, we use angle brackets (<T>) with Type parameter.

For example,

```
    class Square<T>
    {
       // data declaration with T type parameters.
    }
```

Here, we have created a generics class named Square. T used inside the angle bracket is called the Type parameter.
While creating an object of the Square class, we specify the data type of the object which replaces the type parameter T.

### Create an object of Generics Class

Let's create three instances of the generics class.

```
    Square<int> obj1= new Square<int>();
    Square<double> obj2 = new Square<double>();
    Square<float> obj3 = new Square<float>();
```

Here, we have created three objects named obj1, obj2 and obj3 with data type's int, double and float respectively.

During the time of compilation, the type parameter T of the Student class is replaced by,

- int - for instance obj1
- double - for instance obj2
- float – for instance obj3

**Example 1:** Find the area of Square with one Type parameter

```
using System;
namespace DemoApplication
{
    class Square<T>
    {
        private T side;
        public Square(T side)
        {
            this.side = side;
        }
        public T calculateArea()
        {
            dynamic a = side;
            return a * a;
        }
    }

class Example
 {
   public static void Main(string[] args)
   {
     Square<int> obj1 = new Square<int>(5);
     Console.WriteLine("Square Area(int):" + obj1.calculateArea());

     Square<double> obj2 = new Square<double>(4);
     Console.WriteLine("Square Area(double):" + obj2.calculateArea());

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

Here, T is replaced as int for obj1 and perform square area calculation for integer side length. But for obj2, the T type parameter will be replaced as “double”, hence the square calculation done for double side length.

Note: Also, dynamic keyword must be used for the mathematical operation on Type Parameters as given below. Since the dynamic keyword sets the value at runtime not during compile time.

```
            dynamic a = side;
            return a * a;
```

**Example 2:** Calculator Program with two Type parameters

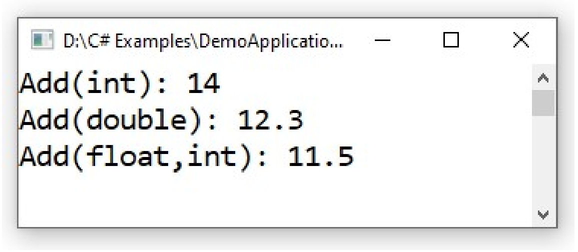
```
using System;
namespace DemoApplication
{
    class Calculator<T,V>
    {
        private T num1;
        private V num2;
        public Calculator(T num1,V num2)
        {
            this.num1 = num1;
            this.num2 = num2;
        }
        public T add()
        {
            dynamic a = num1;
            dynamic b = num2;
            return a + b;
        }
    }

class Example
{
 public static void Main(string[] args)
 {
  Calculator<int,int> obj1 = new Calculator<int,int>(6,8);
  Console.WriteLine("Add(int): " + obj1.add());

  Calculator<double,double> obj2 = new Calculator<double,double>(4.5,7.8);
  Console.WriteLine("Add(double): " + obj2.add());

  Calculator<float, int> obj3 = new Calculator<float, int>(4.5f, 7);
  Console.WriteLine("Add(float,int): " + obj3.add());

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



```
Add(int): 14
Add(double): 12.3
Add(float,int): 11.5
```

The Generics Method

Similar to the generics class, we can also create a method that can be used with any type of data. Such a class is known as the generics Method.

For example,

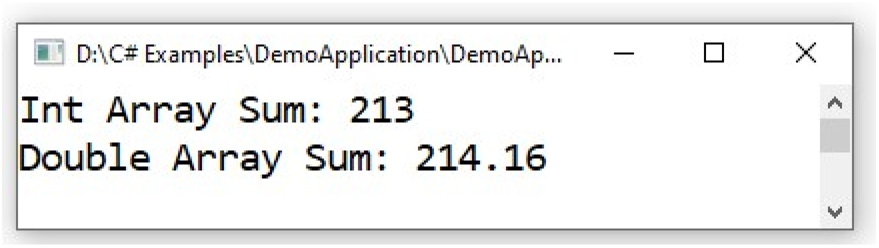
```
      public T arraySum(T[] items)
        {
            // logic goes here.
        }
```

**Example Program:** To find the sum of array elements using Generic Method

```
using System;
namespace DemoApplication
{
    class Calculator<T>
    {
        public T arraySum(T[] items)
        {
            dynamic sum=0;
            foreach(T x in items)
            {
                dynamic a = x;
                sum += a;
            }
            return sum;
        }
    }
    class Example
    {
        public static void Main(string[] args)
        {
         int[] nums = { 10, 23, 12, 78, 90 };
         Calculator<int> obj1 = new Calculator<int>();
         Console.WriteLine("Int Array Sum: " + obj1.arraySum(nums));

         double[] items = { 10.2, 23.1, 12.44, 78.22, 90.2};
         Calculator<double> obj2 = new Calculator<double>();
         Console.WriteLine("Double Array Sum: " + obj2.arraySum(items));

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



```
D:\C# Examples\DemoApplication\DemoAp...
Int Array Sum: 213
Double Array Sum: 214.16
```

## Operator Overloading in C#

Operator overloading gives the ability to use the same operator to do various operations. It provides additional capabilities to C# operators when they are applied to user-defined data class types.

An operator can be overloaded by defining a function to it. The function of the operator is declared by using the operator keyword.

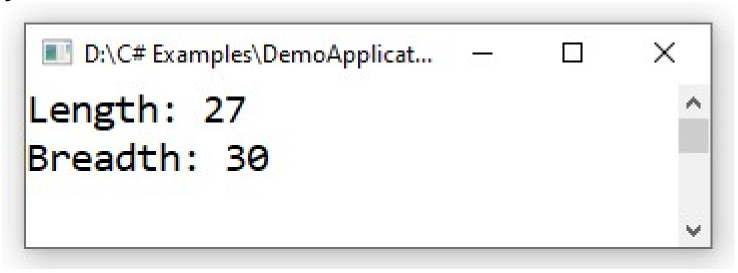
```
access_specifier static className operator operator_symbol (parameters)
 {
     // Code
 }
```

Note: Operator overloading is basically the mechanism of providing a special meaning to an ideal C# operator with respect to a user-defined data classes.

Also, the operator overloading method must be a static method because an operator overload does not reference a class: it takes parameters and returns a value, but it does not have access to the this object or any non-static members of the class.

**Example Program:**

```
using System;
namespace DemoApplication
{
    class Rectangle
    {
        private int length,breadth;
        public Rectangle(int length, int breadth)
        {
            this.length = length;
            this.breadth = breadth;
        }
        public void display()
        {
            Console.WriteLine("Length: " + length);
            Console.WriteLine("Breadth: " + breadth);
        }
        public static Rectangle operator +(Rectangle r1, Rectangle r2)
        {
            int l = r1.length + r2.length;
            int b = r1.breadth + r2.breadth;
            return new Rectangle(l,b);
        }
    }
    class Example
    {
        public static void Main(string[] args)
        {
            Rectangle r1 = new Rectangle(10, 20);
            Rectangle r2 = new Rectangle(17, 10);
            Rectangle r3 = r1 + r2;
            r3.display();
            Console.ReadKey();
        }
    }
}
```



```
Length: 27
Breadth: 30
```