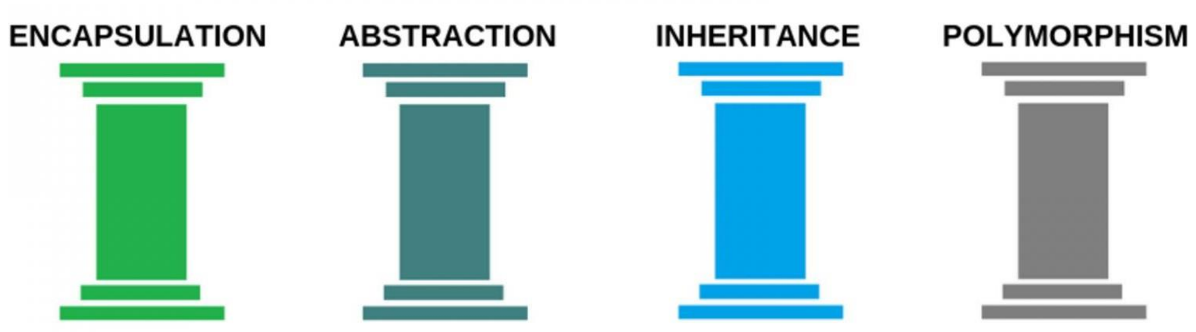
Unit 3

Pillars of OOP

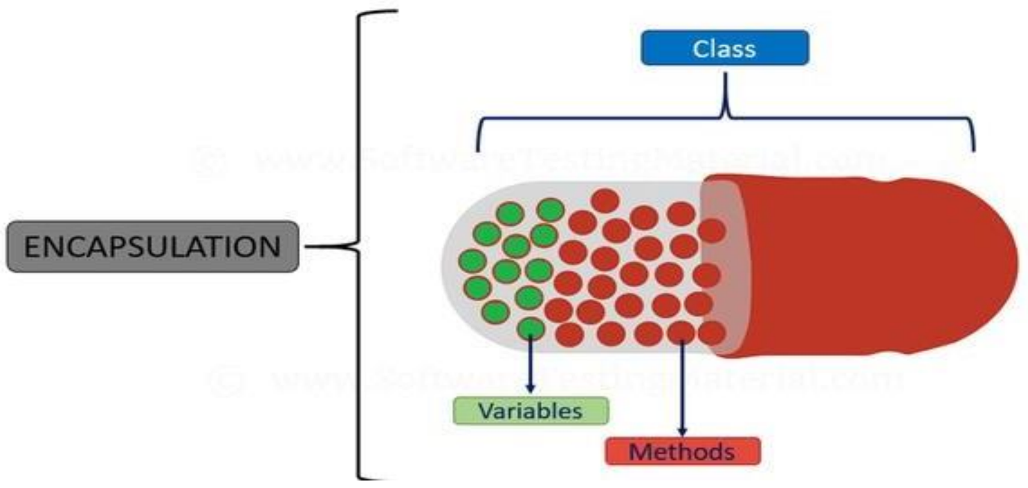
OOP stands for Object-Oriented Programming!

Procedural programming is based on an **unreal world**. Object-oriented programming is based on the **real world**. At starting of our coding life we learn logic building through procedural programming like C Language while for real-world implementation we use OOP like C#, Java, etc.



Encapsulation support

Simply we can say Encapsulation is a process of binding data members and data functions together. Or in another sense, we can say it is a process of binding variables, properties, and methods together.



C# program to illustrate encapsulation

```
using System;
public class DemoEncap
{
  // private variables declared these can only be accessed by public methods of class
  private String studentName;
  private int studentAge;
  // using accessors to get and set the value of studentName
  public String Name
  {
            get
            {
                        return studentName;
            }
            set
            {
                        studentName = value;
            }
  }
  // using accessors to get and set the value of studentAge
  public int Age
  {
            get
            {
                        return studentAge;
            }
            set
            {
                        studentAge = value;
            }
  }
}
```

```
// Driver Class
class Program
{
  // Main Method
  static public void Main()
  {
            DemoEncap obj = new DemoEncap();
            // calls set accessor of the property Name, and pass "Anil" as value of the
            standard field 'value'
            obj.Name = "Anil";
            // calls set accessor of the property Age, and pass "21" as value of the
                standard field 'value'
               obj.Age = 21;
            // Displaying values of the variables
            Console.WriteLine("Name: " + obj.Name);
            Console.WriteLine("Age: " + obj.Age);
       }
}
```

the Name and Age accessors which contain the get and set method to retrieve and set the values of private fields. Accessors are defined as public so that they can access other classes.

Advantages of Encapsulation:

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is stores values in the variables. He only knows that we are passing the values to accessors and that variables are getting initialized to that value.
- **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirements. If we wish to make the variables read-only then we have to only use Get Accessor in the code. If we wish to make the variables write-only then we have to only use Set Accessor.
- **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

Class properties:

Before we start to explain properties, you should have a basic understanding of "**Encapsulation**".

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

- declare fields/variables as private
- provide public get and set methods, through **properties**, to access and update the value of a private field

Properties

private variables can only be accessed within the same class (an outside class has no access to it). However, sometimes we need to access them - and it can be done with properties.

A property is like a combination of a variable and a method, and it has two methods: a get and a set method:

SYNTAX:
```
 Public variable accessor_name;
  {
     get{ }
     set{ }

  }
```

Example
```
class Person
{
  private string name; // field

  public string Name   // property
  {
    get { return name; }   // get method
    set { name = value; }  // set method
  }
}
```

The get method returns the value of the variable name.

The set method assigns a value to the name variable. The value keyword represents the value we assign to the property.

Now we can use the Name property to access and update the private field of the Person class:

```
class Person
{
  private string name; // field
  public string Name   // property
  {
    get { return name; }
    set { name = value; }
  }
}

class Program
{
  static void Main(string[] args)
  {
    Person myObj = new Person();
    myObj.Name = "Liam";
    Console.WriteLine(myObj.Name);
  }
}
```

The output will be:

```
Liam
```

Inheritance support

Deriving classes: **the class which inherits the members of another class is called a derived class** and the class whose members are inherited is called the base class. The derived class is the specialized class for the base class.

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

Advantage of C# Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

C# Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single-level inheritance. Let's see the example of single-level inheritance which inherits the fields only.

```
using System;
   public class Employee
    {
       public float salary = 40000;
    }
   public class Programmer: Employee
   {
       public float bonus = 10000;
   }
   class TestInheritance
    {
       public static void Main(string[] args)
        {
           Programmer p1 = new Programmer();
```

```
            Console.WriteLine("Salary: " + p1.salary);
            Console.WriteLine("Bonus: " + p1.bonus);
        }
    }
```

Output:

```
Salary: 40000
Bonus: 10000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

Let's see another example of inheritance in C# which inherits methods only.

```
using System;
   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...
```

C# Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C#. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C#.

```
using System;
  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();
      }
   }
```

```
Eating...
Barking...
Weeping...
```

Interfaces:

Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.

It is used *to achieve multiple inheritance* which can't be achieved by class. It is used *to achieve fully abstraction* because it cannot have method body.

Its implementation must be provided by class or struct. The class or struct which implements the interface, must provide the implementation of all the methods declared inside the interface.

In C#, an interface can be defined using the interface keyword. An interface can contain declarations of methods, properties, indexers, and events. However, it cannot contain instance fields.

The following interface declares some basic functionalities for the file operations.

Example: C# Interface

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);
}
```

The above declares an interface named IFile. The IFile interface contains two methods, ReadFile() and WriteFile(string) and those are abstract methods.

Note:

- An interface can contain declarations of methods, properties, indexers, and events.
- An interface cannot contain constructors and fields.
- Interface members are by default abstract and public.

Implementing an Interface

A class or a Struct can implement one or more interfaces using colon : . On implementing an interface, you must override all the members of an interface.

}

For example, the following **FileInfo** class implements the **IFile** interface, so it should override all the members of **IFile.**

Example: Interface Implementation

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);
}

class FileInfo : IFile
{
   public void ReadFile()
   {
      Console.WriteLine("Reading File");
   }
   public void WriteFile(string text)
   {
      Console.WriteLine("Writing to file");
   }
}
```

In the above example, the **FileInfo** class implements the **IFile** interface. It overrides all the members of the **IFile** interface with public access modifier. The **FileInfo** class can also contain members other than interface members.

Note

Interface members must be implemented with the public modifier; otherwise, the compiler will give compile-time errors.

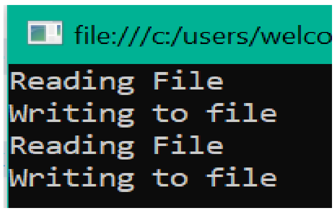
You can create an object of the class and assign it to a variable of an interface type, as shown below.

```
interface IFile
{
   void ReadFile();
   void WriteFile(string text);
}
class FileInfo : IFile
{
        public void ReadFile()
        {
   Console.WriteLine("Reading File");
        }
        public void WriteFile(string text)
        {
   Console.WriteLine("Writing to file");
        }
 }
class Program

{
   static void Main(string[] args)
   {
   IFile file1 = new FileInfo(); //calling base class constructor
   FileInfo file2 = new FileInfo();
   file1.ReadFile();
   file1.WriteFile("content");
   file2.ReadFile();
   file2.WriteFile("content");
   Console.ReadLine();
   }
}
```

}

Output:



```
Reading File
Writing to file
Reading File
Writing to file
```

Above, we created objects of the **FileInfo** class and assign it to **IFile** type variable and **FileInfo** type variable. When interface implemented implicitly, you can access **IFile** members with the **IFile** type variables as well as **FileInfo** type variable.

Non Inheritable classes

Sealed classes are known as non-inheritable classes

Sealed class

Sealed classes are used to restrict the users from inheriting the class. A class can be sealed by using the ***sealed*** keyword. The keyword tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class. The following is the **syntax** of a sealed class :

```
sealed class class_name

{

   // data members

   // methods

}
```

A method can also be sealed, and in that case, the method cannot be overridden. However, a method can be sealed in the classes in which they have been inherited. If you want to declare a method as sealed, then it has to be declared as **virtual** in its base class.

C# sealed class cannot be derived by any class. Let's see an example of sealed class in C#.

```
using System;
sealed public class Animal {
   public void eat() { Console.WriteLine("eating..."); }
}
public class Dog: Animal
{
   public void bark() { Console.WriteLine("barking..."); }
}
public class TestSealed
{
    public static void Main()
    {
        Dog d = new Dog();
        d.eat();
        d.bark();
    }
}
```

Output:

```
Compile Time Error: 'Dog': cannot derive from sealed type 'Animal'
```

Abstract class:

Abstract classes are the way to achieve abstraction in C#. Abstraction in C# is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1. Abstract class
2. Interface

Abstract class and interface both can have abstract methods which are necessary for abstraction.

Abstract Method

A method which is declared abstract and has no body is called abstract method. It can be declared inside the abstract class only. Its implementation must be provided by derived classes. For example:

```
public abstract void draw();
```

In C#, abstract class is a class which is declared abstract. It can have abstract and non-abstract methods. It cannot be instantiated. Its implementation must be provided by derived classes. Here, derived class is forced to provide the implementation of all the abstract methods.

Let's see an example of abstract class in C# which has one abstract method draw(). Its implementation is provided by derived classes: Rectangle and Circle. Both classes have different implementation.

```
using System;
public abstract class Shape
```

```
{
    public abstract void draw();
}
public class Rectangle : Shape
{
    public override void draw()
    {
        Console.WriteLine("drawing rectangle...");
    }
}
public class Circle : Shape
{
    public override void draw()
    {

        Console.WriteLine("drawing circle...");
    }
}
public class TestAbstract
{
    public static void Main()
    {
        Shape s;
        s = new Rectangle();
        s.draw();
        s = new Circle();
        s.draw();
    }
}
```

Output:

```
drawing ractangle...
drawing circle...
```

Interface inheritance

C# allows the user to inherit one interface into another interface. When a class implements the inherited interface then it must provide the implementation of all the members that are defined within the interface inheritance chain.

Important Points:

- If a class implements an interface, then it is necessary to implement all the methods defined by that interface including the base interface methods. Otherwise, the compiler throws an error.
- If both the derived interface and base interface declares the same member then the base interface member name is hidden by the derived interface member name.

Syntax:

```
// declaring an interface
access_modifier interface interface_name
{
  // Your code
}
// inheriting the interface
access_modifier interface interface_name : interface_name
{
   // Your code
}

public interface A
{
        void mymethod1();
        void mymethod2();
}
// The methods of interface A is inherited into interface B
public interface B : A
{
        // method of interface B
        void mymethod3();
}
// Below class is inheriting only interface B This class must implement both interfaces
class Program : B
{
        // implementing the method of interface A
        public void mymethod1()
        {
                Console.WriteLine("Implement method 1");
        }
        // Implement the method of interface A
        public void mymethod2()
        {
                Console.WriteLine("Implement method 2");
        }
        // Implement the method of interface B
        public void mymethod3()
        {
                Console.WriteLine("Implement method 3");
        }
}
// Driver Class
class Program1
{
        static void Main(String []args)
        {
                // creating the object class of Program
                Program obj = new Program();
                // calling the method using object 'obj'
                obj.mymethod1();
                obj.mymethod2();
                obj.mymethod3();
        }
}
```

Output:

```
Implement method 1
Implement method 2
Implement method 3
```

Method Overriding:

If the 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 the **virtual** keyword with base class method and **override** keyword with derived class method.

C# Method Overriding Example

Let's see a simple example of method overriding in C#. In this example, we are overriding the eat() method by the help of override keyword.

```
using System;
public class Animal
{
    public virtual void eat()
   {
        Console.WriteLine("Eating...");
    }
}
public class Dog: Animal
{
    public override void eat()
    {
        Console.WriteLine("Eating  bread...");
    }
}
public class TestOverriding
{
    public static void Main()
    {
        Dog d = new Dog();
        d.eat();
    }
}
```

OUTPUT:

```
Eating bread...
```

Boxing and Unboxing :

Boxing and unboxing are important concepts in C#. The C# Type System contains three data types: Value Types (int, char, etc), Reference Types (object) and Pointer Types.

Boxing: The process of converting a Value Type variable (char, int etc.) to a Reference Type variable (object) is called Boxing.

Value Type variables are always stored in Stack memory, while Reference Type variables are stored in Heap memory.

Example:
```
int i = 23;          // 23 will assigned to i
Object Obj = i; // Boxing
```

Unboxing: The process of converting the Reference type variable into a value type variable is known as unboxing.

Example:
```
int i = 23;
Object obj = i;     // Boxing
int j = (int) obj;  // Unboxing
```

C# implementation to demonstrate the Boxing and Unboxing

```
using System;
class Program
{
    // Main Method
    static public void Main()
    {
            // assigned int value 23 to i
            int i = 23;

            // boxing
            object obj = i;

            // unboxing
            int j = (int)obj;

            // Display result
            Console.WriteLine("Value of ob object is : " + obj);
            Console.WriteLine("Value of i is : " + j);
    }
}
```

Output:
```
Value of ob object is : 23
Value of i is : 23
```

DELEGATES

Definition: It's a type-safe function pointer.

A delegate holds the reference of a method and then calls the method for execution.

In c# we call methods in two ways,

One is by using an **instance of a class** and the second way is by **using the delegates.**

To call a method using delegate we have 3 steps.

1. Define a delegate (declare)

   In C#, the declaration of delegate will be same as method signature

   but only difference is we will use delegate keyword to define it.

   SYNTAX:

   [<Modifiers>] delegate <return type> <Name of delegate> ([<parameter list>])

2. Instantiating the delegate.

3. Now call the delegate by passing the required parameter values, so that internally the method which is bound with the delegate gets executed.

   In c# the delegates must be instantiated with a method or an expression that has a same return type and parameters and we can invoke method through the delegate instance.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace DelegateDemo
{
   public delegate void SampleDelegate(int a, int b);
   class MathOperations
   {
      public void Add(int a, int b)
      {
         Console.WriteLine("Add result is:\n" + (a + b));
      }
      public void Subtract(int x, int y)
      {
```

```
                Console.WriteLine("Subtraction result is:" + (x - y));
            }
        }

        class program
        {
            static void Main(string[] args)
            {
                Console.WriteLine("*****DELEGATE EXAMPLE*******");
                MathOperations m = new MathOperations();
                SampleDelegate delgt = m.Add;
                delgt(100, 200);
                delgt = m.Subtract;
                delgt(200, 10);
                Console.ReadLine();
            }
        }
    }
```

C# Events :

Events in C# are actions that allow classes or objects to inform other classes or objects when an interesting phenomenon occurs.

Some of the properties of events are given as follows:

1. Events have publishers and subscribers. The publishers determine when an event is raised while the subscribers determine the actions that are performed in response to the event.
2. If there are no subscribers for an event, then event is never raised. However, for an event there can be many subscribers.
3. If an event has many subscribers, when the event is raised the event handlers are invoked synchronously.
4. User actions such as menu selections, button clicks etc. are signaled by using events.
5. The events are based on the EventHandler delegate and the EventArgs base class.

Event Syntax

The events are declared using the keyword **event.** The syntax for this is given as followed:

event delegate_name event_name;

In the above syntax, the keyword event is followed by the delegate name and then the event name.

```
using System;
namespace Example
{
 public delegate string demoDelegate(string str1, string str2);
 class MyEvents
 {
    event demoDelegate myEvent;
    public MyEvents()
    {
      this.myEvent += new demoDelegate(this.Display);
    }
    public string Display(string studentname, string subject)
    {
      return "Student: " + studentname + "\nSubject: " +subject;
    }
    static void Main(string[] args)
    {
      MyEvents e = new MyEvents();
      string res = e.myEvent("Jack", "Science");
      Console.WriteLine("RESULT...\n"+res);
    }
  }
}
```

Output:

```
RESULT...

Student: Jack

Subject: Science
```

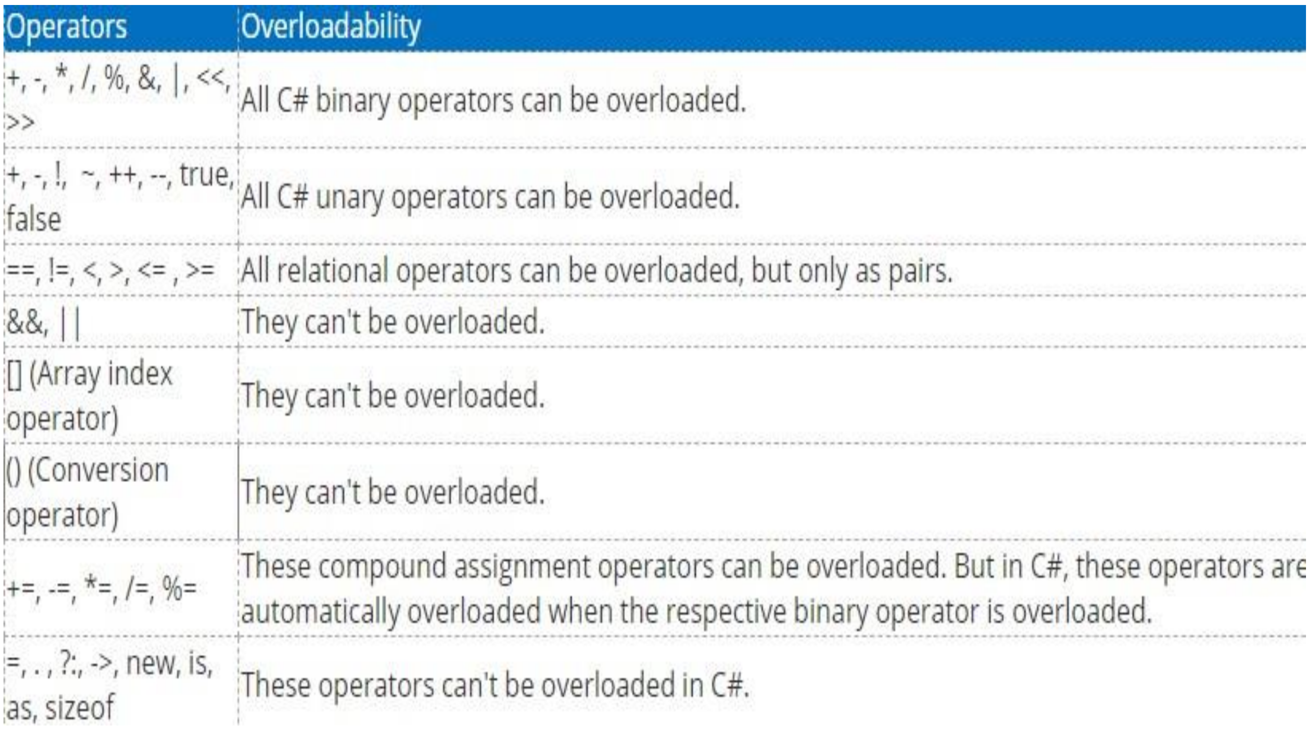
Operator overloading is a technique to redefine a built-in operator. C#, with the help of operator overloading, allows us to use the same built-in operators in different ways. We can build user-defined implementations of various operations where one or both of the operands are of the user-defined type.

Rules for Operator Overloading

To overload an operator, we must use the operator keyword. We have to use both public and static modifiers in the declaration. The unary operator will have one and the binary operator will have two input parameters.

Can We Overload All Operators?

We can overload most of the operators in C#. But there are some operators that we can't overload and some that we can with certain conditions:



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.
[] (Array index operator)	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	These operators can't be overloaded in C#.

Syntax:

```
public static classname  operator op (parameters)
{
// Code
}
```

For Unary Operator

```
    public static classname operator op (t)
    {  // Code
    }
```

For Binary Operator

```
    public static classname operator op (t1, t2)
    {
    // Code
    }
```

The operator is usually is a **keyword** that is used while implementing operator overloading.

Example:

```
using System;
namespace OperatorOverloading
{
   public class Complex
   {
      public int real;
      public int imaginary;
      public Complex(int real, int imaginary)
      {
         this.real = real;
         this.imaginary = imaginary;
      }
      public static Complex operator +(Complex c1, Complex c2)
      {
         return new Complex(c1.real + c2.real, c1.imaginary + c2.imaginary);
      }
      public override string ToString()
      {
         return (String.Format("{0} + {1}i", real, imaginary));
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Complex num1 = new Complex(5, 3);
         Complex num2 = new Complex(3, 4);
         Complex sum = num1 + num2;
         Console.WriteLine("First Complex Number :" + num1);
         Console.WriteLine("Second Complex Number :" + num2);
         Console.WriteLine("The Sum of the Two Numbers :" + sum);
         Console.ReadLine();
      }
```

```
        }
    }
```

Thank You

(Our Creative Info)