Chapter-Three
Creating Types in C#

Object-Oriented Programming (OOPs) in C# / OOPs Concept in C#:

- ☞ Object-Oriented Programming, commonly known as OOPs, is a technique, not a technology. It means it doesn't provide any syntaxes or APIs; instead, it provides suggestions to design and develop objects in programming languages.

How do we Develop Applications?

- ☞ Object-Oriented Programming is a strategy that provides some principles for developing applications or software. It is a methodology. Like OOPs, other methodologies exist, such as Structured Programming, Procedural Programming, or Modular Programming. But nowadays, one of the well-known and famous styles is Object Orientation, i.e., Object-Oriented Programming.

- ☞ Nowadays, almost all the latest programming languages support object orientation. This object orientation is more related to the designing of software, and this deals with the internal design of the software, not the external design. So, it is nowhere related to the users of the software. It is related to the programmers who are working on developing software.

- ☞ With the help of Object Orientation, application development or programming becomes more and more systematic, and we can follow engineering procedures to develop software. Like in other engineering, how a product is developed, in the same way, a software product is developed by adopting object orientation.

- ☞ If we talk a little bit about other engineering, like a civil engineer constructing a building, then first of all, he/she will make a plan or design. While making a design or plan, they may have many options, but they will select and finalize one of the designs. Then, they will start constructing once it is finalized as a blueprint on paper. In the same way, an electronic engineer, when manufacturing any device, will come up with some design that is the circuit design of that device on paper. And once that design or blueprint is finalized, he will start manufacturing the device.

- So, the Object Orientation all depends on how we see the internal system or understand the internal system. So, if you understand the system ideally and if your perspective is very clear, you can develop a better system.

Object-Oriented vs Modular Programming

- Now, I will explain to you Object Orientation by comparing it with Modular Programming. The reason is that people who came to learn C# already know the C language. The C programming language supports Modular or Procedural Programming. Based on that, I can give you an idea of how object orientation differs from modular programming. Let us compare Object-Oriented vs Modular Programming through some examples.
- So first, we are taking an example of a bank. If you're developing an application for a bank using modular programming, how do you see the system, how do you see the workings of a bank, and what will be your design? That depends on how you understand it and how you see the system. So, let us see how we look at the bank system using modular programming.
- In a bank, you can open an account, you can deposit an amount, you can withdraw an amount, you can check your account balance, or you can also apply for a loan, and so on. So, these are the things that you can do at the bank.
- So, Opening an Account, Depositing Money, Withdrawing Money, Checking Your Balance, and Applying For a Loan are functions. All these are nothing but functions. And you can do the specific operations by calling that specific function. So, if you're developing software for a bank, it is nothing but a collection of functions. So, the bank application will be based on these functions, and the user of the bank application will be utilizing these functions to perform his required task. So, you will develop software as a set of functions in Modular Programming.
- Now, for Object Orientation, we would take some different examples. The government provides many services like electric, water supply, education, and transport, and even the government can have banks. So, these are the different departments of a government. Now, what can you do in the electric department as a user? You can apply for a new connection, you can close your connection if you have extra connections, or you can make a bill payment. What are these? These are functions belonging to the Electric Department.
- Now, in the same way, the bank is also there. The same functions like account opening, deposit, withdraw, check balance, apply for a loan, etc., are also there. These are functions belonging to the Bank.
- What do we call these? We call them objects. So, the complete system for the government or a complete software for a government is a collection of objects. Now, each object has its relevant functions. So, complete software is a collection of objects containing functions and data related to those functions.

- ☞ In Modular Programming, the system was a collection of functions. So, if you compare them now, in modular programming, we are looking at the very close level, and in object-oriented programming, we are looking at a little far away level.

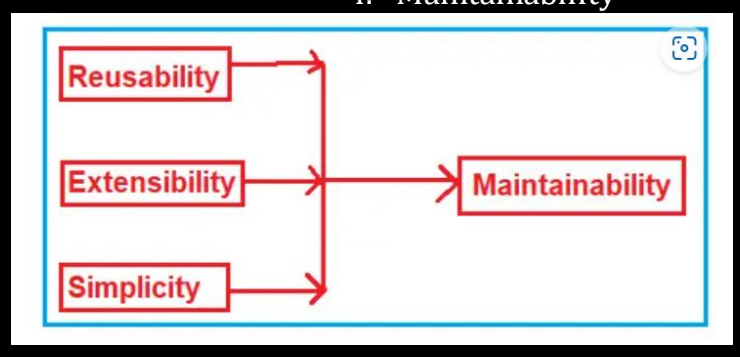
Why Object Orientation?

- ☞ Let us talk about a manufacturing firm which manufactures cars or vehicles. If you look at that manufacturing farm, then it may be working in the form of departments like one is an inventory department that maintains the stock of raw materials and one is manufacturing, which is the production work that they do, and one department will be looking at sales and one department is looking at marketing. One is about payroll, and one is for accounts, and so on. So, there may be many departments.
- ☞ Suppose you are developing software only for payroll or inventory purposes. In that case, you may look at the system just like a modular approach, and in that, you can find functions like placing an order and checking the item in stock. These types of things can have a set of functions so that you can develop the software only for the inventory system as a collection of functions. Still, when developing software for the entire organization, you must see things in objects.
- ☞ So, the inventory item is an object, an employee is an object, an account is an object, and a product manufacturer is an object. The machines used for production are an object. So, all these things are objects. Here, you need to see things in the form of objects and define their data and the functions that they're performing. We are looking at the system at a higher level. So, we can adopt object orientation.

What are the problems of Modular Programming?

- ☞ Modular programming has the following problems.

1. Reusability
2. Extensibility
3. Simplicity
4. Maintainability



Reusability: In Modular Programming, we must write the same code or logic at multiple places, increasing code duplication. Later, if we want to change the logic, we must change it everywhere.

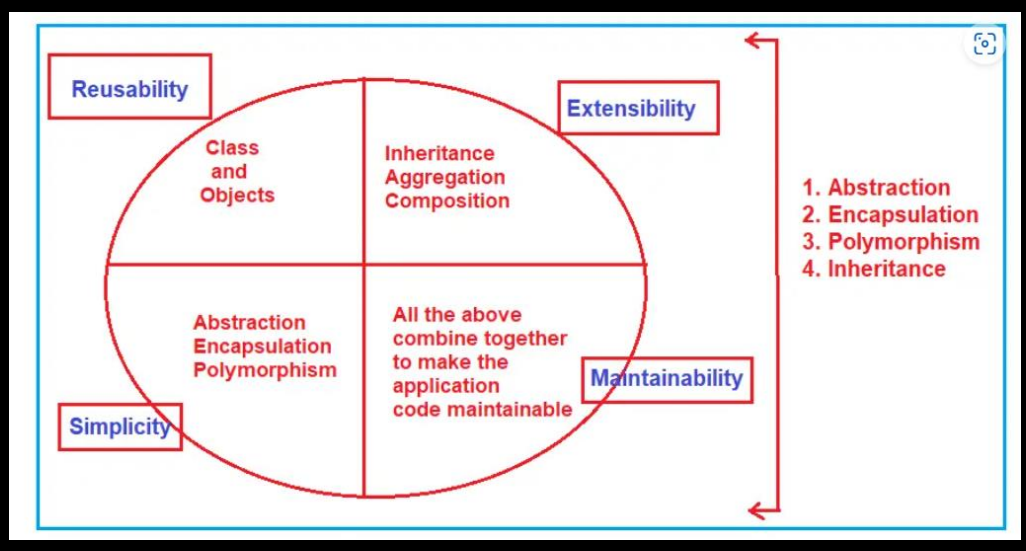
Extensibility: It is not possible in modular programming to extend the features of a function. Suppose you have a function and you want to extend it with some additional features; then it is not possible. You have to create an entirely new function and then change the function as per your requirement.

Simplicity: As extensibility and reusability are impossible in Modular Programming, we usually end up with many functions and scattered code.

Maintainability: As we don't have Reusability, Extensibility, and Simplicity in modular Programming, it is very difficult to manage and maintain the application code.

How Can We Overcome Modular Programming Problems?
We can overcome the modular programming problems (Reusability, Extensibility, Simplicity, and Maintainability) using Object-Oriented Programming. OOPs provide some principles, and using those principles, we can overcome Modular Programming Problems.

What Is Object-Oriented Programming?
Let us understand Object-Oriented Programming, i.e., OOP concepts using C#. Object-oriented programming (OOPs) in C# is a design approach where we think in terms of real-world objects rather than functions or methods. Unlike procedural programming language, in OOPs, programs are organized around objects and data rather than action and logic. Please have a look at the following diagram to understand this better.



Reusability:
To address reusability, object-oriented programming provides something called Classes and Objects. So, rather than copy-pasting the same code repeatedly in different places, you can create a class and make an instance of the class, which is called an object, and reuse it whenever you want.

Extensibility:
Suppose you have a function and want to extend it with some new features that were impossible with functional programming. You have to create an entirely new function and then change the whole function to whatever you want. OOPs, this problem is addressed using concepts called Inheritance, Aggregation, and Composition. In our upcoming article, we will discuss all these concepts in detail.

Simplicity:
Because we don't have extensibility and reusability in modular programming, we end up with lots of functions and scattered code, and from anywhere we can access the functions, security is less. In OOPs, this problem is addressed using Abstraction, Encapsulation, and Polymorphism concepts.

Maintainability:
As OOPs address Reusability, Extensibility, and Simplicity, we have good, maintainable, and clean code, increasing the application's maintainability.

What are the OOPs Principles or OOPs Concepts in C#?

- ☞ OOPs provide 4 principles. They are

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

Note: Don't consider Class and Objects as OOPs principle. We use classes and objects to implement OOP Principles.

Let's understand the definitions of the OOPs Principle in this session. From the next article onwards, we will discuss all these principles in detail using some real-time examples.

CLASSES

- ☞ A classes in C# are templates that are used to create object and to define object data types and methods.

- A core properties include the data types and methods that may be used by the object.
- All class objects should have the basic class properties.
- A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type.
- In general, class declarations can includes the following components in order as:

1. **Modifiers:** A class can be public or has default access.
2. **Class name:** The name should begin with a initial letter (capitalized by convention)
3. **Superclass (if any):** The name of the class's parent (superclass), if any preceded by the (:). A class can only extend (subclass) one parent.
4. **Interfaces (if any):** A common-separated list of interfaces implemented by the class, if any preceded by the (:). A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }.

- General form of a class is shown below:

Access Modifier

Class Name

```
public class Dog
{
        String  breed;
        int age;
        String color;

        void barking()
        {

        }
        void hungry()
        {
        }
        void sleeping()
        {
        }
```

Data Member

Member Function or Methods

}

OBJECT

- A combination of data and function is known as object. An object has state and behavior.
- The state of an object is stored in fields (variables), while methods (functions) display the object's behavior.
- In C#, an object is created using the keyword **"new"**. Object is an instance of a class.
- There are three steps to creating an object in C#
  - Declaration of the object
  - Instantiation of the object
  - Initialization of the object

When an object is declared, a name is associated with that object. The object is instantiated so that memory space can be allocated. Initialization is the process of assigning a proper initial value to this allocated space.

The properties of an object includes:

I. One can only interact with the object through its methods. Hence, internal details are hidden.
II. When coding an existing object may be reused.
III. When a program's operation is hindered by a particular object, that object can be easily removed and replaced.

A new object t from the class "tree" is created using the following syntax:

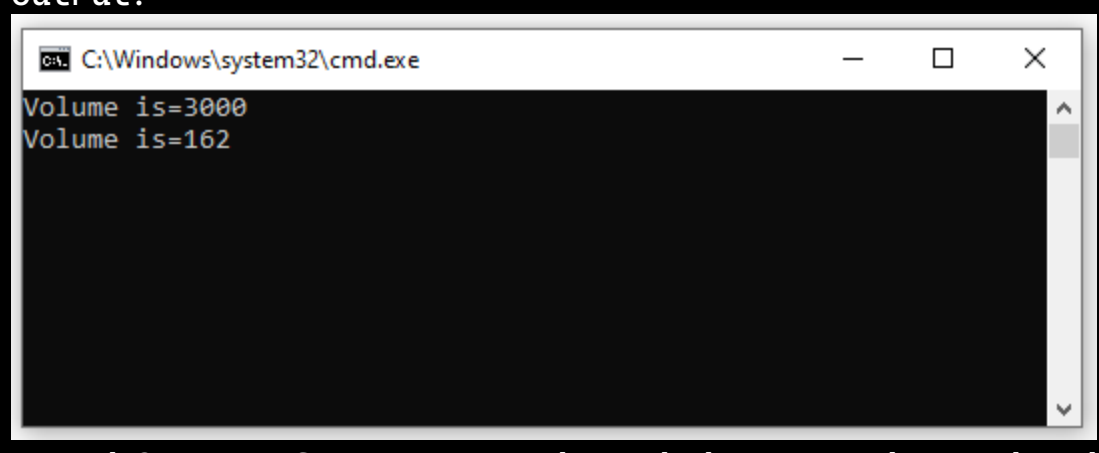
```
Tree t = new Tree( );
```

Example program to demonstrate class and object

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Project1
{
    internal class Box
    {
        double weidth;
        double height;
        double depth;

        double volume()
        {
```

```
    return weidth * height * depth;
}
void setDim(double w, double h, double d)
{
    weidth = w;
    height = h;
    depth = d;

}
class BoxDemo
{


static void Main(string[] args)
{
        Box mybox1 = new Box();
        Box mybox2 = new Box();

        double vol;

        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);
        vol = mybox1.volume();
        Console.WriteLine("Volume is=" + vol);
        vol = mybox2.volume();
        Console.WriteLine("Volume is=" + vol);


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

OutPut:

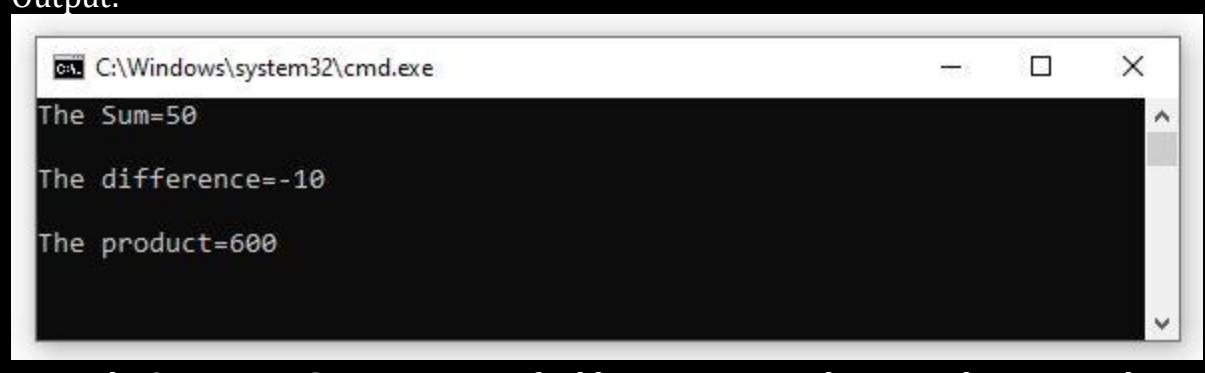


Example2: Write a C# program to make a calculator using object and method?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Math
```

```
{
    internal class Calculator
    {
        int n1;
        int n2;
        int result;
        void add()
        {
            result = n1 + n2;
            Console.WriteLine("The Sum="+result);
            Console.ReadLine();
        }
        void subtract()
        {
            result = n1 - n2;
            Console.WriteLine("The difference="+result);
            Console.ReadLine();
        }
        void mul()
        {
            result = n1 * n2;
            Console.WriteLine("The product="+result);
            Console.ReadLine();
        }
        static void Main(string[] args)
        {
            Calculator obj = new Calculator();
            obj.n1 = 20;
            obj.n2 = 30;
            obj.add();
            obj.subtract();
            obj.mul();

        }
    }
}
```

Output:

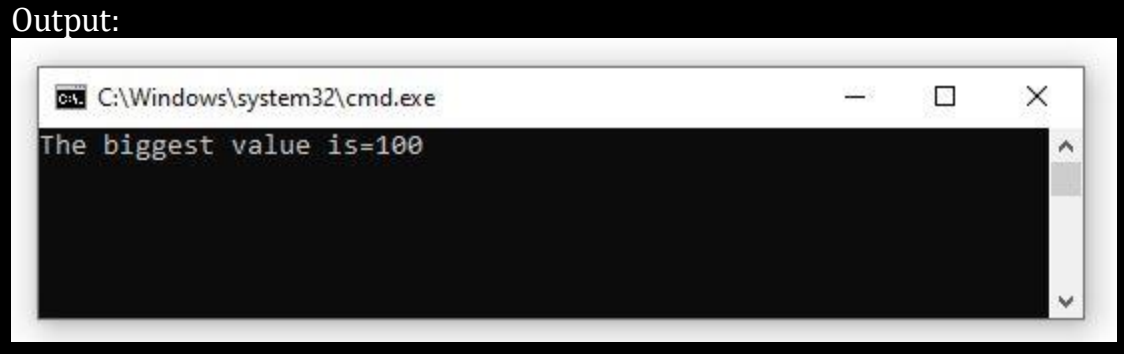


Example-3: Write a C# program to find biggest among three number using object and method?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace Objectoriented
{
    internal class Program
    {
        public int Findbiggest(int n1, int n2)
        {
            int result;
            if(n1>n2)
            {
                result = n1;
            }
            else
            {
                result = n2;

            }
            return result;

        }
        static void Main(string[] args)
        {
            Program fb = new Program();
            int x = 100;
            int y = 50;
            int z = fb.Findbiggest(x, y);
            Console.WriteLine("The biggest value is=" + z);
            Console.ReadKey();
        }
    }
}
```

Output:

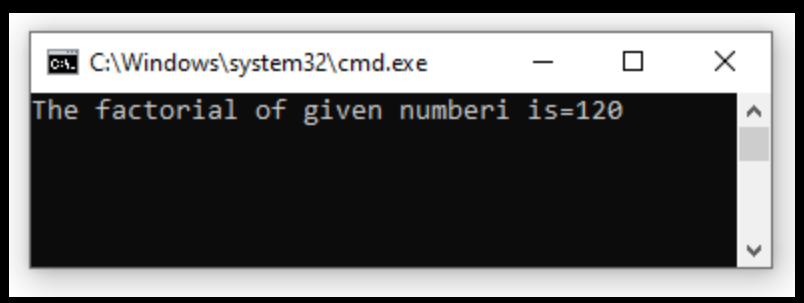


Example-4:

Write a C# program to find the factorial of a number?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Factorial
{
    internal class Factor
    {
        int fact()
```

```
        {
            int num = 5;
            int f = 1;
            for(int i=1;i<=num;i++)
            {
                f = f * i;
            }
            return f;
        }

        static void Main(string[] args)
        {
            Factor fa = new Factor();
            int result=fa.fact();
            Console.WriteLine("The factorial of given numberi is=" + result);
            Console.ReadKey();

        }
    }
}
```

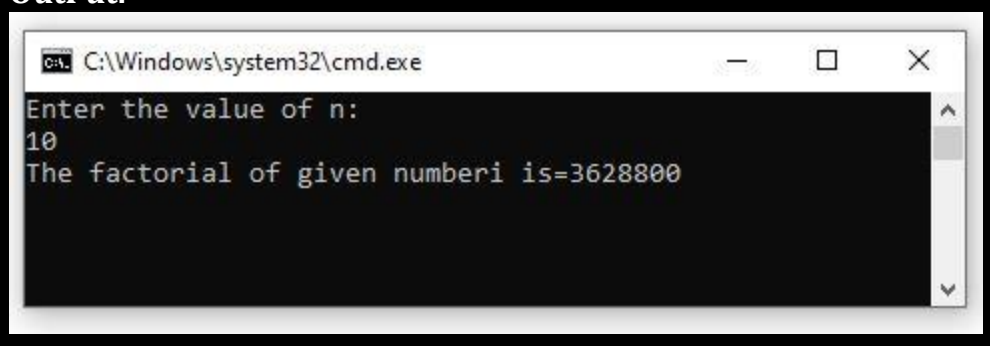


Write a C# program to find the factorial of given number **n** with argument with return value?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Factorial
{
    internal class Factor
    {
        int fact(int num)
        {

            int f = 1;
            for(int i=1;i<=num;i++)
            {
                f = f * i;
            }
            return f;
        }

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

```
            int n;
            Console.WriteLine("Enter the value of n:");
            n = Convert.ToInt32(Console.ReadLine());
            Factor fa = new Factor();
            int result=fa.fact(n);
            Console.WriteLine("The factorial of given numberi is=" + result);
            Console.ReadKey();

        }
    }
}
```

OutPut:



FIELDS

- A field is a variable that is a member of a class.
  For example:
  class Person
  {
  public string name;
  public int age = 30;
  }

Fields allow the following modifiers:

Static modifier	static
Access modifier	public, internal, private, protected
Inheritance modifier	new
Unsafe code modifier	unsafe
Read-only modifier	read-only
Threading modifier	volatile

The readonly modifier

- The readonly modifier prevents a field from being modified after construction. A read-only field can be assigned only in its declaration or enclosing type's constructor.

static readonly int pi = 3.14;

Access modifier

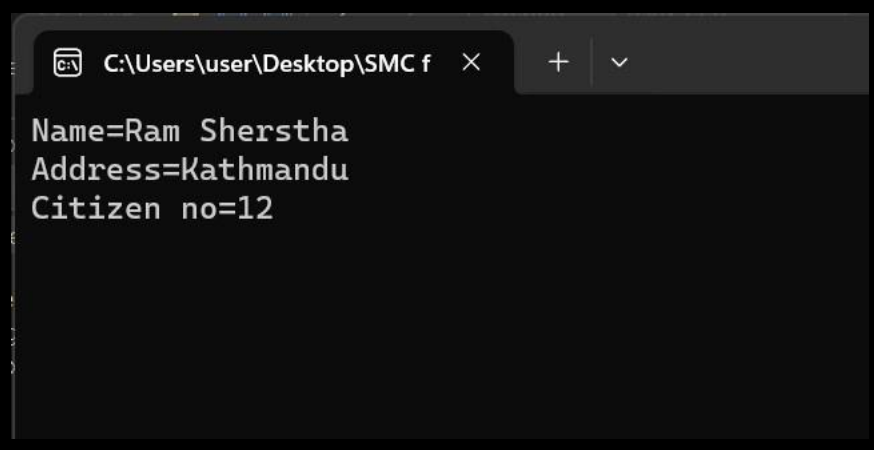
- ☞ Access modifier specify the visibility or accessibility of class and member.
- ☞ Access modifier provide restriction in class and its members.

Modifier	**Description**
private	The code is only accessible within the same class
public	The code is accessible for all classes
internal	The code is only accessible within its own assembly (project) but not from another assembly.
protected	The code is available within the same class or subclass of that class.

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AccessModifier
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Person pe = new Person();
            pe.name = "Ram Sherstha";
            pe.address = "Kathmandu";
            pe.citizen = 0012;
            pe.Displayinfo();
        }
    }
    class Person
    {
      public  string name;
       public string address;
       public double citizen;
        public void Displayinfo()
        {
            Console.WriteLine("Name=" + name);
            Console.WriteLine("Address=" + address);
            Console.WriteLine("Citizen no=" + citizen);
            Console.ReadKey();
        }

    }
}
```

Output:



METHODS/FUNCTIONS

- A method performs an action in a series of statements.
- A method can receive input data from the caller by specifying parameters and output data back to the caller by specifying a return type.
- A method can specify a void return type, indicating that it doesn't return any value to its caller.
- A method can also output data back to the caller via reference parameters.
- A method's signature must be unique within the type.
- A method's signature comprises its name and parameter types in order (but not the parameters name nor the return types).

Method allow the following modifiers:

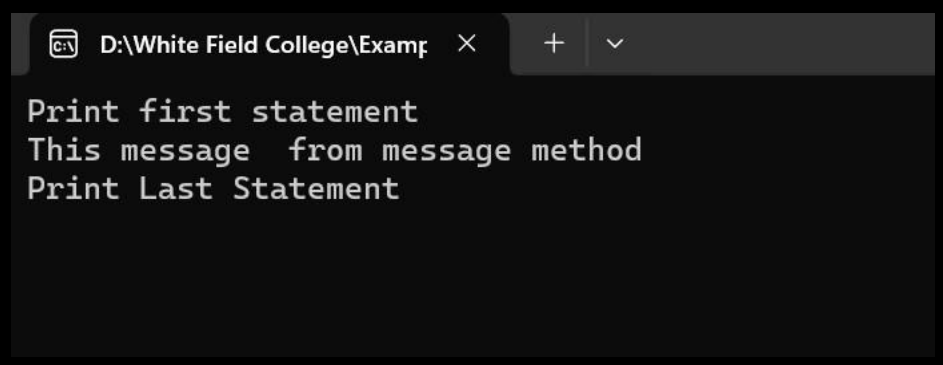
Static modifier	static
Access modifier	public, internal, private, protected
Inheritance modifier	new virtual abstract overfed sealed
Partial method modifier	partial
Unmanaged code modifier	unsafe extern
Asynchronous code modifier	async

Example-1: With no arguments with no return value

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace MethodImplementation
{
    internal class Program
    {
```

```
        static void message()
        {
            Console.WriteLine("This message  from message method");
        }
        static void Main(string[] args)
        {
            Console.WriteLine("Print first statement");
            message();
            Console.WriteLine("Print Last Statement");
            Console.ReadKey();
        }
    }
}
```

Output:

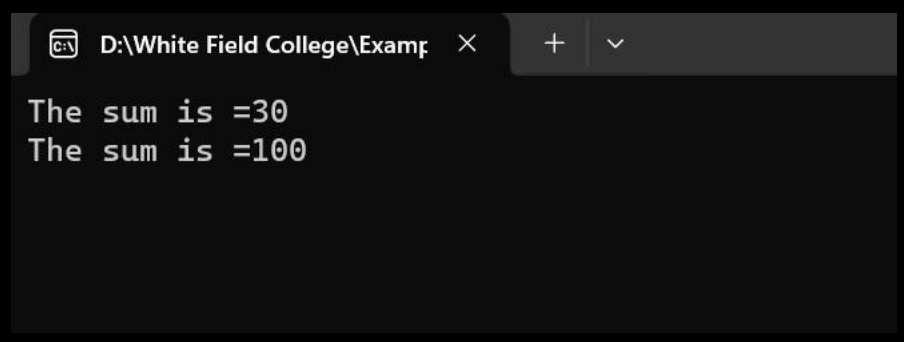


Example-2: With arguments with no return value

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace MethodImplementation
{
    internal class Program
    {
        static void Add(int x,int y)//Formal parameters
        {
            int sum = x + y;
            Console.WriteLine("The sum is =" + sum);

        }
        static void Main(string[] args)
        {
            int num1 = 10, num2 = 20;
            Add(num1, num2);//Actual parameters
            Add(50, 50);//Actual parameters
            Console.ReadKey();
        }
```

```
    }
}
```

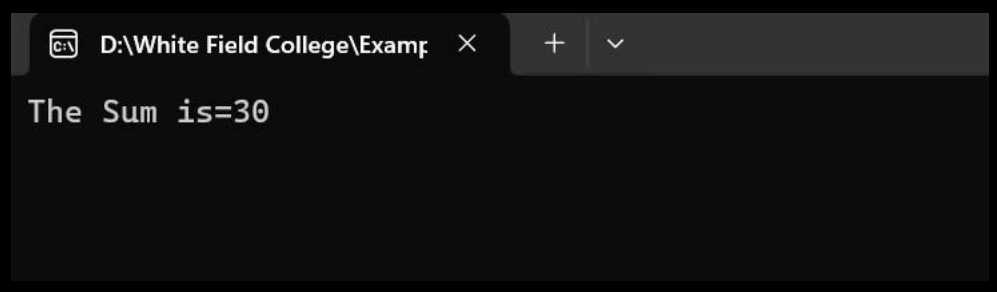
Output:



Example-3: With arguments with return value

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace MethodImplementation
{
    internal class Program
    {
        static int Add(int x,int y)//Formal parameters
        {
            int sum = x + y;
            return sum;

        }
        static void Main(string[] args)
        {
            int num1 = 10, num2 = 20;
            int result=Add(num1, num2);//Actual parameters
            Console.WriteLine("The Sum is=" + result);
            Console.ReadKey();
        }
    }
}
```

Output:

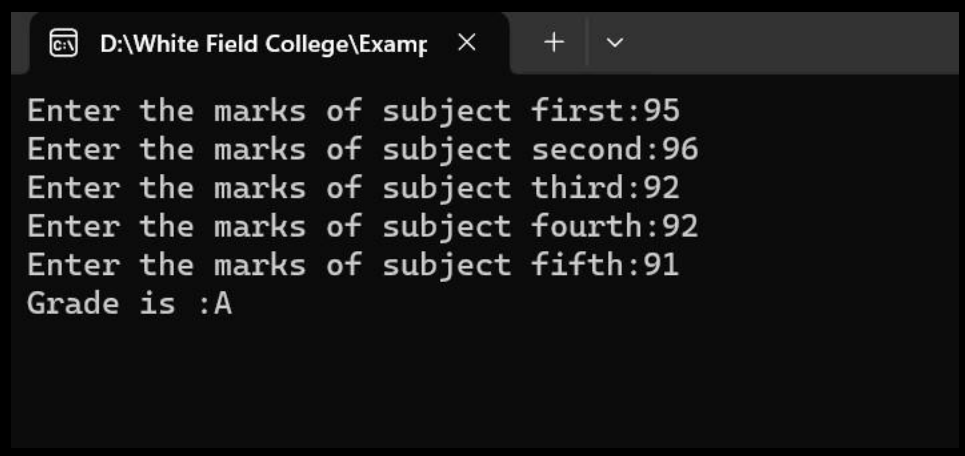


Example-3

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace MethodImplementation
{
    internal class Program
    {
        static char GetGrade()
        {
            float percent = GetPercentage();
            if (percent >= 90)
                return 'A';
            else if (percent >= 60)
                return 'B';
            else if (percent >= 40)
                return 'C';
            else
                return 'D';

        }
        static float GetPercentage()
        {
            float total_mark = GetTotal();
            float per = total_mark / 5;
            return per;

        }
        static float GetTotal()
        {
            Console.Write("Enter the marks of subject first:");
            float mark1 = float.Parse(Console.ReadLine());
            Console.Write("Enter the marks of subject second:");
            float mark2 = float.Parse(Console.ReadLine());
            Console.Write("Enter the marks of subject third:");
            float mark3 = float.Parse(Console.ReadLine());
            Console.Write("Enter the marks of subject fourth:");
            float mark4 = float.Parse(Console.ReadLine());
            Console.Write("Enter the marks of subject fifth:");
            float mark5 = float.Parse(Console.ReadLine());
            float total = mark1 + mark2 + mark3 + mark4 + mark5;
```

```
            return total;
        }
        static void Main(string[] args)
        {
            Console.WriteLine("Grade is :"+GetGrade());
            Console.ReadKey();
        }
    }
}
```

Output:



Method Calling

- ☞ Method can be called in following ways:
    1. **Call by Value**
    2. **Call by Reference**
    3. Call by Output
    4. Call by Params

Note:

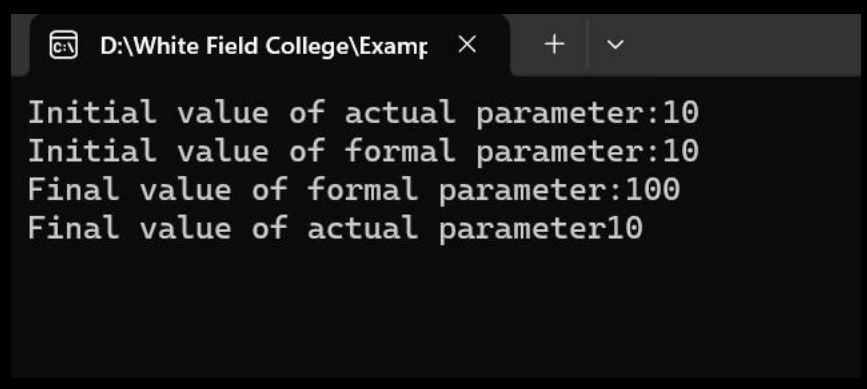
The parameters passed to the function are called **actual parameters** whereas the parameters received by the function are called **formal parameters**.

1. **Call By Value**

- ☞ In call by value method of parameter passing, the values of actual parameters are copied to the function's formal parameters.
    - ❖ There are two copies of parameters stored in different memory locations.
    - ❖ One is the original copy and the other is the function copy.
    - ❖ Any changes made inside functions are not reflected in the actual parameters of the caller.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace MethodImplementation
{
    internal class Program
    {
        static void SimpleMethod(int x)
        {
            Console.WriteLine("Initial value of formal parameter:" + x);
            x = 100;
            Console.WriteLine("Final value of formal parameter:" + x);
        }


        static void Main(string[] args)
        {
            int a= 10;
            Console.WriteLine("Initial value of actual parameter:" + a);
            SimpleMethod(a);
            Console.WriteLine("Final value of actual parameter" + a);

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

Output:



2. **Call By Reference**

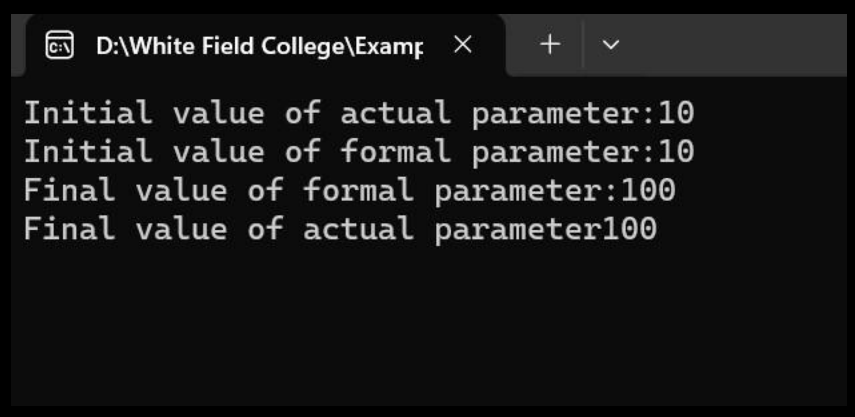
☞ In call by reference method of parameter passing, the address of the actual parameters is passed to the function as the formal parameters.

- ❖ Both the actual and formal parameters refer to the same locations.

- Any changes made inside the function are actually reflected in the actual parameters of the caller.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace MethodImplementation
{
    internal class Program
    {
        static void SimpleMethod(ref int x)
        {
            Console.WriteLine("Initial value of formal parameter:" + x);
            x = 100;
            Console.WriteLine("Final value of formal parameter:" + x);
        }


        static void Main(string[] args)
        {
            int a= 10;
            Console.WriteLine("Initial value of actual parameter:" + a);
            SimpleMethod(ref a);
            Console.WriteLine("Final value of actual parameter" + a);

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

Output:

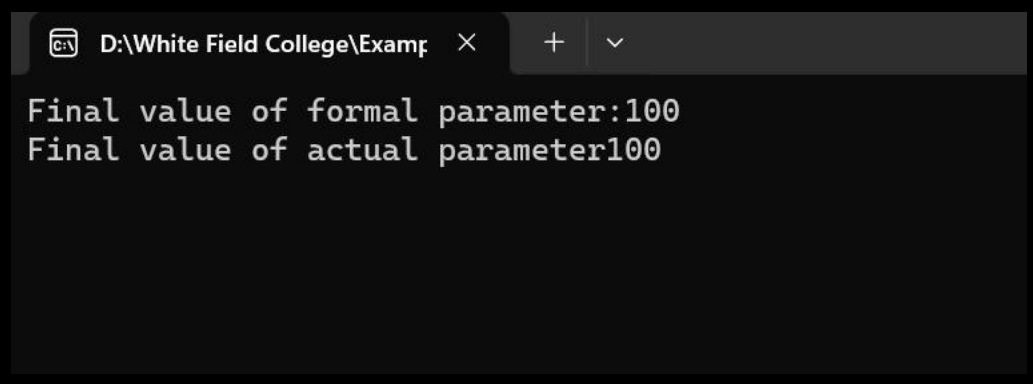


3. Call By Output

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;

namespace MethodImplementation
{
    internal class Program
    {
        static void SimpleMethod(out int x)
        {
            //Console.WriteLine("Initial value of formal parameter:" + x);
            x = 100;
            Console.WriteLine("Final value of formal parameter:" + x);
        }


        static void Main(string[] args)
        {
            int a;//= 10;
            //Console.WriteLine("Initial value of actual parameter:" + a);
            SimpleMethod(out a);
            Console.WriteLine("Final value of actual parameter" + a);

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

Output:

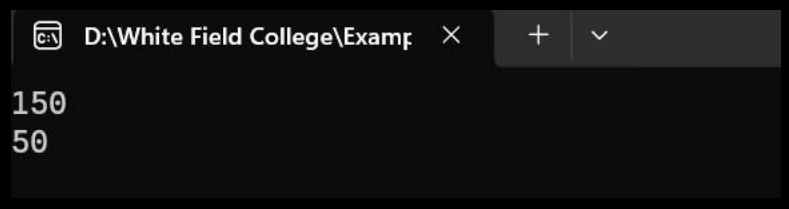


4. Call By params

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
```

```
namespace MethodImplementation
{
    internal class Program
    {
        static int Add( int num1, int num2, params int[] list)
        {
            int sum = num1 + num2;
            foreach(int item in list)
            {
                sum += item;
            }
            return sum;
        }


        static void Main(string[] args)
        {
            Console.WriteLine(Add(10,20,30,40,50));
            Console.WriteLine(Add(30, 20));

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

Output:



Expression-bodied methods

A method that comprises a single expression, such as the following:

```
int     Test    (int x )
        {
                return x *2;
        }
```

This can be written more briefly as an expression-bodied method. A fat arrow replaces the braces and return keyword:

```
int Test (int x ) => x *2;
```

Expression-bodied functions can also have a void return type:
void Test (int x) => Console.WriteLine (x);

Overloading Methods

- A type may overload methods (have multiple methods with the same name), as long as the signatures (i.e. the number of the parameters, order of the parameters, and data types of the parameters) are different. For example, the following methods can all coexist in the same type:

```
void Add (int x) {……..}
void Add (double x ) {……..}
void Add (int x, float y) {…….}
void Add (float x, float y) {…….}
```

Method overloading can be achieved by the following ways:

- By changing number of parameters in a method.
- By changing order of parameters in a method.
- By using different data types for parameters.

By changing numbers of parameters in a method

Syntax:

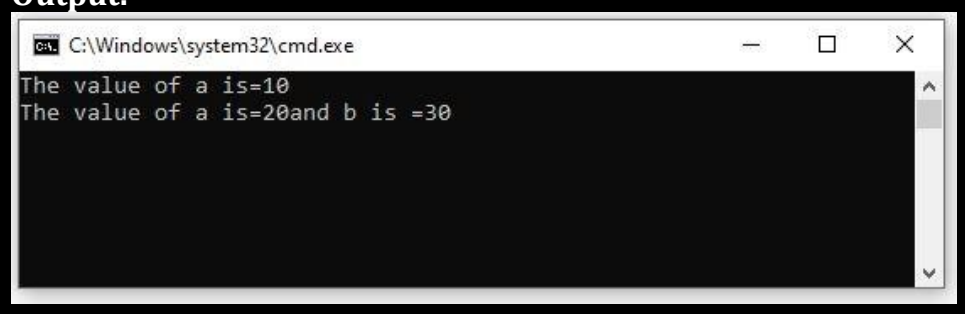
```
void display (int a )
        {
                ……..
        }
        ……..


void display (int a,    int b )
        {
                ……..
        }
```

For example:

```
using System;

namespace Project1
{
    internal class Program
    {
```

```
        void Display(int a) //single parameter
        {
            Console.WriteLine("The value of a is="+ a);
        }
        void Display(int a, int b)
        {
            Console.WriteLine("The value of a is=" + a + "and b is =" + b);
        }
        static void Main(string[] args)
        {
            Program p = new Program();
            p.Display(10);
            p.Display(20, 30);
            Console.ReadKey();
        }

    }
}
```

Output:



By using different data types for parameters.

Syntax:

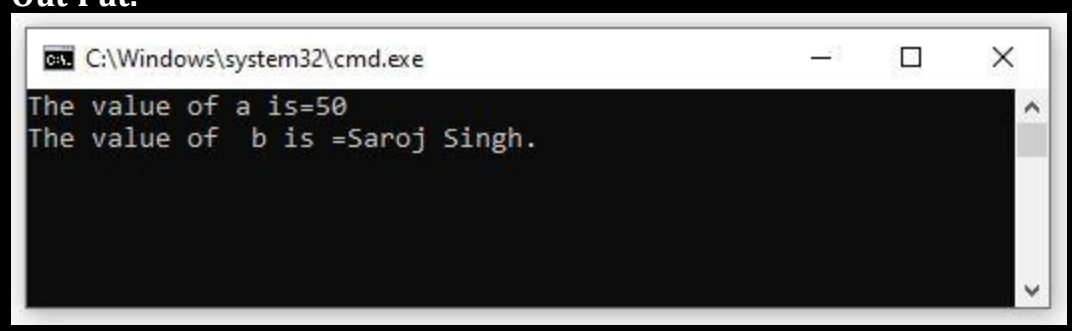
```
void display(int a)
{
      ……..
}
void display(string  b)
{
      ……..
}
```

Example:

```
using System;

namespace Project1
{
    internal class Program
    {
        void Display(int a) //single parameter
        {
            Console.WriteLine("The value of a is="+ a);
```

```
        }
        void Display(string b)
        {
            Console.WriteLine("The value of  b is =" + b);
        }
        static void Main(string[] args)
        {
            Program p = new Program();
            p.Display(50);
            p.Display("Saroj Singh.");Type equation here.
            Console.ReadKey();
        }

    }
}
```

Out Put:



By changing order of parameters in a method.

Syntax:

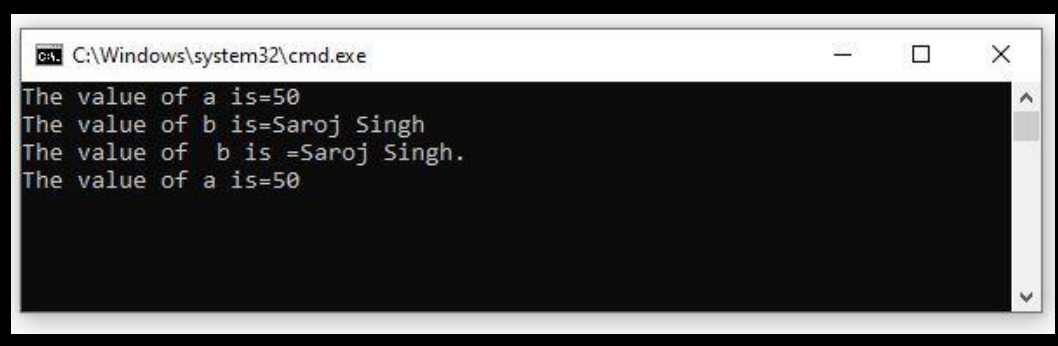
```
void display(int a, string b)
{
      ……..
}
void display(string  b, int a)
{
      ……..
}
```

Example:

```
using System;

namespace Project1
{
    internal class Program
    {
        void Display(int a, string b)
        {
            Console.WriteLine("The value of a is="+ a);
            Console.WriteLine("The value of b is=" + b);
        }
```

```
        void Display(string b, int a)
        {
            Console.WriteLine("The value of  b is =" + b);
            Console.WriteLine("The value of a is=" + a);
        }
        static void Main(string[] args)
        {
            Program p = new Program();
            p.Display(50,"Saroj Singh");
            p.Display("Saroj Singh.",50);
            Console.ReadKey();
        }

    }
}
```

Out Put:

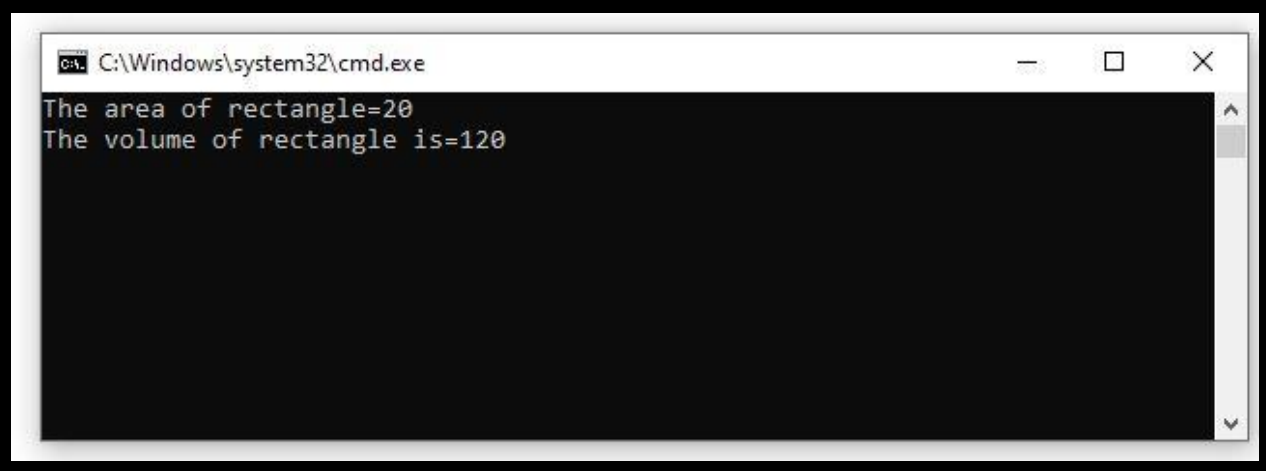


Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace MethodOverloading
{
    internal class Program
    {
        public void Display(int l, int b)
        {
            int area = l * b;
            Console.WriteLine("The area of rectangle=" + area);
        }
        public void Display(int l, int b, int h)
        {
            int volume = l * b * h;
            Console.WriteLine("The volume of rectangle is=" + volume);
        }
        static void Main(string[] args)
        {
            Program obj = new Program();
            obj.Display(4, 5);
            obj.Display(4, 5, 6);
```

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

Output:



CONSTRUCTORS

- ☞ Constructors are special methods in C# that are automatically called when an object of a class is created to initialize all the class data members.
- ☞ If there are no explicitly defined constructors in the class, the compiler creates a default constructor automatically.
- ☞ Some important point remember about the constructors are as follows:
    - ❖ Constructor of a class must have the same name as the class name in which it resides.
    - ❖ A constructor can not be abstract, final, and Synchronized.
    - ❖ Within a class, you can create only one static constructor.
    - ❖ A constructor doesn't have any return type, not even void.
    - ❖ A static constructor cannot be a parameterized constructor.
    - ❖ A class can have any number of constructors.
    - ❖ Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

The key difference between constructor and methods are as follows:

- ❖ A constructor does not have return type.
- ❖ The name of constructor must be the same as the name of the class.
- ❖ Unlike methods, constructors are not considered members of a class.
- ❖ A constructor is called automatically when a new instance of an object is created.

Types of Constructors

I. Default Constructor
II. Instance Constructor
III. Overloaded Constructor
IV. Static Constructor

Default Constructor:

- ☞ A constructor with no parameters is called a default constructor.
- ☞ A default constructor has every instance of the class to be initialized to the same values.
- ☞ The default constructor initializes all numeric fields to zero and all string and object fields to null inside a class.
- ☞ It is also known as ***nullary*** constructor.

Syntax:

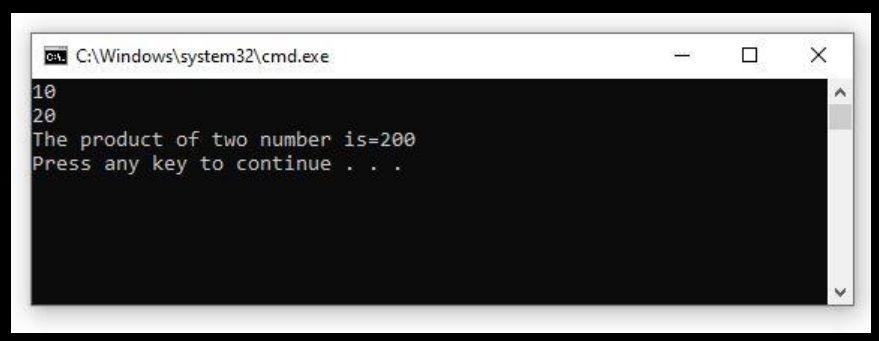
```
class ClassName
{
      ClassName()
      {
      }
}
```

Example: Example program to demonstrate default constructor?

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Dconstructor
{
    internal class multiplication
    {
        int a,b;
        public multiplication()    //Default Constructor
        {
            a = 10;
            b = 20;
        }
        static void Main(string[] args)
```

```
        {
            multiplication mul = new multiplication();
            Console.WriteLine(mul.a);
            Console.WriteLine(mul.b);
            Console.WriteLine("The product of two number is=" + mul.a * mul.b);
        }
    }
}
```

Output:



2. Instance Constructor/parameterized

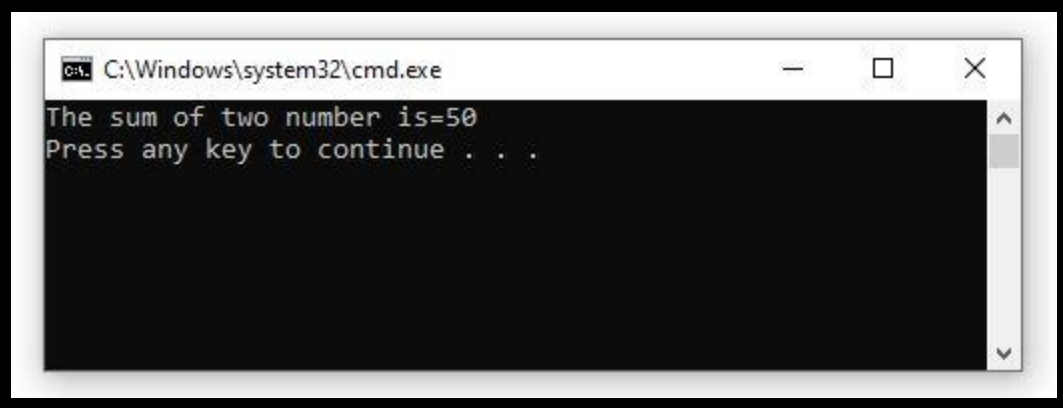
- ☞ We can declare an instance constructor to specify the code that is executed when you create a new instance of a type with the **new** expression.
- ☞ To initialize a static class or static variables in a non-static class, you can define a static constructor.

Example: Example program to demonstrate instance constructor.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Instancecon
{
    internal class Add
    {
        private int sum;
        public Add(int a, int b)//Instance Constructor
        {
            sum = a + b;
            Console.WriteLine("The sum of two number is=" + sum);

        }

            static void Main(string[] args)
            {
                new Add(20, 30);
            }
```

```
    }
}
```

Output:



Note: Instance constructors allow the following modifiers:

public, internal, private, protected

3. Overloaded Constructors /Constructor Overloading

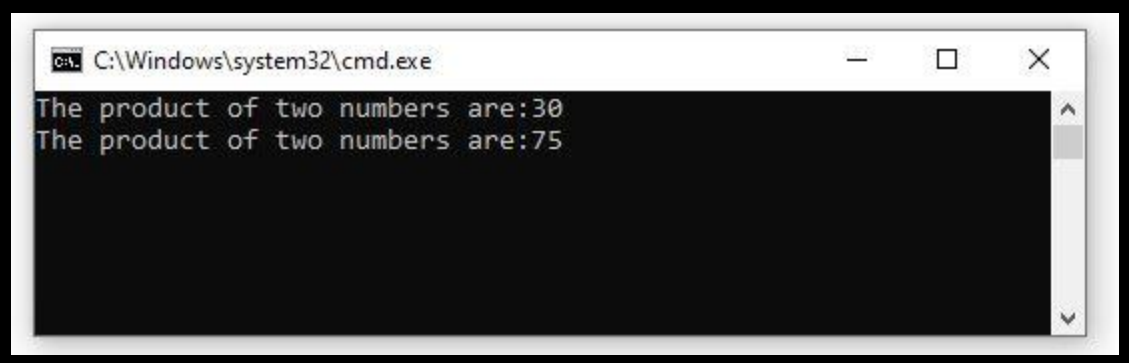
- ☞ A class may overloaded constructor.
- ☞ When more than one constructor with the same name is defined in the same class then they are called constructor overloaded.
- ☞ The parameters are different in each constructor.

Example: Example program to demonstrate constructor overloaded.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Constructor
{
    internal class Coverload
    {
        private int mul;
        public Coverload( int a, int b)
        {
            mul = a * b;
            Console.WriteLine("The product of two numbers are:" + mul);
        }
        public Coverload(int a, int b, int c)//constructor overloaded
        {
            mul = a * b * c;
            Console.WriteLine("The product of two numbers are:" + mul);
        }
        static void Main(string[] args)
```

```
        {
            new Coverload(5, 6);
            new Coverload(5, 5, 3);
            Console.ReadKey();
        }
    }
}
```

Output:



4. Static Constructor

- ☞ A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed only once.
- ☞ It is called automatically before the first instance is created or any static members are referenced.
- ☞ Generally, in C# the static constructor will not accept any access modifiers and parameters. In words we can say it's a parameter less.
- ☞ **Properties of static constructor in C# as:**
  - I. Static constructor in C# won't accept any parameters and access modifiers.
  - II. The static constructor will invoke automatically, whenever we create a first instance of class.
  - III. The static constructor will be invoked by CLR so we don't have a control on static constructor execution order in C#.
  - IV. In C#, only one static constructor is allowed to create.

Syntax:

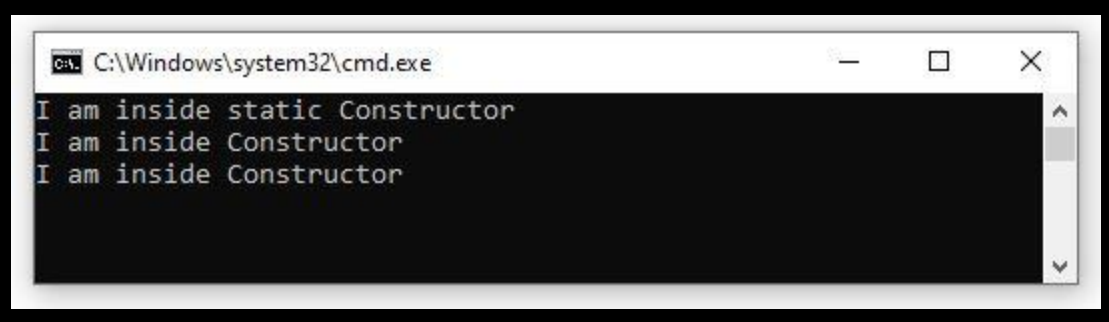
```
class SCons
{
        static SCons()          //static constructor
```

```
        {
                //your custom code here.
        }
}
```

Example: Example program to demonstrate static constructor as

```
using System;

namespace Constructor
{
    internal class Cons
    {
        public Cons()
        {
            Console.WriteLine("I am inside Constructor");
        }
        static Cons()//Will be called only once.
        {
            Console.WriteLine("I am inside static Constructor");
        }

        static void Main(string[] args)
        {
            new Cons();//both constructor called
            new Cons();
             //static constructor will not be called now.

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

Output:



DESTRUCTOR

- ☞ In C#, destructor (finalizer) is used to destroy objects of class when the scope of an object ends. It has the same name as the class and starts with a tilde ~.
- ☞ Characteristics of destructors are as follows:

I. We can only have one destructor in a class.
II. A destructor cannot have access modifiers, parameters, or return types.
III. A destructor is called implicitly by the Garbage collector of the .NET Framework.
IV. We cannot overload or inherit destructors.
V. We cannot define destructors in struts.
VI. It is called when program exit.
VII. Internally, Destructor called the Finalize method on the base class of object.

Syntax:

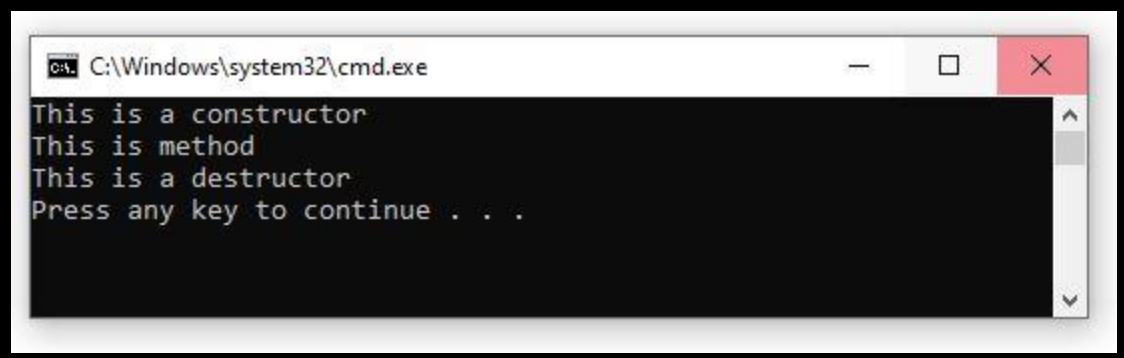
```
Class Example
{
        //Rest of the class
        //members  and method
        ~Example() // Destructor
                {
                      //your code
                }
}
```

Example: Example program to demonstrate destructor.

```
using System;

namespace Destructor
{
    internal class ConsDes
    {
        public ConsDes( string message)
        {
            Console.WriteLine(message);
        }
       public void Test()
        {
            Console.WriteLine("This is method");
        }

            ~ConsDes()
        {
            Console.WriteLine("This is a destructor");
            Console.ReadKey();
```

```
        }

        static void Main(string[] args)
        {
            string msg = "This is a constructor";
            ConsDes obj = new ConsDes(msg);
            obj.Test();
        }
    }
}
```

Output:



THE THIS REFERENCE

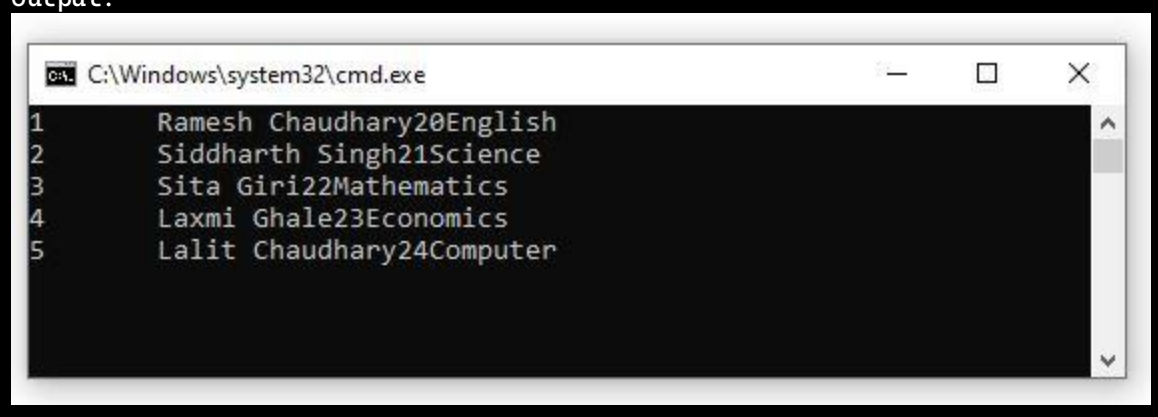
- ☞ The "this" keyword in C# is used to refer to the current instance of the class. It is also used to differentiate between the method parameters and class fields if they both have the same name
- ☞ Another usage of "this" keyword is to call another constructor from a constructor in the same class.
- ☞ Here, for an example, we are showing a record of Students i.e: id, Name, Age, and Subject. To refer to the fields of the current class, we have used the "this" keyword in C#.

```
public Student(int id, String name, int age, String subject) {
  this.id = id;
  this.name = name;
  this.subject = subject;
  this.age = age;
}
```

Example: Example program to demonstrate this keyword.

```
using System;

namespace StudentInfo
{
    internal class Student
    {
        public int id, age;
        public string name, subject;
        public Student( int id, string name, int age, string subject)
        {
            this.id = id;
            this.name = name;
            this.subject = subject;
            this.age = age;
        }
        public void showInfo()
        {
            Console.WriteLine(id + "" + name + "" + age + "" + subject);
        }

        }
    class StudentDetails
    {
        public static void Main(string[] args)
        {
            Student s1 = new Student(01,"\t"+ "Ramesh Chaudhary", 20, "English");
            Student s2 = new Student(02,"\t"+ "Siddharth Singh", 21, "Science");
            Student s3 = new Student(03,"\t"+ "Sita Giri", 22, "Mathematics");
            Student s4 = new Student(04,"\t"+ "Laxmi Ghale", 23, "Economics");
            Student s5 = new Student(05,"\t"+ "Lalit Chaudhary", 24, "Computer");
            s1.showInfo();
            s2.showInfo();
            s3.showInfo();
            s4.showInfo();
            s5.showInfo();
            Console.ReadKey();

        }
    }

}
```

Output:



PROPERTIES (GET & SET)

- Property in C# is a member of a class that provides a flexible mechanism for classes to expose private fields. Internally, C# properties are special methods called accessors. A C# property have two accessors, get property accessor and set property accessor. A get accessor returns a property value, and a set accessor assigns a new value. The **value** keyword represents the value of a property.

- Properties in C# and .NET have various access levels that is defined by an access modifier. Properties can be read-write, read-only, or write-only. The read-write property implements both, a get and a set accessor. A write-only property implements a set accessor, but no get accessor. A read-only property implements a get accessor, but no set accessor.
- In C#, properties are nothing but a natural extension of data fields. They are usually known as 'smart fields' in C# community. We know that data encapsulation and hiding are the two fundamental characteristics of any object oriented programming language. In C#, data encapsulation is possible through either classes or structures. By using various access modifiers like private, public, protected, internal etc it is possible to control the accessibility of the class members.
- Usually, inside a class, we declare a data field as private and will provide a set of public SET and GET methods to access the data fields. This is a good programming practice since the data fields are not directly accessible outside the class. We must use the set/get methods to access the data fields.

The general syntax for declaring properties are as follows:

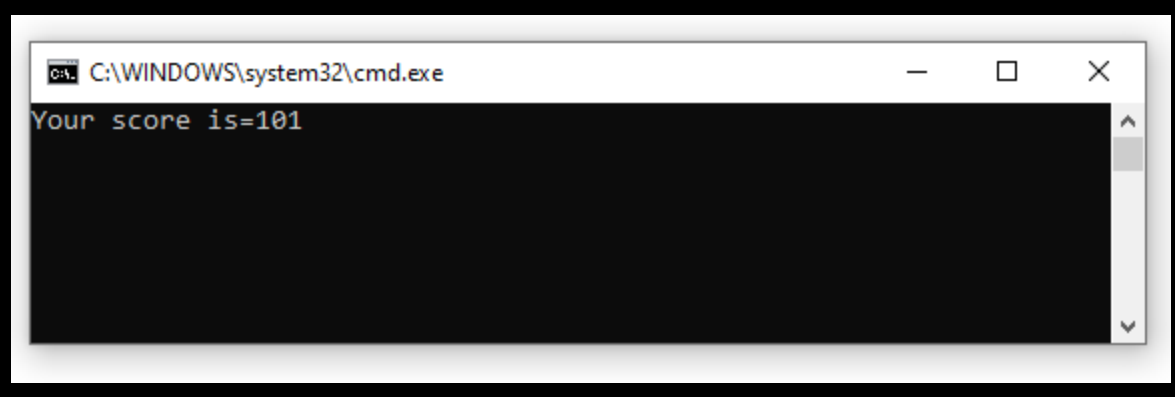
```
<acces_modifier> <return_type> <property_name>
  {
        get
        {
              //body part
         }
        set
        {
              //body part
```

```
      }
}
```

Example-1:

```
using System;

namespace Properties
{
    internal class Program
    {
        private int number;
        public int score              //properties
        {
            get
            {
                return number;

            }
            set
            {
                number = value;
            }
        }

        class Testscore
        {
            static void Main(string[] args)
            {
                Program p = new Program();
                p.score = 101;//assigning the score property causes the 'set'
accessor to be called
                Console.WriteLine("Your score is=" + p.score);//evaluating  the
score property causes the 'get' accessor to be called
                Console.ReadKey();
            }
        }
    }

}
```

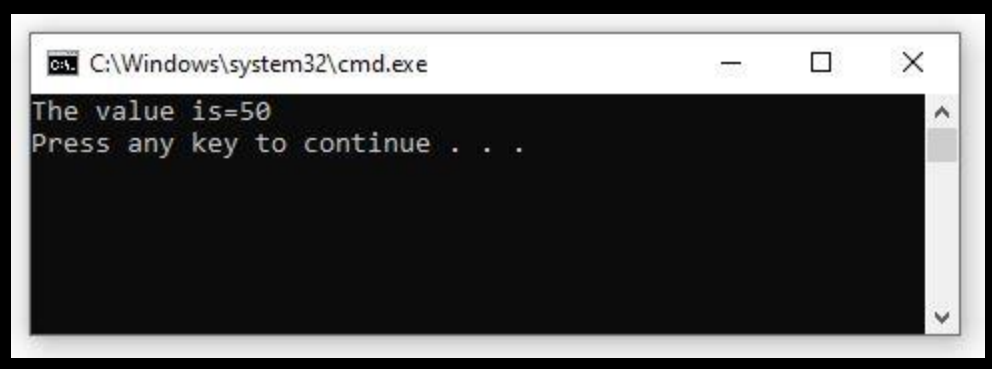
Output:



Example-2: Example program to demonstrate get set properties.

```
using System;

namespace Display
{
    internal class Findnumber
    {
        private int x;
        public void SetX(int i)
        {
            x = i;
        }
        public int GetX()
        {
            return x;
        }
        class MyNumber
        {
            public static void Main(string[] args)
            {
                Findnumber n = new Findnumber();
                n.SetX(50);
                int xVal = n.GetX();
                Console.WriteLine("The value is=" + xVal);


            }
        }
    }

}
```

Output:



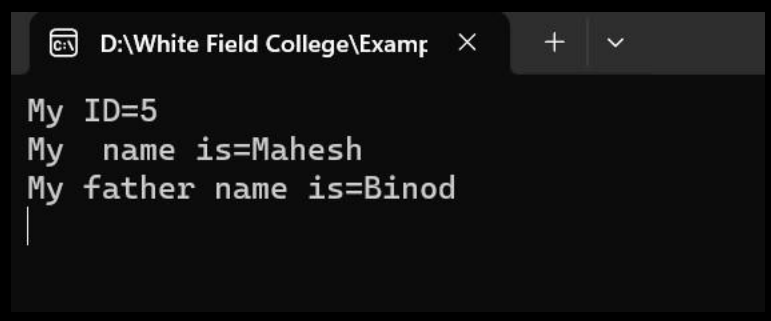
Example-3:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PropertiesExample
```

```
{
    class Student
    {
        private int _StdId;
        private string _Name;
        private string _Fname;

        public int StdId
        {
            set
            {
                if (value <= 0)
                {
                    Console.WriteLine("The ID can not be zero or negative");

                }
                else
                {
                    this._StdId = value;
                }

            }
            get
            {
                return this._StdId;

            }

        }
        public string Name
        {
            set
            {
                if (string.IsNullOrEmpty(value))
                {
                    Console.WriteLine("Please Enter Your Name");
                }
                else
                {
                    this._Name = value;
                }


            }
            get
            {
                return this._Name;

            }
        }
        public string Fname
        {
            set
            {
                if (string.IsNullOrEmpty(value))
                {
                    Console.WriteLine("Plz Enter your father name");
                }
```

```
                else
                {
                    this._Fname = value;
                }

            }
            get
            {
                return this._Fname;
            }
        }
    }
    internal class Program
    {
        static void Main(string[] args)
        {
            Student s = new Student();
            s.StdId = 5;
            s.Name = "Mahesh";
            s.Fname = "Binod";
            Console.WriteLine("My ID="+s.StdId);
            Console.WriteLine("My  name is="+s.Name);
            Console.WriteLine("My father name is="+s.Fname);
            Console.ReadKey();

        }
    }
}
```

Output:



Automatic Properties

- ☞ Automatic property in C# is a property that has backing field generated by compiler. It saves developers from writing primitive getters and setters that just return value of backing field or assign to it.
- ☞ The most common implementation for a property is a getter and setter that simply reads and writes to a private field of the same type as the property.

Example:

Example program to demonstrate automatic properties.

```
using Automatic;
using System;

namespace Automatic
{
    internal class Checkautomatic
    {
        public int a { get; set; }
        public int b { get; set; }
        public int prod
        {
            get { return a* b; }
    }
        }
    class Test
    {
        static void Main(string[] args)
        {
    Checkautomatic ca = new Checkautomatic();
    ca.a = 5;
    ca.b = 4;
    Console.WriteLine("The prod of two number is=" + ca.prod);
    Console.ReadKey();

        }
    }
}
```

Output:

INDEXERS

- An indexer is a special type of property that allows a class or a structure to be accessed like an array for its internal collection.
- The syntax for using indexers is like that for using arrays, except that the index arguments can be of any types.

- C# indexers are usually known as smart arrays.
- A C# indexer is a class property that allows you to access member variable of a class or struct using the features of an array.
- In C#, indexers are created using **this** keyword.
- Indexer in C# are applicable on both classes and structs.
- Defining an indexer allows you to create a class like that can allows its items to be accessed an array. Instances of that class can be accessed using the [ ] array access operator.

Syntax:

<modifier> <return type> this [list of arguments]

{

get

{

//your get block of code

}

set

{

//your set block of code.

}

}

Where.

<modifier> : It can be private, public protected or internal.

<return type>: It can be valid C# type.

this: this is special keyword in C# to indicate the object of the current class.

[list of arguments]: The formal-argument list specifies the parameters of the indexer.

Example:

Example program to demonstrate indexers

```
using Indexer;
using System;
```

```
namespace Indexer
{
    internal class Program
    {
        class IndexerClass
        {
            private string[] names = new string[7];
            public string this[int i]
            {
                get
                {
                    return names[i];
                }
                set
                {
                    names[i] = value;
                }
            }
        }
        static void Main(string[] args)
        {
            IndexerClass week = new IndexerClass();
            week[0] = "SUNDAY";
            week[1] = "MONDAY";
            week[2] = "TUESDAY";
            week[3] = "WEDNESDAY";
            week[4] = "THURSDAY";
            week[5] = "FRIDAY";
            week[6] = "SATURDAY";
            Console.WriteLine("The name of week are:");
            for (int i = 0; i < 7; i++)
            {
                Console.WriteLine(week[i]);
            }

            Console.ReadKey();
        }

    }
}
```

Output:

Static Classes

- In C#, a static class is a class that can't be instantiated.
- The main purpose of the class is to provide blueprints of its inherited classes.
- A static class is created using the **"static"** keyword in C#.
- A static class can contain static members only.
- We can't create an object for the static class.

Syntax for declaring static class

static **class** class_name

{

//static data members

// static methods

}

Note: if we declare any members of a class as **static** we can access it without creating object of that class.

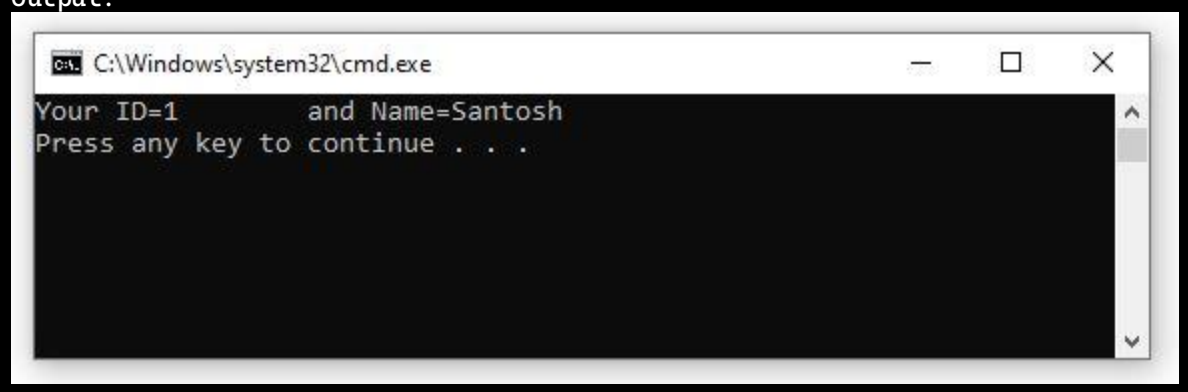
Example:

Example program to demonstrate static class.

```
using Indexer;
using System;

namespace Indexer
{
    internal class Program
    {
        static class Person
        {
            public static int id;
            public static string name;
            public static void Display()
            {
                Console.WriteLine("Your ID={0}\t and Name={1}", id, name);
            }

        }
        class TestProgram
        {
            static void Main(string[] args)
            {
                Person.id = 001;
                Person.name = "Santosh";
```

```
                Person.Display();
            }
        }
    }
}
```

Output:



FINALIZERS

- ☞ Finalize method is also called a **destructor** of the class.
- ☞ Finalizes are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector.
- ☞ The syntax for a finalizer is the name of the class prefixed with the ~ symbol.
- ☞ Finalizers cannot be defined in structs. They are only used with classes.
- ☞ A class can only have one finalizer.
- ☞ Finalizers cannot be inherited or overloaded.
- ☞ Finalizers cannot be called. They are invoked automatically.
- ☞ A finalizer does not take modifiers or have parameters.

Syntax:

```
class Final
{
    ~ Final()
        {
            //code here
        }
}
```

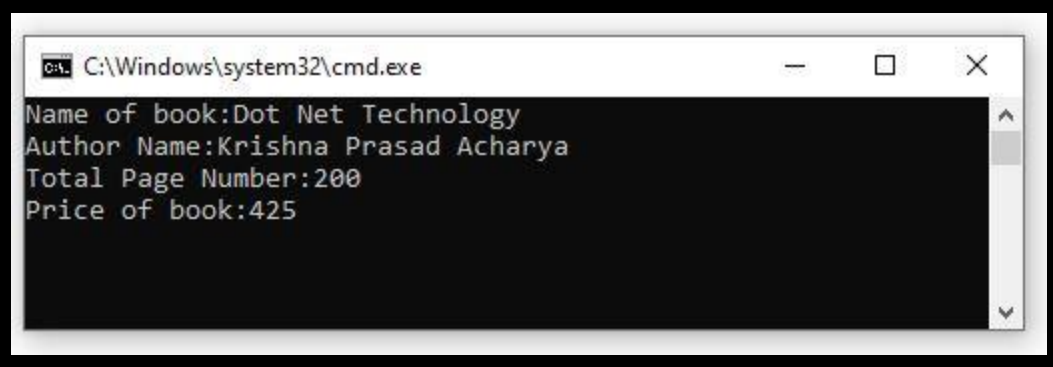
STRUCTS

- ☞ The struct (structure) is like a class in C# that is used to store data. However, unlike classes, a struct is a value type.
- ☞ It initialize with the keyword **struct.**
- ☞ The key difference of structure and class are:
  - ❖ A struct is a value type whereas a class is a reference type.
  - ❖ A struct does not support inheritance.

Example program to demonstrate structure

```
using System;

namespace Sturcture
{
    struct book
    {
        public string book_title;
        public string author;
        public int page;
        public float price;
    };
    internal class Program
    {
        public class TestStructure
        {


            static void Main(string[] args )
            {
                book b1;
                b1.book_title = "Dot Net Technology";
                b1.author = "Krishna Prasad Acharya";
                b1.page = 200;
                b1.price = 425;
                Console.WriteLine("Name of book:{0}", b1.book_title);
                Console.WriteLine("Author Name:{0}", b1.author);
                Console.WriteLine("Total Page Number:{0}", b1.page);
                Console.WriteLine("Price of book:{0}", b1.price);
                Console.ReadKey();
            }
        }
    }
}
```

Output:



ACCESS MODIFIER

- Access modifier in C# are used to specify the scope of accessibility of a member of a class or type of the class itself.
- Access modifiers are an integral part of object-oriented programming.
- It is used to implement encapsulation of OOP.
- Access modifiers allow you to define who does or who doesn't have access to certain features.
- In C# there are 6 different types of Access Modifiers:

Modifiers	Description
public	There are no restrictions on accessing public members.
private	Access is limited to within the class definition. This is the default access modifier type if none is formally specified.
protected	Access is limited to within the class definition and any class that inherit from the class.
internal	Access is limited exclusively to classes defined within the current project assembly.
protected internal	Access is limited to the current assembly and types derived from the containing class. All members in current project and all members in derived class can access the variables.
private protected	Access is limited to the containing class or types derived from the containing class within the current assembly.

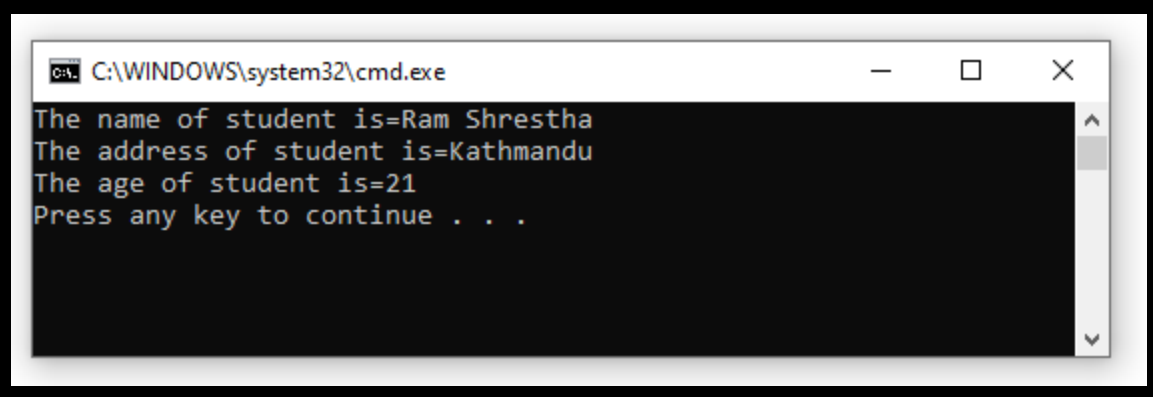
Example:

Example program to demonstrate access modifier (private, public, internal).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Threading.Tasks;

namespace StudentDetails
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Student st = new Student();
            st.name = "Ram Shrestha";
            st.address = "Kathmandu";
            st.age = 21;
            st.ShowDetails();

        }

    }
    class Student
    {
         internal string name;
         internal string address;
         internal int age;

         internal void ShowDetails()
        {
            Console.WriteLine("The name of student is=" + name);
            Console.WriteLine("The address of student is=" + address);
            Console.WriteLine("The age of student is=" + age);
        }

    }

}
```

Output:



INHERITENCE

- ☞ The process by which one class acquires the properties (data members) and functionalities (methods) of another class is called inheritance.
- ☞ It is a mechanism by which one class is allow to inherit the features (field and method) of another class.
- ☞ The aim of inheritance is to provide the reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be extended from the another class.
- ☞ The idea behind inheritance in C# is that you can create new classes that are built upon existing classes.
- ☞ When you inherit from an existing class, you can reuse methods and field of the parent class. Moreover, you can add new methods and fields in your current class also.

Use of Inheritance:

- ☞ Inheritance is used in C# for the following purpose:
  - ❖ For method Overriding (so runtime polymorphism can be achieved).
  - ❖ For Code Reusability.

Terminology in Inheritance:

- ❖ **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- ❖ **Sub Class/ Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extend class or child class.
- ❖ **Super Class/ Parent Class:** Super class is the class from where a subclass inherits the features. It is also called a base class or parent class.
- ❖ **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

Important Features of Inheritance in C#:

I. **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.

II. 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#.

III. 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.

IV. 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 according its private fields, then a subclass can inherit.

Syntax for declaring inheritance in C#

```
class    Subclass_name  : Suprclass_name
{
        //methods and fields
}
```

Where,

The **:** indicates that you are making a new class that derives from an existing class. The meaning of **"extends"** is to increase the functionality.

In the terminology of C#, a class which is inherited is called a parent or superclass and the new class is called child or subclass.

Types of Inheritance in C#

- ❖ On the basis of class, there can be three types of inheritance in Java: single, multilevel and hierarchical.
- ❖ In C# programming, multiple and hybrid inheritance is supported through **interface** only.

Single Inheritance

- ☞ Single Inheritance refers to a child and parent class relationship where a class extends another class.
- ☞ In single inheritance, subclasses inherit the features of one superclass. In the figure below, the class **A** serves as a base class for the derived class **B**.

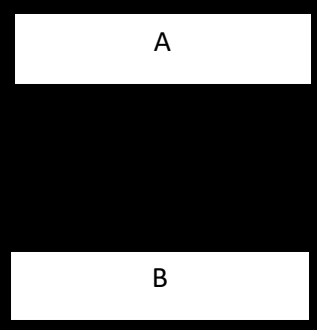


Fig: Single Inheritance

Syntax:

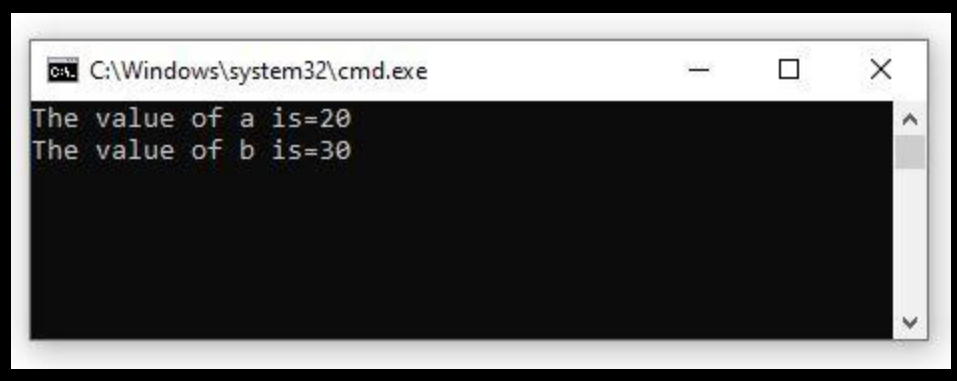
```
class A         // base class
{
       //data members and methods
}
class B : A     //derived class
{
       //data members and methods
}
```

Example: Example program to demonstrate the single inheritance.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Inheritance
{
    internal class Program
    {
        class A
        {
```

```
            public int a = 20, b = 30;
        }
        class B : A
        {
            public void Test()
            {
                Console.WriteLine("The value of a is=" + a);
                Console.WriteLine("The value of b is=" + b);
            }

        }
        class Inherit //derived class
        {
            static void Main(string[] args)
            {
                B obj = new B();
                obj.Test();
                Console.ReadKey();
            }
        }
    }
}
```

Output:



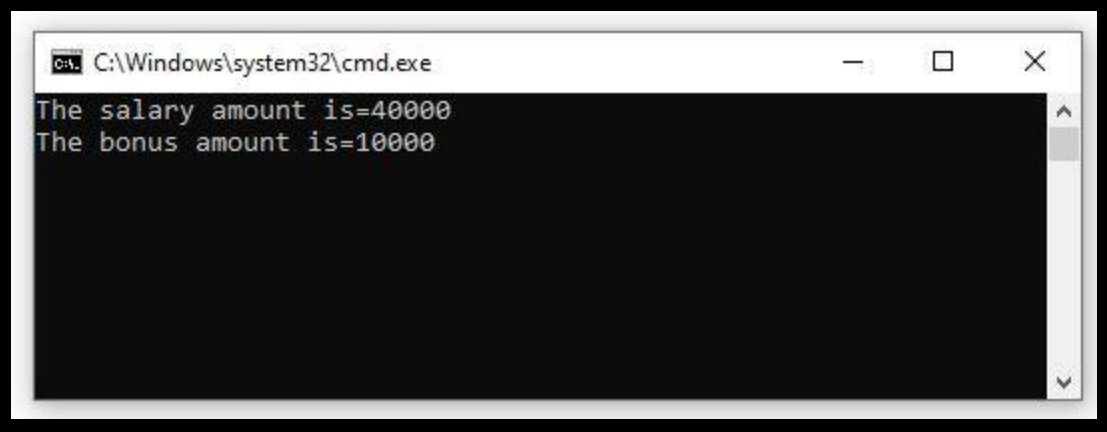
Example-1:

Write a C# program to create base class name employee has a salary 40,000 and who is a programmer get bonus 10,000 pre month.

```
using System;

namespace Inheritance
{
    internal class Program
    {
        class Employee
        {
            public int salary = 40000;
        }
        class Programmar : Employee
```

```
        {
            public int bonus = 10000;
            public void Display()

            {
                Console.WriteLine("The salary amount is=" + salary);
                Console.WriteLine("The bonus amount is=" + bonus);
            }

        }
        class Inherit
        {
            static void Main(string[] args)
            {
                Programmar obj = new Programmar();
                obj.Display();
                Console.ReadKey();
            }
        }
    }
}
```

Output:



The **base** Keyword

- We can use the **base** keyword to access the fields of the base class within derived class.
- It is useful if base and derived classes have the same fields.
- If derived class does not define same field, there is no need to use base keyword. Base class field can be directly accessed by the derived class.

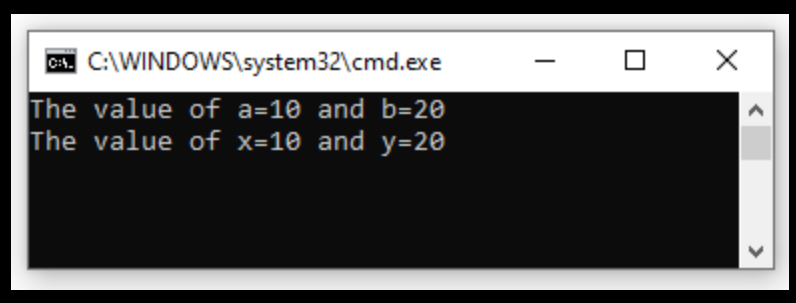
Use of base keyword:

- Generally, there are two uses of **base** keyword.
  - To call base constructor from a derived class constructor.
  - To call a base class method which is overridden in the derived class.

To call base constructor from a derived class constructor.

```
using System;
namespace Useinterface
{
     class Base
    {
       public Base(int a, int b)
        {
            Console.WriteLine("The value of a={0} and b={1}", a, b);
        }
    }
    class Derived:Base
    {
        public Derived(int x, int y): base(x,y)
        {
            Console.WriteLine("The value of x={0} and y={1}", x, y);

        }
    }

    class BaseEx
    {

        static void Main(string[] args)
        {
            new Derived(10, 20);
            Console.ReadKey();

        }
    }

}
```

Output:

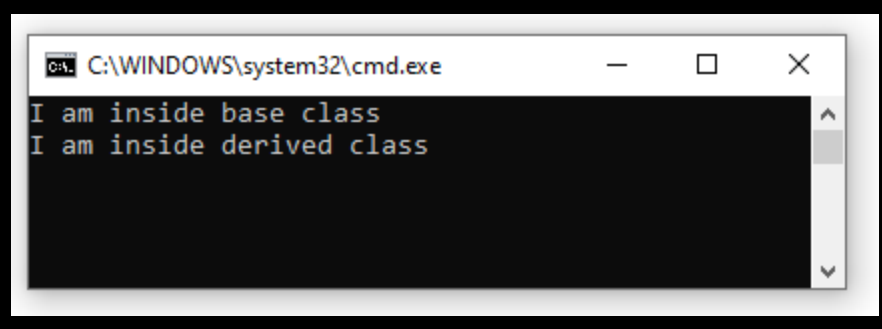


Calling a base class method which is overridden in derived class

```
using System;

namespace Useinterface
```

```
{
     class Base
    {
       public virtual void BaseMethod()
        {
            Console.WriteLine("I am inside base class");
        }
    }
    class Derived:Base
    {
        public override void BaseMethod()//Overridden
        {
            base.BaseMethod();
            Console.WriteLine("I am inside derived class");

        }
    }

    class BaseEx
    {
        static void Main(string[] args)
        {
            Derived obj = new Derived();
            obj.BaseMethod();
            Console.ReadKey();

        }
    }

}
```

Output:



Multilevel Inheritance

- ☞ Multilevel inheritance refers to a child and parent class relationship where a class extends the child class. For example: class C extends class B and class B extends class A.

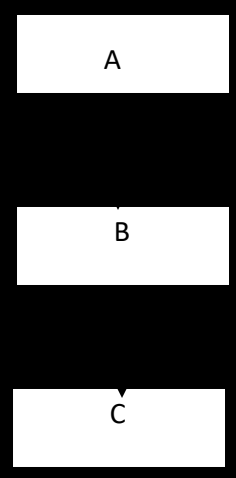


Fig: Multilevel Inheritance

Note: In the above figure, 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.

Syntax:

Class A

{

//data member and methods

}

Class B: A

{

//data members and methods

}

Class C: B

{

//data members and methods

}

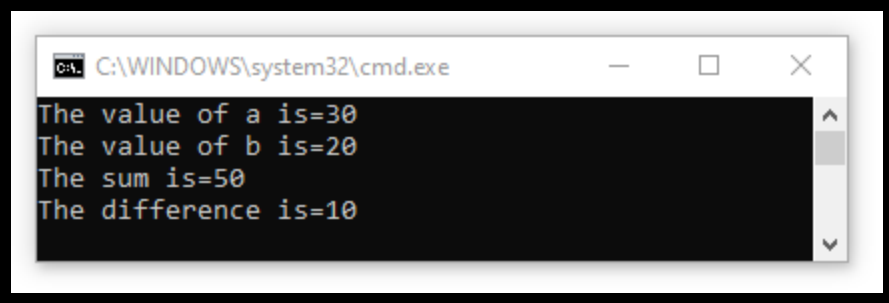
Example:

Example program to demonstrate multilevel Inheritance

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

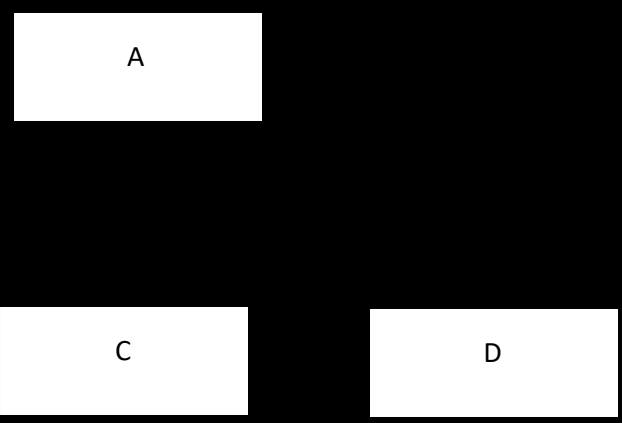
```
namespace Multilevel
{
    class A
    {
        public int a, b, c;
        public void ReadData(int a,int b)
        {
            this.a = a;
            this.b = b;
        }
        public void Display()
        {
            Console.WriteLine("The value of a is=" + a);
            Console.WriteLine("The value of b is=" + b);
        }

    }
    class B :A
    {
        public void add()
        {
            base.c = base.a + base.b;
            Console.WriteLine("The sum is=" + base.c);
        }
    }
    class C : B
    {
        public void Sub()
        {
            base.c = base.a - base.b;
            Console.WriteLine("The difference is=" + base.c);
        }
    }
    class Mlevel
    {
        internal class Program
        {
            static void Main(string[] args)
            {
                C obj = new C();
                obj.ReadData(30, 20);
                obj.Display();
                obj.add();
                obj.Sub();
                Console.ReadKey();
            }
        }
    }
}
```

Output:



Hierarchical Inheritance

- Hierarchical inheritance refers to a child and parent class relationship where more than one class extends the same class. For example: class B, C & D extends the same class A.
- In this inheritance one class serves as a superclass (base class) for more than one subclass.



Syntax:

```
Class  A
{
    // data members  & methods
}
Class B : A
{
```

//data members & methods

}

Class C: A

{

//data members & methods

}

Class D : A

{

//data member & methods

}

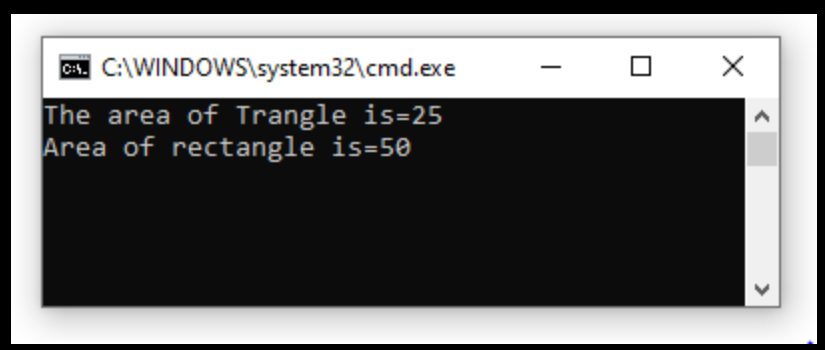
Example:

Example program to demonstrate the hierarchical inheritance.

```
using System;

namespace Multilevel
{
    class Program
    {
        public int dim1, dim2;
        public void ReadDimension(int dim1, int dim2)
        {
            this.dim1 = dim1;
            this.dim2 = dim2;
        }
    }
    class Rectangle : Program
    {

        public void AreaRec()
        {
            base.ReadDimension(10, 5);
            int area = base.dim1 * base.dim2;
            Console.WriteLine("Area of rectangle is=" + area);
        }

    }
    class Trangle : Program
    {
        public void AreaTri()
        {
            base.ReadDimension(10, 5);
            double area = 0.5 * base.dim1 * base.dim2;
            Console.WriteLine("The area of Trangle is=" + area);
        }
```

```
    }
    class Inheritance
    {
        static void Main(string[] args)
        {
            Trangle tri = new Trangle();
            tri.AreaTri();
            Rectangle rec = new Rectangle();
            rec.AreaRec();
            Console.ReadLine();

        }

    }
}
```

Output:



Multiple Inheritance

- ☞ When one class extends from more than one classes then this is called multiple inheritance. For example: class C extends from class A and B then this type of inheritance is known as multiple inheritance.

Note:

C# does not allow multiple inheritance. We can use **interfaces** instead of classes to achieve the same purpose.

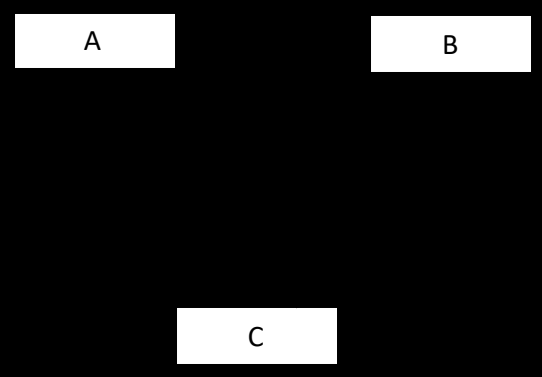


Fig: Multiple Inheritance

Syntax:

```
interface    A
{
    //methods (only signature)
}
Class   B
{
   //data members & methods
}
Class   C : B,A
{
    //data members and methods
}
```

Hybrid Inheritance

- ☞ It is the combination of two or more of the above types of inheritance. Since C# does not support multiple inheritance with classes, the hybrid inheritance is also not possible with classes. In C#, we can achieve hybrid inheritance only through Interfaces.

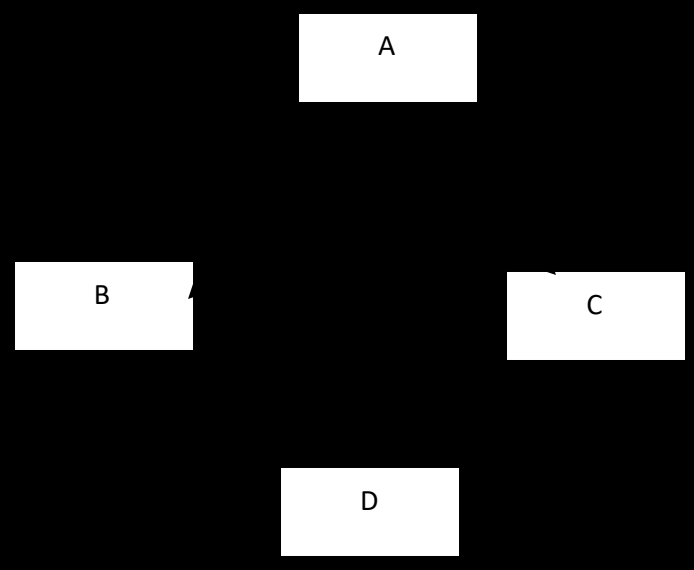


Fig: Hybrid Inheritance

Syntax:

```
interface     A
{
    // code here
}
Class    B :  A
{
}
Class    C  :  B
{
  // code here
}
}
Class   D  : C, A
{
    // code here
}
```

ABSTRACT IN C#

- ☞ 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:
  - ❖ Abstract class
  - ❖ Interface
- ☞ Abstract class and interface both can have abstract methods which are necessary for abstraction.

Abstract Class

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

Syntax for declaration of abstract class

abstract class class_name

{

//data member and properties

// method

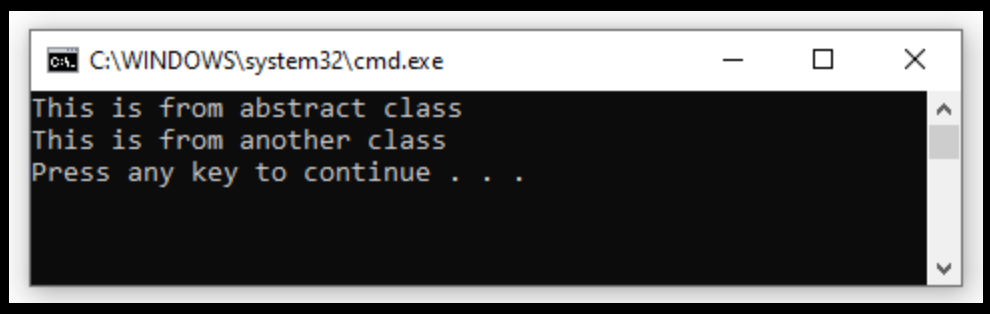
}

Example:

Example program to demonstrate abstract class.

```
using System;

namespace Useinterface
{
    abstract class A
    {
        public void MessageA()
        {
            Console.WriteLine("This is from abstract class");
        }

    }
    class B : A
    {
        public void MessageB()
        {
            Console.WriteLine("This is from another class");
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            B obj = new B();
            obj.MessageA();
            obj.MessageB();

        }
    }

}
```

Output:



Abstract Method/Abstract Member

- A method which is declared with **abstract** keyword 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.
- When the derived class inherits the abstract method from the abstract class, it must override the abstract method.
- This requirement is enforced at compile time and is also called dynamic polymorphism.

Note: Abstract members are used to achieve total abstraction.

Syntax for using abstract method

<access modifier> abstract<return_type>method name(parameters)

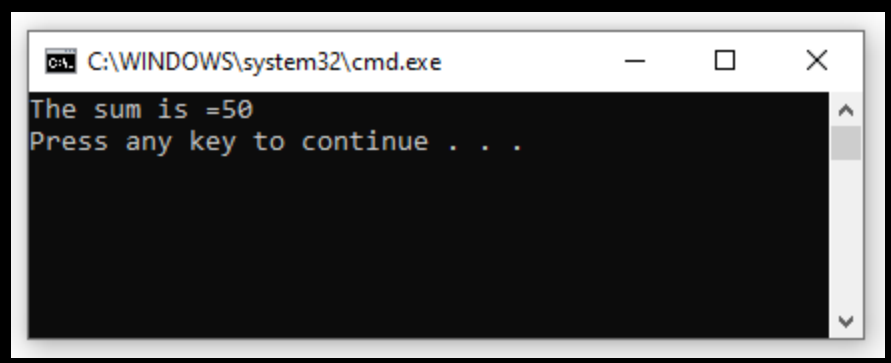
Example:

Example program to demonstrate abstract class.

```
using System;
namespace Useinterface
{
    abstract class A
    {
        public abstract int AddData(int a, int b);
    }
    class B : A
    {
        public override int AddData(int a, int b)
        {
            return a + b;
        }
    }
    class Program
    {
```

```
        static void Main(string[] args)
        {
            B obj = new B();
            int res = obj.AddData(30, 20);
            Console.WriteLine("The sum is =" + res);

        }
    }

}
```

Output:



INTERFACE IN C#

- ☞ 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.

Characteristics of Interface

- ❖ Interface can contain declarations of method, properties, indexers, and events.
- ❖ Interface cannot include private, protected, or internal members. All the members are public by default.
- ❖ Interface cannot contain fields, and auto-implemented properties.
- ❖ A class or a struct can implement one or more interfaces implicitly or explicitly. Use public modifier when implementing interface implicitly, whereas don't use it in case of explicit implementation.
- ❖ Implement interface explicitly using **InterfaceName.MemberName.**

- An interface can inherit one or more interfaces.

Syntax of Interface in C#

interface interface_name

{

//method signature

}

class class_name : interface_name

{

//method implementation

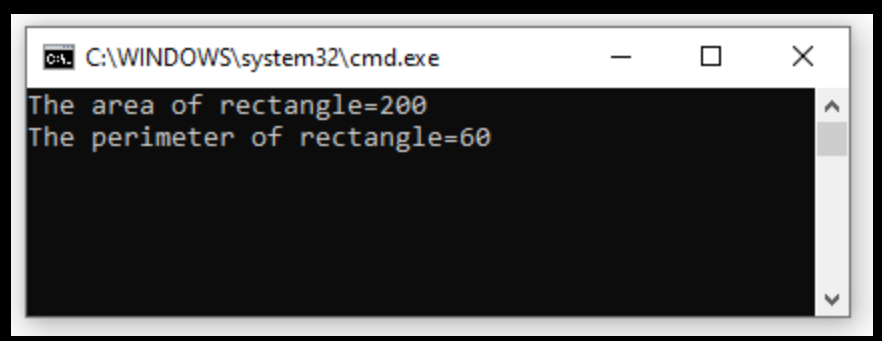
}

Example:

```
using System;

namespace Useinterface
{
    interface A
    {
    interface A
    {
        void GetData(int l, int b);
        int CalculateArea();
        int CalculatePerimeter();
    }
    class B : A
    {
        int l, b;
        public void GetData(int l, int b)
        {
            this.l = l;
            this.b = b;
        }
        public int CalculateArea()
        {
            int area = l * b;
            return area;
        }
        public int CalculatePerimeter()
        {
            int peri = 2 * (l + b);
            return peri;
        }

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

```
            B obj = new B();
            obj.GetData(10, 20);
            Console.WriteLine("The area of rectangle=" + obj.CalculateArea());
            Console.WriteLine("The perimeter of rectangle=" +
obj.CalculatePerimeter());
            Console.ReadKey();

        }
    }

}
```

Output:



POLYMORPHISM

- ☞ The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a Greek word.
- ☞ There are two types of polymorphism in C#: compile time polymorphism and runtime polymorphism.
- ☞ Compile time polymorphism is achieved by method overloading and operator overloading in C#. It is also known as static binding or early binding.
- ☞ Runtime polymorphism in achieved by method overriding which is also known as dynamic binding or late binding.

Polymorphism in C#

Compile Timed
- ❖ **Method Overloading**
- ❖ **Operator Overloading**

Runtime
- ❖ **Method Overriding**
- ❖ **Virtual Function**

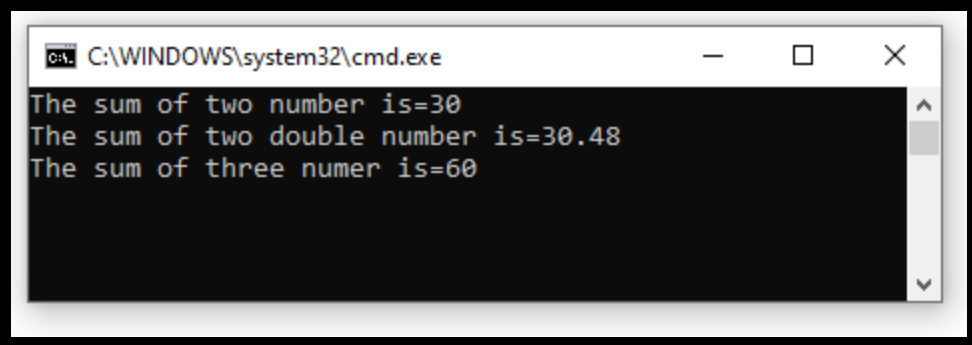
Method Overloading

- ☞ Method overloading can be achieved by changing number of parameters, type of parameter and order of parameter.

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Useinterface
{
    class Addition
    {
        public void Sum(int a, int b)
        {
            int s = a + b;
            Console.WriteLine("The sum of two number is=" + s);
        }
        public void Sum(double a, double b)
        {
            double d = a + b;
            Console.WriteLine("The sum of two double number is=" + d);
        }
        public void Sum(int a, int b, int c)
        {
            int t = a + b + c;
            Console.WriteLine("The sum of three numer is=" + t);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Addition obj = new Addition();
            obj.Sum(20, 10);
            obj.Sum(10.25, 20.23);
            obj.Sum(10, 20, 30);
            Console.ReadKey();

        }
    }

}
```

Output:



Method Overriding

- ☞ Method Overriding is a technique that allows the invoking of functions from another class (base class) in the derived class.
- ☞ Creating a method in the derived class with the same signature as a method in the base class is called as method overriding.
- ☞ Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- ☞ When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.
- ☞ Method overriding is one of the ways by which C# achieve Run Time Polymorphism(Dynamic Polymorphism).
- ☞ The overridden base method must be virtual, abstract, or override.

Rules for Overriding

- ❖ A method, property, indexer, or event can be overridden in the derived class.
- ❖ Static methods cannot be overridden.
- ❖ Must use virtual keyword in the base class methods to indicate that the methods can be overridden.
- ❖ Must use the override keyword in the derived class to override the base class method.

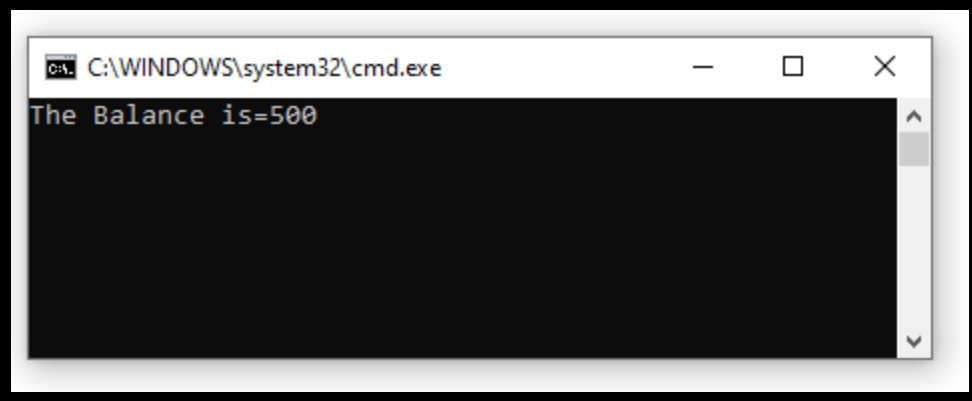
Example:

Example program to demonstrate method overriding.

```
using System;

namespace Useinterface
```

```
{
   public class Account
    {
        public virtual int balance()
        {
            return 10;
        }
    }
    public class Amount : Account
    {
        public override int balance()
        {
            return 500;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Amount obj = new Amount();
            Console.WriteLine("The Balance is=" + obj.balance());
            Console.ReadKey();

        }
    }

}
```

Output:



Virtual Method

- A virtual method is a method that can be redefined in derived classes.
- In C#, a virtual method has an implementation in a base class as well as derived class.
- It is used when a method's basic functionality is the same but sometimes more functionality is needed in the derived class.
- A virtual method is created in the base class that can be overridden in the derived class. We create a virtual method in the base class using the **virtual**

keyword and that method is overridden in the derived class using the override keyword.

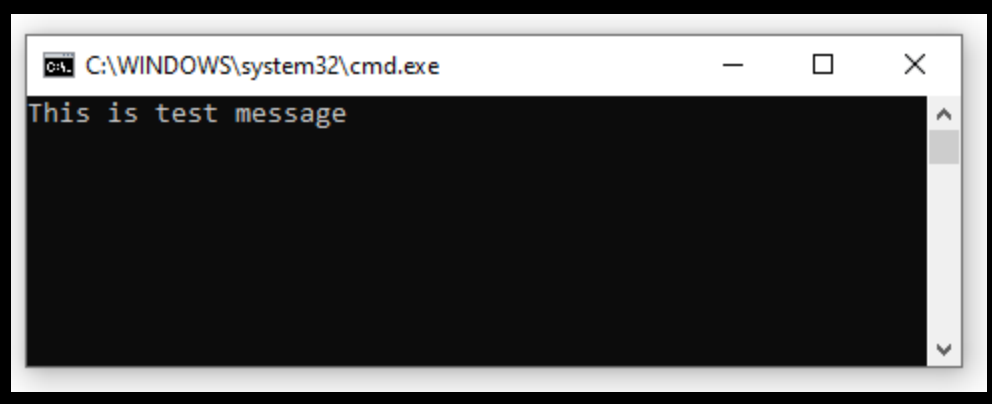
Features of Virtual Method

- By default, methods are non-virtual. We can't override a non-virtual method.
- We can't use the virtual modifier with the static, abstract, private or override modifiers.
- If a class is not inherited, behavior of virtual method is same as non-virtual method, but in case of inheritance it is used for method overriding.

Behavior of virtual method without using inheritance

```
using System;

namespace Virtual_Method
{
    internal class Program
    {
        public virtual void message()
        {
            Console.WriteLine("This is test message");
        }
        static void Main(string[] args)
        {
            Program obj = new Program();
            obj.message();
            Console.ReadKey();
        }
    }
}
```

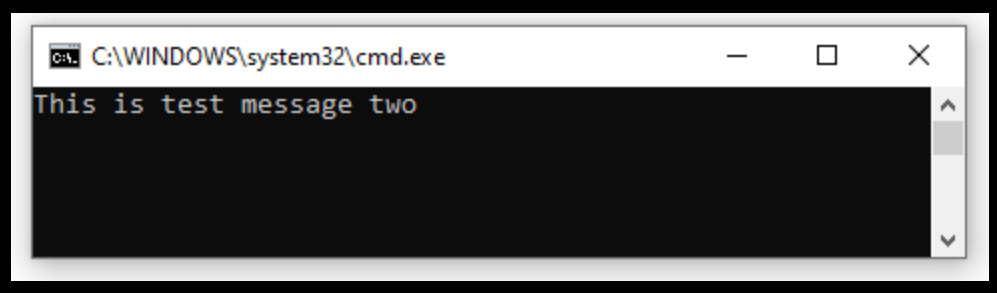
Output:



Behavior of virtual method with inheritance (used for overloading)

```
using System;

namespace Virtual_Method
{
    internal class A
    {
        public virtual void message()
        {
            Console.WriteLine("This is test message");
        }
        class B:A
        {
            public override void message()
            {
                Console.WriteLine("This is test message two");
            }
        }
        static void Main(string[] args)
        {
            B obj = new B();
            obj.message();
            Console.ReadKey();
        }
    }
}
```

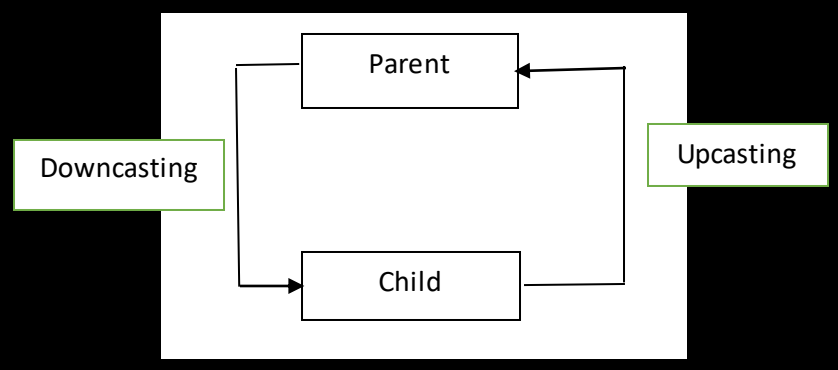
Output:



UPCASTING AND DOWNCASTING

- Upcasting converts an object of a specialized type to a more general type. An upcast operation creates a base class reference from a subclass reference.

For example:
Stock msft = new Stock();
Asset a = msft; // Upcasting



☞ Downcasting converts an object from a general type to a more specialized type. A downcast operation creates a subclass reference from a base class reference.

For example:

```
Stock msft = new Stock();
Asset a = msft; // Upcast
Stock s = (Stock)a; // Downcast
```

Bank Account

Check Account Saving Account Lottery Account

```
BankAccount    ba1,
               ba2 =   new BankAccount("John",  250.0M,  0.01);
   LotteryAccount la1,
               la2 =   new LotteryAccount("Bent", 100.0M);

   ba1 = la2;                          // upcasting   - OK
//  la1 = ba2;                          // downcasting - Illegal
                                        //   discovered at compile time
//  la1 = (LotteryAccount)ba2;     // downcasting - Illegal
                                       //   discovered at run time
   la1 = (LotteryAccount)ba1;     // downcasting - OK
                                       //   ba1 already refers to a LotteryAccount
```

Example : Example program to demonstrate UPCASTING

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Upcasting
{
    class A
    {


    }
    class B:A
    {

    }
    internal class Program
    {
        static void Main(string[] args)
        {
            A obj = new B();//Upcasting
            //B obj1 = new A();//Downcasting is not allowed in C#
        }
    }

}
```

OPERATOR OVERLOADING

- ☞ The concept of overloading a function can also be applied to operators. 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 types. It enables to make user-defined implementations of various operations where one or both of the operands are of a user-defined class.
- ☞ Only the predefined set of C# operators can be overloaded. To make operations on a user-defined data type is not as simple as the operations on a built-in data type. To use operators with user-defined data types, they need to be overloaded according to a programmer's requirement. An operator can be overloaded by defining a function to it. The function of the operator is declared by using the **operator** keyword.

Syntax:

access specifier className **operator** Operator_symbol (parameters)

{

// Code

}

The following table describes the overloading ability of the various operators available in C# :

OPERATORS	DESCRIPTION
+, -, !, ~, ++, – –	unary operators take one operand and can be overloaded.
+, -, *, /, %	Binary operators take two operands and can be overloaded.
==, !=, =	Comparison operators can be overloaded.
&&,||	Conditional logical operators cannot be overloaded directly
+=, -+, *=, /=, %=, =	**Assignment operators cannot be overloaded.**

Overloading Unary Operators

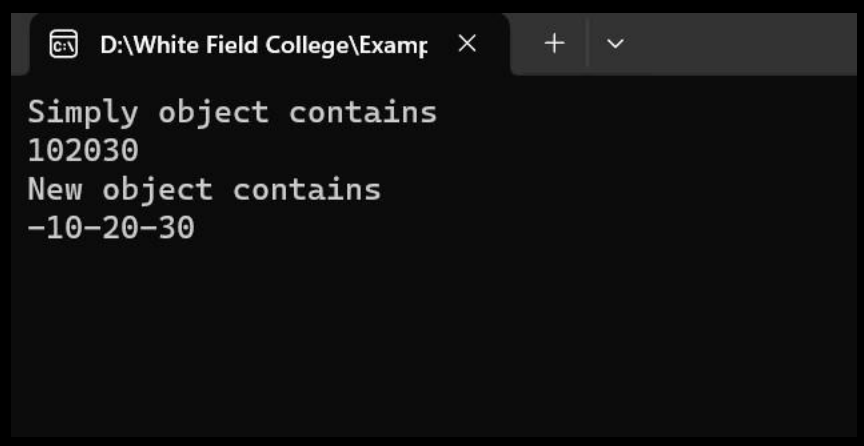
Unary Operator Overloading (-)

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UnaryMinus
{
    class Test
    {
        int x, y, z;
        public Test(int a,int b,int c)
        {
            x = a;
            y = b;
            z = c;

        }
        public void display()
        {
```

```
            Console.WriteLine("" + x + "" + y + "" + z);
        }
        public static Test operator- (Test obj)
        {
            obj.x = -obj.x;
            obj.y = -obj.y;
            obj.z = -obj.z;
            return obj;
        }

    }
    internal class Program
    {
        static void Main(string[] args)
        {
            Test obj = new Test(10, 20, 30);
            Console.WriteLine("Simply object contains");
            obj.display();
            obj = -obj;
            Console.WriteLine("New object contains");
            obj.display();
            Console.ReadKey();

        }
    }
}
```

Output:

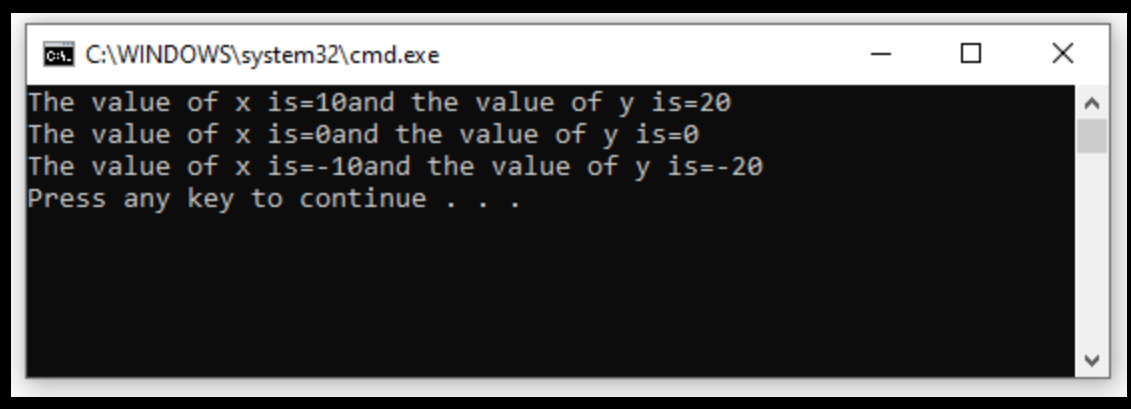


☞ The following program overloads the unary operator inside the class Complex.

```
using System;

namespace Virtual_Method
{
    internal class Complex
    {
        private int x;
```

```
        private int y;
        public Complex()
        {

        }
        public Complex(int i, int j)
        {
            x = i;
            y = j;
        }
        public void ShowXY()
        {
            Console.WriteLine("The value of x is={0}and the value of y is={1}", x,
y);
        }
        public static Complex operator -(Complex c)
        {
            Complex temp = new Complex();
            temp.x = -c.x;
            temp.y = -c.y;
            return temp;


        }



    }
    class Drive
    {
        static void Main(string[] args)
        {
            Complex c1 = new Complex(10, 20);
            c1.ShowXY();
            Complex c2 = new Complex();
            c2.ShowXY();
            c2 = -c1;
            c2.ShowXY();

        }

    }
}
```

Out Put:



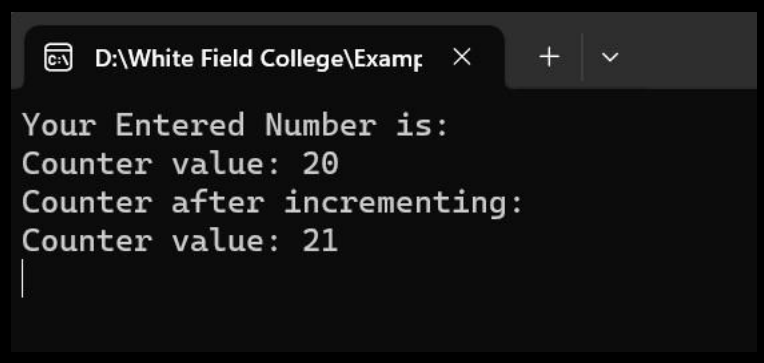
Increment operator (++) Overloading:

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace IncrementOperator
{
    class Counter
    {
        private int value;

        public Counter(int x)
        {
            value = x;
        }

        // Overloading the ++ operator
        public static Counter operator ++(Counter counter)
        {
            counter.value++;
            return counter;
        }

        public void Display()
        {
            Console.WriteLine("Counter value: " + value);
        }
    }

    internal class Program
    {
        static void Main(string[] args)
        {
            Counter obj= new Counter(20);

            Console.WriteLine("Your Entered Number is:");
            obj.Display();
```

```
            // Using the overloaded ++ operator
            obj++;

            Console.WriteLine("Counter after incrementing:");
            obj.Display();
            Console.ReadKey();
        }
    }
}
```

Output:



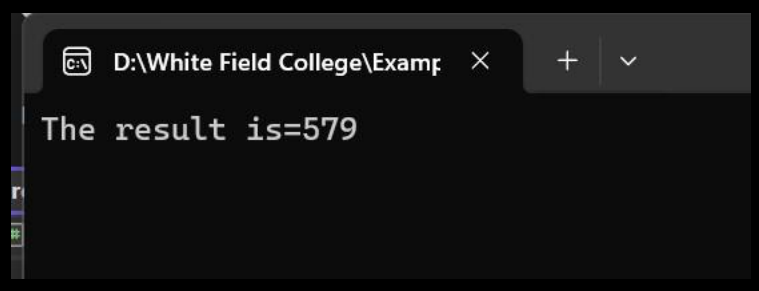
Overloading Binary Operators

- ☞ An overloading binary operator must take two arguments at least one of them must be of the type class or struct, in which the operator is defined.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BinaryOperatorOverloading
{
    class Test
    {
        int x, y, z;
        public Test() {

        }
        public Test(int a, int b, int c)
        {
            x = a;
            y = b;
            z = c;

        }
        public void display()
```

```
        {
            Console.WriteLine("The result is=" + x + "" + y + "" + z);
        }
        public static Test operator +(Test obj1, Test obj2)
        {
            Test obj3 = new Test();
            obj3.x = obj1.x + obj2.x;
            obj3.y = obj1.y + obj2.y;
            obj3.z = obj1.z + obj2.z;
            return obj3;
        }

    }
    internal class Program
    {


        static void Main(string[] args)
        {
            Test onj3 = new Test();
            Test obj1 = new Test(1, 2, 3);
            Test obj2 = new Test(4, 5, 6);
            Test obj3 = new Test();
            obj3 = obj1 + obj2;
            obj3.display();
            Console.ReadKey();
        }
    }
}
```

Output:

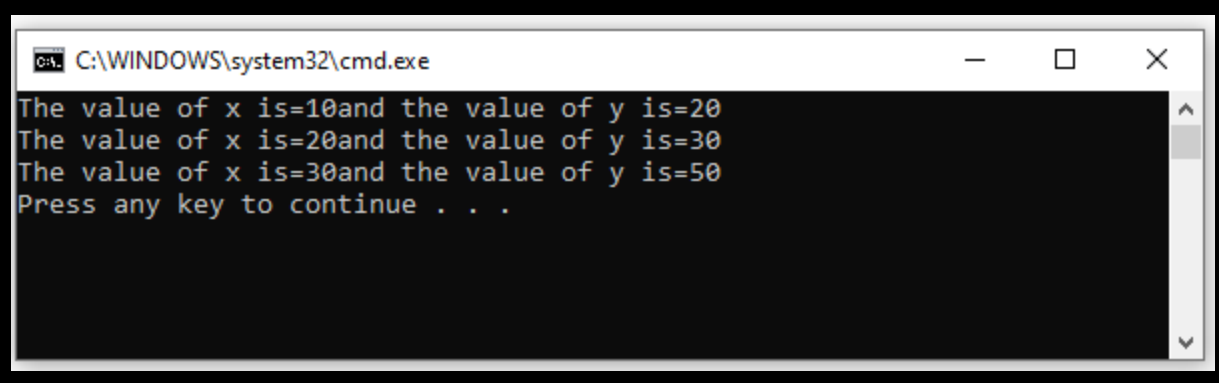


Example:

```
using System;

namespace Virtual_Method
{
    internal class Complex
    {
        private int x;
        private int y;
        public Complex()
        {

        }
        public Complex(int i, int j)
        {
            x = i;
```

```
            y = j;
        }
        public void ShowXY()
        {
            Console.WriteLine("The value of x is={0}and the value of y is={1}", x,
y);


        }
        public static Complex operator + (Complex c1, Complex c2)
        {


            Complex temp = new Complex();
                temp.x = c1.x + c2.x;
            temp.y = c1.y+c2.y;
            return temp;


        }



    }
    class Drive
    {
        static void Main(string[] args)
        {
            Complex c1 = new Complex(10, 20);
            c1.ShowXY();
            Complex c2 = new Complex(20,30);
               Complex c3 = c1 + c2;
            c2.ShowXY();

            c3.ShowXY();

        }

    }
}
```

OutPut:



SEALED FUNCTION AND CLASSES

C# sealed Class

- ☞ Sealed class is used to restrict the inheritance features of object oriented programming.
- ☞ Once is class is defined as a sealed class, this class can not be inherited. In C# **sealed** modifier is used to declare a class as sealed. If a class is derived from sealed class, compiler throws an error.

Characteristics of sealed Class

- ❖ A sealed class is completely opposite to an abstract class.
- ❖ This sealed class cannot contain abstract methods.
- ❖ It should be the bottom most class within the inheritance hierarchy.
- ❖ A sealed class can never be used as a base class.
- ❖ This sealed class is specially used to avoid further inheritance.
- ❖ The keyword **sealed** can be used with classes, instance method, and properties.

Syntax of sealed class

sealed class SealedClass_name{

//code here

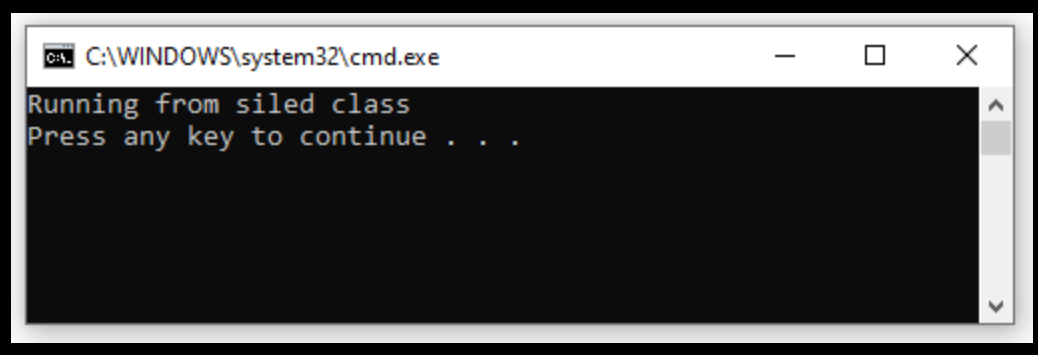
}

Example:

```
using System;
namespace Virtual_Method
{
    sealed class Test
    {
        public void message()
        {
            Console.WriteLine("Running from siled class");
        }

    }
    class Drive
    {
        static void Main(string[] args)
        {
            Test obj = new Test();
            obj.message();
```

```
        }
    }
}
```

Out Put:



BOXING AND UNBOXING

Boxing:

- ☞ It is process of converting a value type to reference type.
  Eg.
  Int, float, character object

BOXIN

Eg.

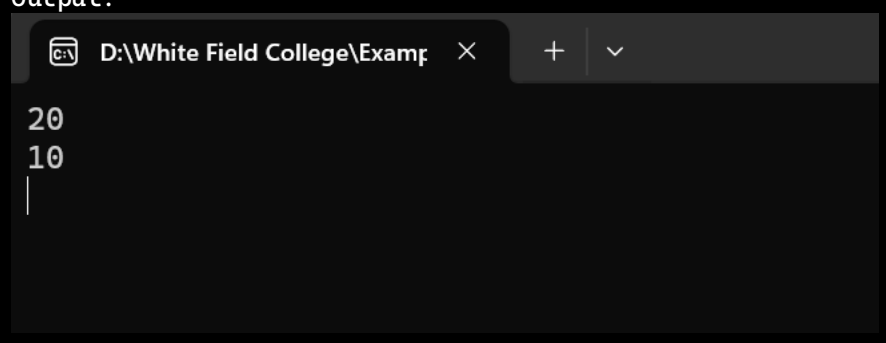
```
int x=10;
object obj = x; // Box the int

x=20;
console.WriteLine(x); //It will print 20
console.WriteLine(obj); // It will print 10
```

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace BoxingUnboxing
{
    internal class Program
```

```
    {
        static void Main(string[] args)
        {
            int x = 10;
            Object obj = x;//Boxing
            x=20;
            Console.WriteLine(x);
            Console.WriteLine(obj);
            Console.ReadKey();


        }
    }
}
```

Output:



UNBOXING

☞ It is process of converting reference type to value type is known as UNBOXING.

Object int, float, char

Unboxing re UNBOXIN peration, by casting the object back to the original value type.

Eg.

int i=10;

Object obj=i;

int j = (int)obj; // unbox the int

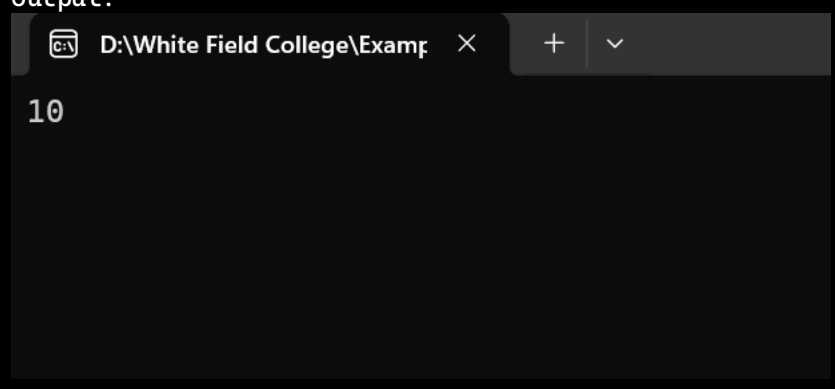
console.WriteLine(j);

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace BoxingUnboxing
{
    internal class Program
    {
        static void Main(string[] args)
        {
            int i = 10;
            Object obj = i;
            int j = (int)obj;//Unboxing the int
            Console.WriteLine(j);
            Console.ReadKey();


        }
    }
}
```

Output:



GENERICS

- Generic introduce in C# 2.0
- Generic allow you to write a class or method that can work with any data type.
- Generic allows you to define a class with place holders for the type of its fields, methods, parameters etc. Generics replace these placeholders with some specific type at compile time. It helps you to maximize code reuse, type safety and performance.
- You can create your own generic interface, class, methods, events and delegates.
- You may get information on the types used in a generic data type at run- time.
- A generic class or method can be defined using angle bracket <>.
- The detailed specification for each collection is found under the **System.Collection.Generic** namespace.

Generic can be applied to the following

- Interface
- Abstract class
- Class
- Method
- Static method
- Property
- Event
- Delegate
- Operator

Advantages of Generic

- Increase the reusability of the code.
- Generic are type safe. You get compile time errors if you try to use a different type of data than the one specified in the definition.
- Generic has a performance advantage because it remove the possibility of boxing and unboxing.

How does work Generic Method

- Generic methods process value whose data types are known only when accessing the variables that store these value.
- A generic method is declare with generic type parameter list enclosed within angular brackets.
- Defining methods with type parameter allow you to call the method with a different type every time.
- You can declare a generic method within generic or non-generic class declarations.

  Generic method can be declare with the following keywords:
  - Virtual
  - Override
  - Abstract

Generic Classes

- The generic class can be defined by putting the <T> sign after the class name.
- It is mandatory to put the "T" word in the Generic type definition. You can use any word in the Test Class <> class declaration.

  Public class TestClass<>

{

}

- ☞ The **System .Collection.Generic** namespace also defines a number of classes that implement many of these key interface. The following table describes the core class type of this namespace.

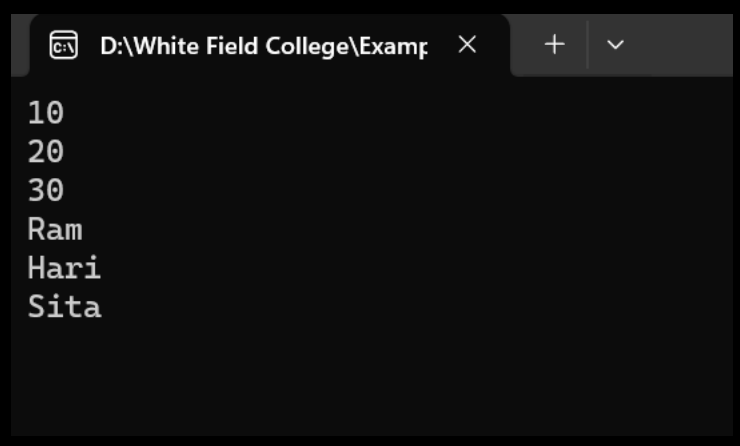
Generic Class	Description
Collection<T>	The basic for a generic collection Comparer compares two generic object for equality.
Dictionary<TKey, Tvalue>	A generic collection of name/value pairs
List<T>	A dynamically resizable list of Items
Queue<T>	A generic implementation of a first-in, first out (FIFO) list
Stack<T>	A generic implementation of a last-in, first out(LOFO)

Example:1

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Generic
{
    class Example
    {
        public static void ShowArray(int[] arr)
        {
            for (int i=0;i<arr.Length;i++)
            {
                Console.WriteLine(arr[i]);
            }
        }
        public static void ShowArray(String[] arr)
        {
            for (int i = 0; i < arr.Length; i++)
            {
                Console.WriteLine(arr[i]);
            }
        }
    }
    internal class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = new int[3];
            numbers[0] = 10;
            numbers[1] = 20;
            numbers[2] = 30;
```

```
            String[] name = new string[3];
            name[0] = "Ram";
            name[1] = "Hari";
            name[2] = "Sita";
            Example.ShowArray(numbers);
            Example.ShowArray(name);
            Console.ReadKey();
        }
    }
}
```

Output:

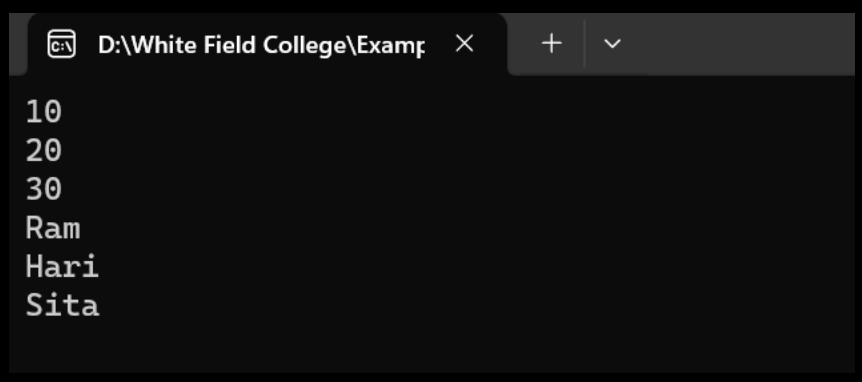


Example-2:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Generic
{
    class Example
    {
        public static void ShowArray<T>(T[] arr)
        {
            for (int i=0;i<arr.Length;i++)
            {
                Console.WriteLine(arr[i]);
            }
        }
        /*
        public static void ShowArray(String[] arr)
        {
            for (int i = 0; i < arr.Length; i++)
            {
                Console.WriteLine(arr[i]);
            }
        }
        */
```

```
    }
    internal class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = new int[3];
            numbers[0] = 10;
            numbers[1] = 20;
            numbers[2] = 30;
            String[] name = new string[3];
            name[0] = "Ram";
            name[1] = "Hari";
            name[2] = "Sita";
            Example.ShowArray(numbers);
            Example.ShowArray(name);
            Console.ReadKey();
        }
    }
}
```

Output:

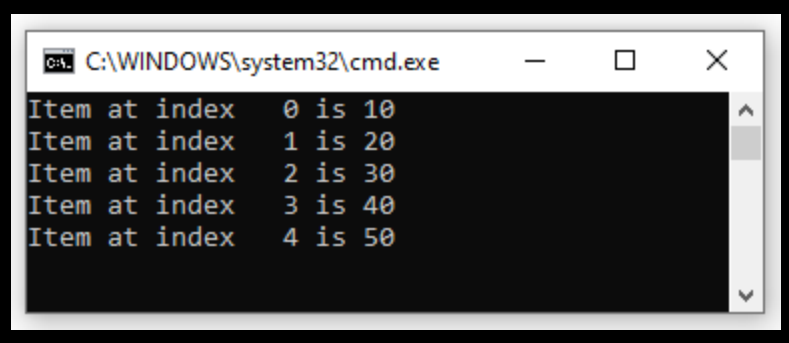


Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;

namespace Virtual_Method
{
    sealed class Test<T>
    {
        T[] t = new T[5];
        int count = 0;
        public void addItem(T item)
        {
            if(count<5)
            {
                t[count] = item;
```

```
            count++;
        }
        else
        {
            Console.WriteLine("Overflow exits");
        }
    }
    public void displayItem()
    {
        for(int i=0;i<count;i++)
        {
            Console.WriteLine("Item at index\t{0} is {1}", i, t[i]);
        }
    }


}
class GenericEx
{
    static void Main(string[] args)
    {
        Test<int> obj = new Test<int>();
        obj.addItem(10);
        obj.addItem(20);
        obj.addItem(30);
        obj.addItem(40);
        obj.addItem(50);
       // obj.addItem(60);
        obj.displayItem();
        Console.ReadKey();


        }

    }
}
```

Output:



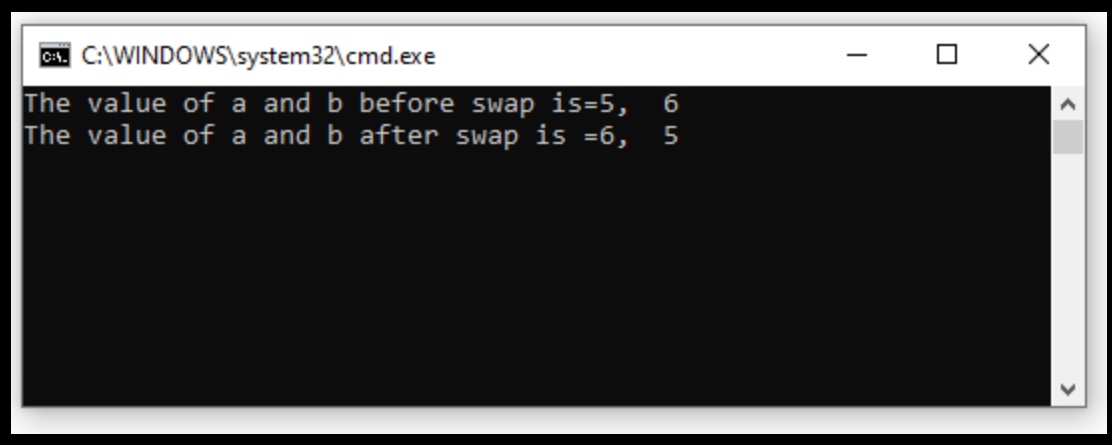
Generic Methods

- ☞ The objective of this example is to build a swap method that can operate on any possible data type (value based or reference based) using a single type parameter. Due to the nature of swapping algorithms, the incoming parameters will be sent by reference via ref keyword.

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;

namespace Virtual_Method
{
    sealed class Test
    {
        static void Swap<T>(ref T a, ref T b)
        {
            T temp;
            temp = a;
            a = b;
            b = temp;
        }
        class GenericEx
        {
            static void Main(string[] args)
            {
                int a = 5, b = 6;
               Console.WriteLine("The value of a and b before swap is={0},\t{1}", a, b);
                Swap<int>(ref a, ref b);
               Console.WriteLine("The value of a and b after swap is ={0},\t{1}", a, b);

                Console.ReadKey();
            }


        }
    }
}
```

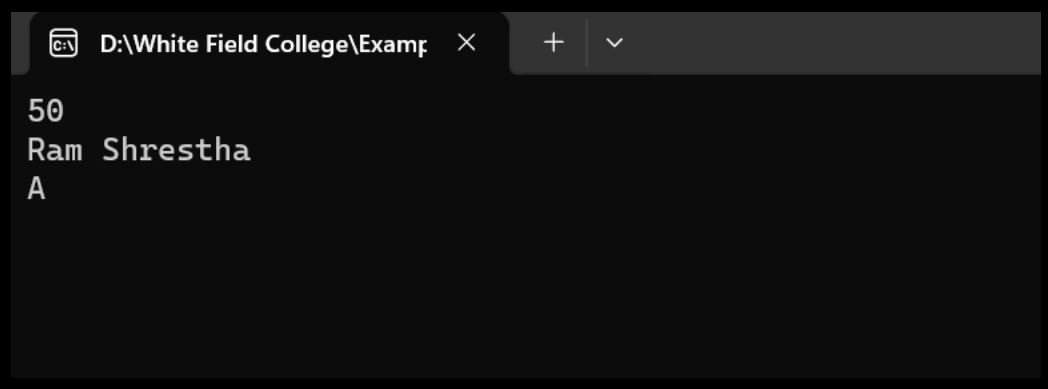
Out Put:



Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Generic
{
    class Example<T>
    {
        T box;
        public Example(T b)
        {
            this.box = b;

        }
        public T getbox()
        {
            return this.box;
        }
    }
    internal class Program
    {
        static void Main(string[] args)
        {
            Example<int> e = new Example<int>(50);
            Example<string> e1 = new Example<string>("Ram Shrestha");
            Example<char> e2 = new Example<char>('A');
            Console.WriteLine(e.getbox());
            Console.WriteLine(e1.getbox());
            Console.WriteLine(e2.getbox());
            Console.ReadKey();


        }
    }
}
```

Output:



QUEUES

Queue is linear data structure that permits insertion of elements at one end and deletion of an element at another end which are rear and front respectively. It represents a first in and first out collection of objects.

Count	Returns the total count of elements in the Queue.

Enqueue	Adds an item into the queue.
Peek	Returns an first item from the queue
Clear	Removes all the items from the queue.

Example:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace QueueApp
{
   class Program
   {
      static void Main(string[] args)
      {
         Queue q = new Queue();
         q.Enqueue('B');
         q.Enqueue('H');
         q.Enqueue('U');
         q.Enqueue('P');
         q.Enqueue('I');
         Console.WriteLine("content of queue :");
         foreach(char c in q)
         {
            Console.Write(c + " ");
         }
         Console.WriteLine();
         Console.WriteLine("Removing some values:");
         char ch = (char)q.Dequeue();
         Console.WriteLine("the removed value is :{0}", ch);
         Console.WriteLine("after removing q element the content of queue is:");
         foreach (char c in q)
         {
            Console.Write(c + " ");
         }
         Console.WriteLine();
         Console.ReadKey();



   }
}
```

}

Stack

stack is linear data structure that permits insertion and deletion of elements are at one end. It represents a last in and first out (LIFO) collection of objects.

Count	Returns the total count of elements in the Stack.

Push	Inserts an item at the top of the stack.
Pop	Removes and returns items from the top of the stack.
Clear	Removes all items from the stack.

Example:

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
namespace StackApp
{
   class Program
   {
      static void Main(string[] args)
      {
         Stack st = new Stack();
         st.Push('I');
         st.Push('P');
         st.Push('U');
         st.Push('H');
         st.Push('B');
         Console.WriteLine("CURRENT STACK:");
         foreach (char c in st)
         {
            Console.Write(c+ " ");
         }
         Console.WriteLine();
         Console.WriteLine("the peak item in stack st:{0}", st.Peek());
         Console.WriteLine("the stack after popes some element:");
         st.Pop();
         st.Pop();
         foreach (char c in st)
            Console.Write( c + " ");
         Console.WriteLine();
         Console.ReadKey();


      }
   }
}
```

Linked List

Linked list is very common data structure often used to store similar data in memory whose storage location is non contiguous

data link

Initialization

LinkedList<data type> <linked list name>= new LinkedList<data type>.();

Property	Description
Count	Gets the number of nodes actually contained in the LinkedList<T>.
First	Gets the first node of the LinkedList<T>.

Last	Gets the last node of the LinkedList<T>.

Method	Description
AddAfter(LinkedListNode<T>T) ,	Adds a new node containing the specified value after the specified existing node in theLinkedList<T>.
AddBefore(LinkedListNode<T>,T)	Adds a new node containing the specified value before the specified existing node in theLinkedList<T>.
AddFirst(T)	Adds a new node containing the specified value at the start of the LinkedList<T>.
AddLast(T)	Adds a new node containing the specified value at the end of the LinkedList<T>.
Clear()	Removes all nodes from the LinkedList<T>.
Find(T)	Finds the first node that contains the specified value.
Remove(T)	Removes the first occurrence of the specified value from the LinkedList<T>.
Remove(LinkedListNode<T>)	Removes the specified node from the LinkedList<T>.
RemoveFirst()	Removes the node at the start of the LinkedList<T>.
RemoveLast()	Removes the node at the end of the LinkedList<T>.

Example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LinkedlistApp
{
   class Program
   {
      static void Main(string[] args)
      {
         LinkedList<int> ld = new LinkedList<int>();
         ld.AddFirst(90);
         ld.AddAfter(ld.Find(90),80);
         ld.AddBefore(ld.Find(90),60);
         ld.AddLast(55);
         Console.WriteLine("the total number element in linked list is:{0}", ld.Count);
         Console.WriteLine("print the value of linked list:");
         foreach (int i in ld)
         {
            Console.WriteLine(i);
         }
         ld.Remove(55);
         ld.RemoveFirst();
```

```
            ld.RemoveLast();
            Console.WriteLine("print the value of linked list:");
            foreach (int i in ld)
            {
                Console.WriteLine(i);
            }
            Console.ReadKey();

        }
    }
}
```