

UNIT-III Object Oriented Programming Concepts.

C# Class and Objects

Class- *Class is a single unit which consists of data fields, methods, constructors, properties etc.*

A class is like a blueprint of a specific object that has certain attributes and features.

For example, a car should have some attributes such as four wheels, two or more doors, steering, a windshield, etc. It should also have some functionalities like start, stop, run, move, etc.

Now, any object that has these attributes and functionalities is a car. Here, the **car** is a **class** that defines some specific attributes and functionalities. *Each individual car is an object of the car class.* You can say that the car you are having is an object of the car class.

A class defines the kinds of data and the functionality their objects will have.

Define a Class

In C#, a class can be defined by using the class keyword. Let's define a class named 'Student'.

Example: Define a Class

```
class Student
{

}
```

A class can contain one or more constructors, fields, methods, properties, delegates, and events. They are called **class members**.

A class and its members can have access modifiers such as **public**, **private**, **protected**, and **internal**, to restrict access from other parts of the program.

Objects of a Class-

Object is an instance of a class.

You can create one or more objects of a class. *Each object can have different values of properties and field but methods and events behaves the same.*

In C#, an object of a class can be created using the `new` keyword and assign that object to a variable of a class type. For example, the following creates an object of the `Student` class and assign it to a variable of the `Student` type.

Example: Create an Object of a Class

```
Student mystudent = new Student();
```

You can now access **public** members of a class using the `object.MemberName` notation.

Example: Access Members of a Class

```
Student mystudent = new Student();  
mystudent.FirstName = "Steve";  
mystudent.LastName = "Jobs";  
mystudent.show();  
mystudent.GetFullName();
```

Let's discuss different members of a class.

Field

A class can have one or more fields.

It is a class-level variable that holds a value..

Example: Field

```
class Student  
{  
    public int id;  
  
}
```

Method

A Method is a simply block of statements.

A method can contain one or more statements to be executed as a single unit.

A method may or may not return a value.

A method can have one or more input parameters.

Syntax

```
[access-modifier] return-type MethodName(type parameterName1, type  
parameterName2,...)  
{  
    //Method Body...  
  
}
```

*The following code defines the **Sum** method that returns the sum of two numbers.*

Example: C# Method

```
public int Sum(int num1, int num2)
{
    int total = num1 + num2;
    return total;
}
```

*The following method doesn't return anything and doesn't have any parameters. The return type is **void**.*

Example: C# Method

```
public void Greet()
{
    Console.WriteLine("Hello World!");
}
```

Constructor

A constructor is a special type of method having same name as class name and it will be called automatically when we create an instance of a class.

Constructor can be used to initialize values of data fields to object

A constructor is defined by using an access modifier and class name

Syntax –

```
<access-modifier> <class-name>( optional args)
{
    // body
}
```

Example: Constructor

```
class Student
{
    public Student()
    {
        //constructor
    }
}
```

Features-

- *A constructor name must be the same as a class name.*
- *A constructor can be public, private, or protected.*
- *The constructor cannot return any value so cannot have a return type.*
- *A class can have multiple constructors with different parameters but can only have one parameter less (i.e. default) constructor.*
- *If no constructor is defined, the C# compiler would create it internally.*

Types of Constructors-

1. Default Constructor

A default constructor takes no arguments.

Each object of the class is initialized with the default values that are specified in the default constructor.

Therefore, it is not possible to initialize the different objects with different values.

Source Code: Program that demonstrates a default constructor in C#

```
using System;
namespace DefaultConstructorDemo
{
    class Sum
    {
        private int x;
        private int y;
        public Sum()
        {
            x = 5;
            y = 7;
        }
        public int getSum()
        {
            return x + y;
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            Sum s = new Sum();
```

```
        Console.WriteLine("Sum: " + s.getSum());  
    }  
}  
}
```

The output of the above program is as follows:

Sum: 12

2. Parameterized Constructor

A parameterized constructor can take one or more parameters.

Therefore, it can initialize different class objects to different values. This is an advantage over the default constructor.

Source Code: Program that demonstrates a parameterized constructor in C#

```
using System;  
namespace ParameterizedConstructorDemo  
{  
    class Sum  
    {  
        private int x;  
        private int y;  
        public Sum(int a, int b)  
        {  
            x = a;  
            y = b;  
        }  
        public int getSum()  
        {  
            return x + y;  
        }  
    }  
    class Test  
    {  
        static void Main(string[] args)  
        {  
            Sum s = new Sum(15,9);  
            Console.WriteLine("Sum: " + s.getSum());  
        }  
    }  
}
```

The output of the above program is as follows:
Sum: 24

3.Copy Constructor

A copy constructor initializes the object values by copying the data from another object.

It basically creates a new object which is a copy of the old object.

Source Code: Program that demonstrates a copy constructor in C#

```
using System;
namespace CopyConstructorDemo
{
    class Sum
    {
        private int x,y;
        public Sum(int a, int b)
        {
            x = a;
            y = b;
        }
        public Sum( Sum s )
        {
            x = s.x;
            y = s.y;
        }
        public int getSum()
        {
            return x + y;
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            Sum s1 = new Sum(2,9);
            Sum s2 = new Sum(s1); //invokes copy constructor
            Console.WriteLine("Sum of 2 and 9: " + s2.getSum());
        }
    }
}
```

The output of the above program is as follows:

Sum of 2 and 9: 11

4. Static Constructor

A static constructor initializes the static data members of a class. This is done implicitly when they are referenced for the first time i.e. when the class object is created.

A static constructor does not contain any parameters.

Source Code: Program that demonstrates a static constructor in C#

```
using System;
namespace StaticConstructorDemo
{
    public class SClass
    {
        public int x;
        public int y;
        public static int z; //static data field/member
        public SClass()
        {
            x = 5;
            y = 8;
        }
        static SClass() //Static Constructor
        {
            z = 10;
        }
        public void display()
        {
            Console.WriteLine("x = {0}", x);
            Console.WriteLine("y = {0}", y);
            Console.WriteLine("z = {0}", z);
        }
    }
    class Test
    {
        static void Main(string[] args)
        {
            SClass obj = new SClass();
        }
    }
}
```

```
        obj.display();  
    }  
}  
}
```

The output of the above program is as follows:

```
x = 5  
y = 8  
z = 10
```

5. Private Constructor-

If a constructor is created with private specifier is known as Private Constructor.

It is not possible for other classes to derive from this class having private constructor and also it's not possible to create an instance of this class.

Points To Remember:

- *It is the implementation of a singleton class pattern.*
- *Use private constructor when we have only static members.*
- *Using private constructor, prevents the creation of the instances of that class.*

Example 1: Private Constructor

```
using System;  
namespace Constructor {  
    class Car {  
        // private constructor  
        private Car () {  
            Console.WriteLine("Private Constructor");  
        }  
    }  
    class CarDrive {  
        static void Main(string[] args) {  
            // call private constructor  
            Car car1 = new Car();  
            Console.ReadLine();  
        }  
    }  
}
```


- In the above example, we have created a private constructor `Car()`. Since private members are not accessed outside of the class, when we try to create an object of `Car`

```
// inside CarDrive class  
Car car1 = new Car();  
we get an error
```

- error CS0122: '`Car.Car()`' is inaccessible due to its protection level

- *Note: If a constructor is private, we cannot create objects of the class. Hence, all fields and methods of the class should be declared static, so that they can be accessed using the class name.*

Static Keyword in C#

The static keyword is used to make a data item non-instantiable.

It can be used with classes, methods, variables, constructors, operators etc.

However, it cannot be used with destructors, indexers etc.

Some of the implementations of the static keyword are given as follows:

Static Class

A static class is non-instantiable i.e. a variable of the class cannot be created using the new keyword. So, the static class members have to be accessed using the class name itself.

A static class is defined using the keyword static. *It can only have static data members and static methods.* If this rule is not followed, there is a compile time error.

Source Code: Program that demonstrates a static class in C#

```
using System;  
namespace StaticClassDemo  
{  
    static class SClass  
    {  
        public static int staticVar = 5;  
        public static void staticMethod()  
        {  
            Console.WriteLine("Inside Static Method");  
        }  
    }  
}
```

```
}  
}  
class Test  
{  
    static void Main(string[] args)  
    {  
        Console.WriteLine("Value of static variable:"+SClass.staticVar);  
        SClass.staticMethod();  
    }  
}
```

The output of the above program is as follows:
Value of static variable: 5
Inside Static Method

C# Inheritance

Inheritance means create a new class from an existing class.

It is a key feature of Object-Oriented Programming (OOP).

The class from which a new class is created is known as the base class (parent or superclass). And, the new class is called derived class (child or subclass)

The derived class inherits the fields and methods of the base class. This helps with the code reusability in C#.

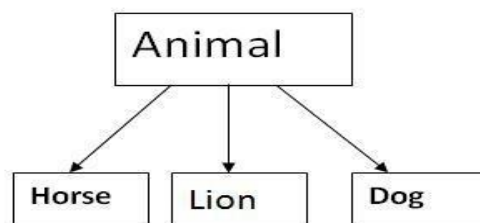
Inheritance is achieved in two different forms.

- 1) Classical Form
- 2) Containment Form.

1) Classical Form of Inheritance

The classical inheritance may be implemented in different combination. It is example of '**is-A**' relationship.

Classical Form



I

2) Containment Form of Inheritance

We can also define another form of inheritance relationship known as containership between class A and B.

It is example of '**Has-A**' relationship.

Example:

```

Class A
{
-----
-----
}

Class B
{
  A ob;
-----
}

```

In such case we say that object 'ob' of class A is contained in class B. This relationship between A and B is referred as 'has –a – relationship'. This outer class B which contain the inner class A is term the parent class and contain class |A is term as child class.

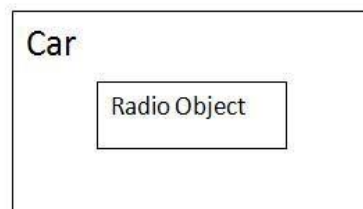
Example

1)Car has a radio

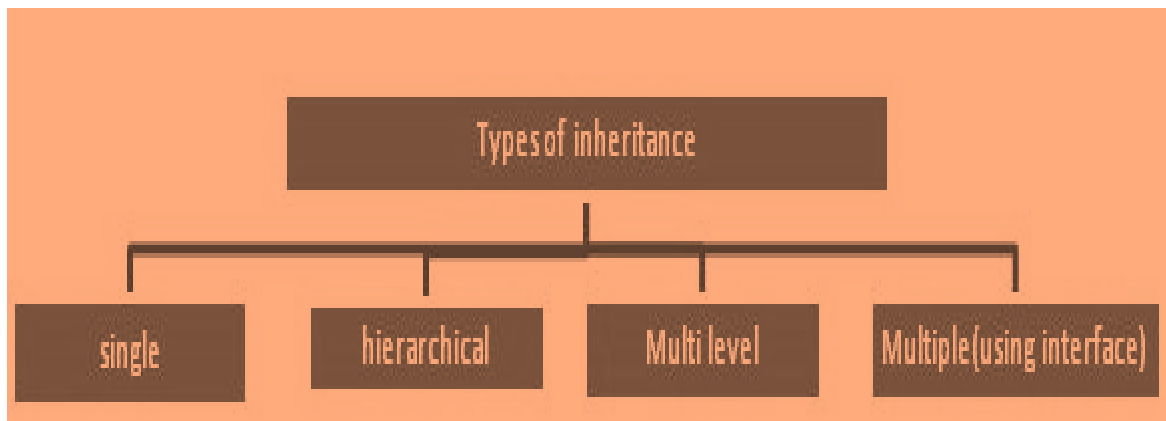
2)house has a store room

The has-a relationship is shown below.

Containment Form

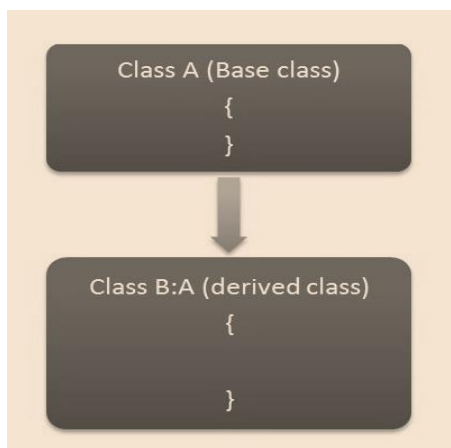
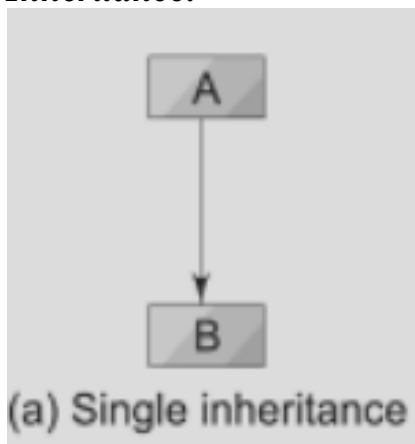


The following are the types of classical inheritance in C#.



1.Single inheritance

Derivation of a one child class using single parent class is known as Single Inheritance.



For example,

```
using System;
namespace InheritanceDemo
{
    class A
    {
        public void Method1()
        {
            Console.WriteLine("Method 1");
        }
        public void Method2()
```

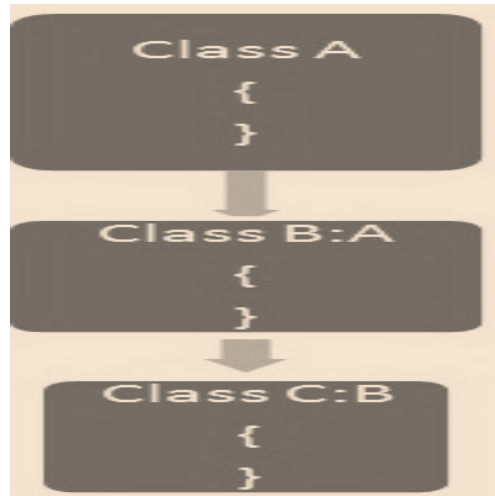
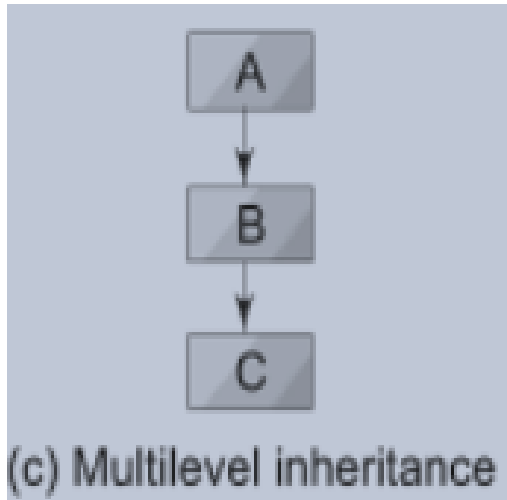
```
{  
    Console.WriteLine("Method 2");  
}  
}  
class B : A  
{  
    public void Method3()  
    {  
        Console.WriteLine("Method 3");  
    }  
    static void Main()  
    {  
        B obj = new B();  
        obj.Method1(); //parent class method  
        obj.Method2(); //parent class method  
        obj.Method3();  
        Console.ReadKey();  
    }  
}  
}
```

Output:

```
Method 1  
Method 2  
Method 3
```

2. Multilevel inheritance

When one class is derived from another derived class then this type of inheritance is called multilevel inheritance.

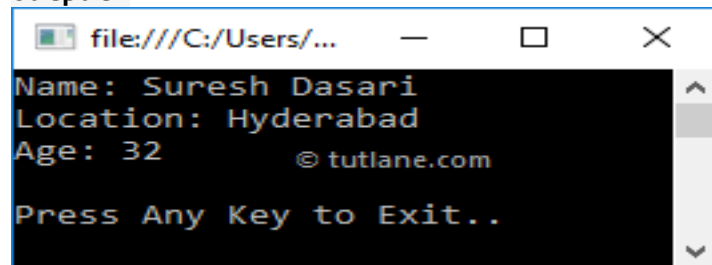


For example,

```
using System;
namespace MultilevelInheritDemo
{
    public class A    //Grand Parent Class
    {
        public string Name;
        public void GetName()
        {
            Console.WriteLine("Name:" + Name);
        }
    }
    public class B: A    //Intermediate Parent Class
    {
        public string Location;
        public void GetLocation()
        {
            Console.WriteLine("Location:" + Location);
        }
    }
    public class C: B    //Child Class
    {
        public int Age;
        public void GetAge()
        {
            Console.WriteLine("Age:" + Age);
        }
    }
}
```

```
}  
}  
class Program  
{  
    static void Main(string[] args)  
    {  
        C c = new C();  
        c.Name = "Suresh Dasari";  
        c.Location = "Hyderabad";  
        c.Age = 32;  
        c.GetName();  
        c.GetLocation();  
        c.GetAge();  
        Console.WriteLine("\nPress Any Key to Exit..");  
        Console.ReadLine();  
    }  
}
```

Output-

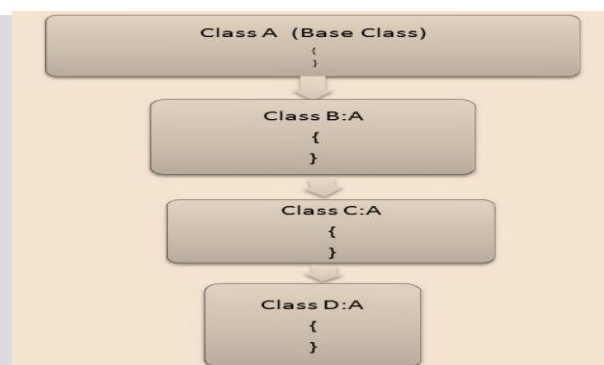
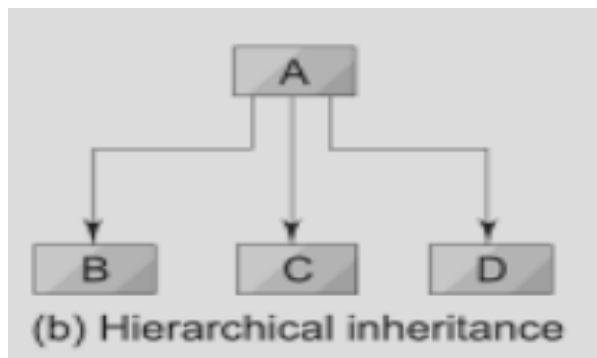


```
file:///C:/Users/...  
Name: Suresh Dasari  
Location: Hyderabad  
Age: 32  
© tutlane.com  
Press Any Key to Exit..
```

3. Hierarchical inheritance-

Derivation of multiple child classes from one base class is known as Hierarchical Inheritance.

This type of inheritance is used when there is a requirement of one class feature that is needed in multiple classes.



For example,

```
using System;
public class Father           // Base Class
{
    public string FatherName()
    {
        return "Ravi";
    }
}
public class ChildFirst : Father    // Derived Class 1
{
    public string ChildDName()
    {
        return "Rohan";
    }
}
public class ChildSecond : Father  // Derived Class 2
{
    public string ChildDName()
    {
        return "Nikhil";
    }
}
class Program
{
    static public void Main()
    {
        ChildFirst first = new ChildFirst(); // Object of Child 1
        Console.WriteLine("My name is " + first.ChildDName() + ". My father name is " + first.FatherName() + ".");
        ChildSecond second = new ChildSecond(); // Object of Child 2
        Console.WriteLine("My name is " + second.ChildDName() + ". My father name is " + second.FatherName() + ".");
    }
}
```

Output

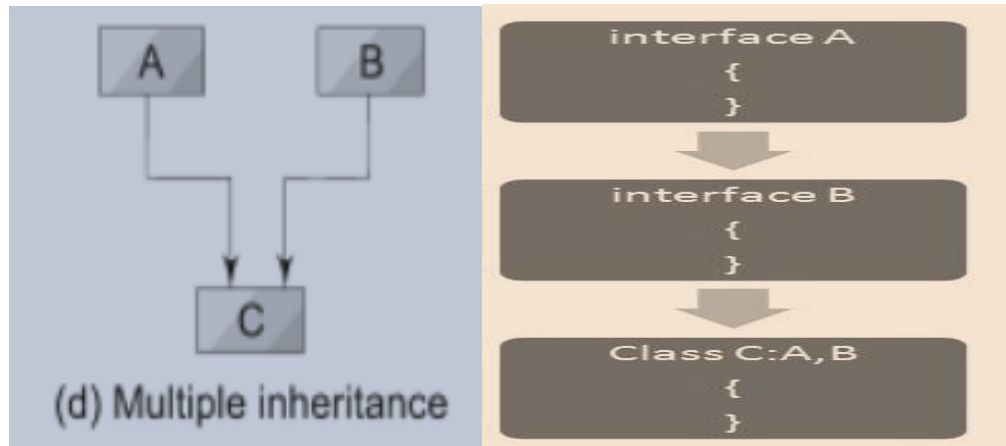
My name is Rohan. My father name is Ravi.

My name is Nikhil. My father name is Ravi.

4. Multiple inheritance using Interfaces

C# does not support multiple inheritances of classes.

To overcome this problem we can use interfaces.



Interface-

Classes in c# cannot have more than one super class. For Instance a definition Like ***Class A:B,C*** is not permitted in c#.

A large number of real life application require the use of multiple inheritance whereby we inherit methods and properties from several distinct classes. So c# provide an alternate approach known as interface to support the concept of multiple inheritance.

An interface in c# is a reference type.

It is basically a kind of class with some differences.

- 1) All members of an interface are implicitly **public and abstract**.
- 2) An interface **cannot** contain **constant field's, constructors, destructors**.
- 3) Since the method in an interface are **abstract** they **do not include implementation code**.
- 4) **Interface members cannot be static.**
- 5) An interface can inherit multiple Interfaces.

Defining an interface

An interface contain one or more methods.

It is the responsibility of the class that implement the interface to define the code for implementation of these members.

The Syntax for defining an interface

```
interface interface_name
{
    Member declaration
}
```

Here 'interface' is a keyword and interface_name is valid c# identifier, member declaration will contain a list of members without implementation code.

Implementing an interface

Interface are used as super classes whose methods are inherited by child classes.

It is therefore necessary to create a class that inherit given interface.

```
Class class_name : interface_name
{
    Body of class
}
```

For example,

Program 14.1**IMPLEMENTATION OF MULTIPLE INTERFACES**

```
using System;
interface Addition
{
    int Add ( );
}
interface Multiplication
{
    int Mul ( );
}
class Computation : Addition, Multiplication
{
    int x, y;
    public Computation (int x, int y )           //Constructor
    {
        this.x = x;
        this.y = y;
    }
    public int Add ( )                           //Implement Add ( )
    {
        return ( x + y );
    }
    public int Mul ( )                           //Implement Mul ( )
    {
        return ( x * y );
    }
}
class InterfaceTest1
{
    public static void Main( )
    {
        Computation com = new Computation (10,20);
        Addition add = (Addition ) com;          // casting
        Console.WriteLine ("Sum = " + add.Add ( ));
        Multiplication mul = (Multiplication) com; // casting
        Console.WriteLine("Product = " + mul.Mul ( ));
    }
}
```

Interface: Multiple Inheritance

Output of Program 14.1 will be:

Sum = 30
Product = 200

Explicit Interface-

One of the reasons that C# does not support multiple inheritance is the problem of name collision. However, this problem still persists in C# when it implements more than one interface. *Example:*

```
interface I1 { void Display ( ); }
interface I2 { void Display ( ); }
class C1 : I1, I2
{
    public void Display { }
```

Does **C1.Display()** implement **I1.Display()**, or **I2.Display()**? It is ambiguous and so the compiler reports an error. Such problems of name collision may occur when we implement interfaces from different sources.

C# supports a technique known as *explicit interface implementation*, which allows a method to specify explicitly the name of the interface it is implementing. Program 14.4 illustrates the use of explicit interface implementation.

Program 14.4 | EXPLICIT INTERFACE IMPLEMENTATION

```
using System;
interface I1
{
    void display ( );
}
interface I2
{
    void Display ( );
}
class C1 : I1, I2
{
    void I1.Display ( ) //no access modifier
    {
        Console.WriteLine("I1 Display");
    }
```

282 Programming in C#

```
    void I2.Display ( ) //no access modifier
    {
        Console.WriteLine("I2 Display");
    }
}
class InterfaceTest4
{
    public static void Main( )
    {
        C1 c = new C1( );

        I1 i1 = (I1) c;
        i1.display;

        I2 i2 = (I2)c;
        i2.Display ( );
    }
}
```

Output of Program 14.4 would be:

```
I1 Display
I2 Display
```

Note that while implementing the interface members, we have qualified explicitly with the interface name.

```
void I1. Display ( );
void I2. Display ( );
```

Also note that access modifiers are prohibited on explicit interface implementations.

Access Modifiers in C#

Access Modifiers are keywords that define the accessibility of a member, class or datatype in a program.

These are mainly used to restrict unwanted data manipulation by external programs or classes.

The accessibility level controls whether they can be used from other code in your assembly or other assemblies.

*An **assembly** is a .dll or .exe created by compiling one or more .cs files in a single compilation.*

Use the following access modifiers to specify the accessibility of a type or member when you declare it:

Access Specifier	Description
Public	It specifies that access is not restricted.
Protected	It specifies that access is limited to the containing class or in derived class.
Internal	It specifies that access is limited to the current assembly.
protected internal	It specifies that access is limited to the current assembly or types derived from the containing class.
Private	It specifies that access is limited to the containing type.

Summary table

Caller's location	public	protected internal	protected	internal	private protected	private
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

Constructor In Inheritance-

In C#, both the base class and the derived class can have their own constructor.

The constructor of a base class used to instantiate the objects of the base class and the constructor of the derived class used to instantiate the object of the derived class.

In inheritance, the derived class inherits all the members (fields, methods) of the base class, but derived class cannot inherit the constructor of the base class.

Instead of inheriting constructors by the derived class, it is only allowed to invoke the constructor of base class.

The Subclass Constructor uses the keyword **base** (*args*) to invoke constructor of superclass.

Example-

```

using System;
class Room          // base class
{
    public int length;
    public int breadth;
    public Room (int x , int y)    // base constructor
    {
        length      =    x;
        breadth     =    y;
    }
    public int Area ( )
    {
        return (length * breadth );
    }
}
class BedRoom : Room    //Inheriting Room
{
    int height;
    //subclass constructor
    public BedRoom (int x, int y, int z):base (x,y)
    {
        height = z;
    }
    public int Volume ( )
    {
        return (length * breadth * height);
    }
}
class InherTest
{
    public static void Main( )
    {
        BedRoom room1 = new BedRoom (14, 12, 10);
        int areal = room1.Area ( );    // superclass method
        int volume1 = room1.Volume ( );    // subclass method
        Console.WriteLine("Area1 = " + areal1);
        Console.WriteLine("Volume1 = " + volume1);
    }
}

```

The output of Program 13.2 would be:

```

Area1   = 168
Volume1 = 1680

```

The program defines a class **Room** and extends it to another class **BedRoom**. Note that the class **BedRoom** defines its own data members and methods. The subclass **BedRoom** now includes three instance variables, namely, **length**, **breadth** and **height** and two methods, **Area** and **Volume**.

The constructor in the derived class uses the **base** keyword to pass values that are required by the base constructor. The statement

```
BedRoom room1 = new BedRoom (14,12,10);
```

calls first the **BedRoom** constructor method which in turn calls the room constructor method by using the **base** keyword.

Finally, the object **room1** of the subclass **BedRoom** calls the method **Area** defined in the super class as well as the method **Volume** defined in the subclass itself.

Properties-

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

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

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,

1. get property accessor

A get accessor returns a property value

2. set property accessor.

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.

Types of properties-

Properties can be read-write, read-only, or write-only.

1. *The read-write property implements both, a get and a set accessor*
2. *A write-only property implements a set accessor, but no get accessor*
3. *A read-only property implements a get accessor, but no set accessor.*

Ex.1

```
using System;
class Number
{
    private int number;
    public int Anumber // property
    {
        get
        {
            return number;
        }
        set
        {
            number = value;
        }
    }
}
class PropertyTest
{
    public void static Main ( )
    {
        Number n = new Number ( );
        n.Anumber = 100;
        int m = n.Anumber;
        Console.WriteLine("Number = " + m);
    }
}
```

Ex.2

```
using System;
namespace PropertyDemo
{
    public class Employee
    {
        //Private Data Members
        private int EmpId;
        private string EmpName;
        public int eid
        {
            //The Set Accessor is used to set the EmpId private variable value
            set
            {
                _EmpId = value;
            }
        }
    }
}
```

```
//The Get Accessor is used to return the EmpId private variable value
get
{
return EmpId;
}
}
public string eName
{
//The Set Accessor is used to set the EmpName private variable value
set
{
EmpName = value;
}
//The Get Accessor is used to return the _EmpName private variable
value
get
{
return EmpName;
}
}
}
class Program
{
static void Main(string[] args)
{
Employee employee = new Employee();
//We cannot access the private data members
//So, using public properties (SET Accessor) we are setting
//the values of private data members
employee.eid = 101;
employee.eName = "Pranaya";
//Using public properties (Get Accessor) we are Getting
//the values of private data members
Console.WriteLine("Employee Details:");
Console.WriteLine("Employee id:" + employee.eid);
}
```

```
Console.WriteLine("Employee name:" + employee.eName);  
Console.ReadKey();  
}  
}  
}
```

Output:

```
Employee Details:  
Employee id:101  
Employee name:Pranaya
```

Indexers-

C# indexers are usually known as smart arrays.

A C# indexer is a class property that allows you to access a member variable of a class or struct using the features of an array.

In C#, indexers are created using this keyword. Indexers in C# are applicable on both classes and structs.

Instances of that class can be accessed using the [] array access operator.

Syntax-

```
1. <modifier> <return type> this [argument list]  
2. {  
3. get  
4. {  
5. // your get block code  
6. }  
7. set  
8. {  
9. // your set block code  
10. }  
11. }
```

Ex 1.

```
namespace Indexer_example1
{
    class Program
    {
        class IndexerClass
        {
            private string[] names = new string[5];
            public string this[int i]
            {
                get
                {
                    return names[i];
                }
                set
                {
                    names[i] = value;
                }
            }
        }
    }
    static void Main(string[] args)
    {
        IndexerClass Team = new IndexerClass();
        Team[0] = "Rocky";
        Team[1] = "Teena";
        Team[2] = "Ana";
        Team[3] = "Victoria";
        Team[4] = "Rani";

        for (int i = 0; i < 10; i++)
        {
            Console.WriteLine(Team[i]);
        }
        Console.ReadKey();
    }
}
```

Ex.2

```
namespace IndexerDEMO2
{
    class Program
    {
        int eno,salary;
        string ename;
        public object this[int i]
        {
            set
            {
                if (i == 1)
                    eno = (int)value;
                else if (i == 2)
                    salary = (int)value;
                else if (i == 3)
                    ename = (string)value;
            }
            get
            {
                if (i == 1) return eno;
                else if (i == 2) return salary;
                else if (i == 3) return ename;
                else return null;
            }
        }
    }
    static void Main(string[] args)
    {
        Program pob = new Program(); //First Object
        pob[1] = 101;
        pob[2] = 80000;
        pob[3] = "Saanvi";

        Console.WriteLine("Values In First Object ....");
        for (int i = 1; i <=3; i++)
            Console.WriteLine(pob[i]);
    }
}
```

```

Program pob2 = new Program();//Second Object
pob2[1] = 102;
pob2[2] = 75000;
pob2[3] = "Piyush";

Console.WriteLine("Values In Second Object ....");
for (int i = 1; i <= 3; i++)
    Console.WriteLine(pob2[i]);
Console.ReadKey();

    }
}
}

```

Output-

Values In First Object

101

80000

Saanvi

Values In Second Object

102

75000

Piyush

<i>Indexers</i>	<i>Properties</i>
<i>Indexers are created with this keyword.</i>	<i>Properties don't require this keyword.</i>
<i>Indexers are identified by signature.</i>	<i>Properties are identified by their names.</i>
<i>Indexers are accessed using indexes.</i>	<i>Properties are accessed by their names.</i>
<i>Indexer are instance member, so can't be static.</i>	<i>Properties can be static as well as instance members.</i>
<i>A get accessor of an indexer has the same formal parameter list as the indexer.</i>	<i>A get accessor of a property has no parameters.</i>

Polymorphism-

In c#, Polymorphism means providing an ability to take more than one form.

It's one of the main pillar concepts of object-oriented programming after [encapsulation](#) and [inheritance](#).

Generally, polymorphism is a combination of two words, **poly**, and another one is **morphs**. Here **poly** means “multiple” and **morphs** means “forms” so *polymorphism means many forms.*

In c#, we have two different kinds of polymorphisms available, those are

1. Operation Polymorphism / Compile Time Polymorphism

-Method Overloading

-Operator Overloading

2. Inclusion Polymorphism / Run Time Polymorphism

-Method Overriding

1. C# Compile Time Polymorphism-

*In c#, **Compile Time Polymorphism** means defining multiple [methods](#) with the same name but with different parameters. Using compile-time polymorphism, we can perform different tasks with the same method name by passing different parameters.*

In c#, the compile-time polymorphism can be achieved by using **method overloading**, and it is also called **early binding** or **static binding**.

Following is the code snippet of implementing a **method overloading** to achieve compile-time polymorphism in c#.

Ex. Program For Method Overloading.

```
using System;
namespace MethodOverloading
{
    class Program
    {
        public void Add(int a, int b)
        {
```

```
Console.WriteLine(a + b);  
}  
public void Add(float x, float y)  
{  
    Console.WriteLine(x + y);  
}  
public void Add(string s1, string s2)  
{  
    Console.WriteLine(s1 + " " + s2);  
}  
static void Main(string[] args)  
{  
    Program obj = new Program();  
    obj.Add(10, 20);  
    obj.Add(10.5f, 20.5f);  
    obj.Add("Pranaya", "Rout");  
    Console.ReadKey();  
}  
}  
}
```

Output:

```
30  
31  
Pranaya Rout
```

Operator Overloading

In C#, it is possible to make operators work with user-defined data types like classes. That means C# has *the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading*.

For example, we can overload the + operator in a class like String so that we can concatenate two strings by just using +.

Using operator overloading in C# we can specify more than one meaning for an operator in one scope. The purpose of operator overloading is to provide a special meaning of an operator for a user-defined data type.

The syntax for C# Operator Overloading:

To overload an operator in C#, we use a special operator function.

We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

```
public static return_type operator op (Type t)
{
    //Statements
}
Where Type must be a class or struct.
```

Here,

1. The return type is the return type of the function.
2. the operator is a keyword.
3. Op is the symbol of the operator that we want to overload. Like: +, <, -, ++, etc.
4. The type must be a class or struct. It can also have more parameters.
5. It should be a static function.

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.

Ex. Demonstration of Binary Operator Overloading....

```
namespace BinaryOperatorDemo
{
    class Program
    {
        int a, b;
        public Program(int x,int y)
        {
            a = x;
            b = y;
        }
        public void show()
        {
            Console.WriteLine("Value Of.." +a + "\t " +b);
        }
    }
}
```

```

public static Program operator +(Program p,Program q)
{
    Program r = new Program(0, 0);
    r.a = p.a + q.a;
    r.b = p.b + q.b;
    return r;
}
static void Main(string[] args)
{
    Program ob1 = new Program(10, 20);
    Program ob2 = new Program(30, 40);
    Program ob3 = new Program(0,0);
    ob3 = ob1 + ob2;
    ob1.show();
    ob2.show();
    ob3.show();
    Console.ReadLine();
}
}
}

```

Output-

Value Of..10 20

Value Of..30 40

Value Of..40 60

Ex.2: Unary - (minus) Operator Overloading.

```

namespace UnaryOperatorDemo
{
    class Program
    {
        int a, b, c;
        public Program(int x,int y,int z) //Constructor
        {
            a = x; b = y; c = z;
        }
        public static Program operator-(Program p)
        {
            p.a = -p.a;
            p.b = -p.b;
            p.c = -p.c;
            return p;
        }
    }
}

```

```
public void show()
{
    Console.WriteLine("a=" + a + "\t b=" + b + "\t c=" + c);
}
static void Main(string[] args)
{
    Program pob = new Program(10,-20,30);
    pob.show();

    pob=-pob;

    pob.show();
    Console.ReadLine();
}
}
```

Output-

a= 10 b= -20 c= 30

a= -10 b= 20 c= -30

Method Overriding in C#-

The process of re-implementing the superclass non-static, non-private, and non-sealed method in the subclass with the same signature is called Method Overriding.

The same signature means the name and the parameters (type, number, and order of the parameters) should be the same.

When do we need to override a method in C#?

If the Super Class or Parent Class method logic is not fulfilling the Sub Class or Child Class business requirements, then the Sub Class or Child Class needs to override the superclass method with the required business logic. Usually, in most real-time applications, the Parent Class methods are implemented with generic logic which is common for all the next-level sub-classes.

Program 13.4 | ILLUSTRATION OF METHOD OVERRIDING

```

using System;
class Super                                //base class
{
    protected int x ;
    public Super (int x)
    {
        this.x = x ;
    }

    public virtual void Display () // method defined with virtual
    {
        Console.WriteLine ("Super x = " + x);
    }
}

```

Inheritance and Polymorphism **257**

```

    }
}
class Sub : Super                        //derived class
{
    int y;
    public Sub (int x, int y) : base (x)
    {
        this.y = y;
    }
    public override void Display ( )      // method defined again
                                          //with override
    {
        Console.WriteLine("Super x = " + x) ;
        Console.WriteLine("Sub y = " + y);
    }
}
class OverrideTest
{
    public static void Main( )
    {
        Sub s1 = new Sub (100,200) ;
        s1.Display ( ) ;
    }
}

```

Output of Program 13.4 shows that the base class method has not been called.

```

Super x = 100
Sub y = 200

```

When overriding a method of a base class, we must be aware that we cannot change the accessibility level of the method. For some reason, if we want to call the base method we may do so using the **base** reference as shown below:

```
base.Display ( );
```

Note:

1. An override declaration may include the **abstract** modifier.
2. It is an error for an override declaration to include **new** or **static** or **virtual** modifier.
3. The overridden base method cannot be **static** or nonvirtual.
4. The overridden base method cannot be a **sealed** method.

Abstract Class-

Data abstraction is the process of hiding certain details and showing only essential information to the user which can be achieved with either **abstract classes** or **interfaces**.

A class is declared abstract to be an abstract class which includes abstract and non-abstract methods.

We cannot instantiate an abstract class.

We can use an abstract class as a base class and all derived classes must implement abstract definitions.

An abstract method must be implemented in all non-abstract classes using the **override** keyword.

C# Abstract Class Features

1. An abstract class can inherit from a class and one or more interfaces.
2. An abstract class can implement code with non-Abstract methods.
3. An Abstract class can have modifiers for methods, properties etc.
4. An Abstract class can have constants and fields.
5. An abstract class can implement a property.
6. An abstract class can have constructors or destructors.
7. An abstract class **cannot** be inherited from by structures.
8. An abstract class **cannot** support multiple inheritance.

EX.

```
using System;
public abstract class Vehicle {
    public abstract void display();
}

public class Bus : Vehicle {
    public override void display() {
        Console.WriteLine("Bus");
    }
}
```

```
public class Car : Vehicle {  
    public override void display() {  
        Console.WriteLine("Car");  
    }  
}  
  
public class Motorcycle : Vehicle {  
    public override void display() {  
        Console.WriteLine("Motorcycle");  
    }  
}  
  
public class MyClass {  
    public static void Main() {  
        Vehicle v;  
        v = new Bus();  
        v.display();  
        v = new Car();  
        v.display();  
        v = new Motorcycle();  
        v.display();  
    }  
}
```

Output

Bus

Car

Motorcycle

Sealed Classes-

Sealed classes are used to restrict the users from inheriting the class.

A class can be sealed by using the *sealed* keyword.

The keyword sealed tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class.

Features

1. A class, which restricts inheritance for security reason is declared sealed class.
2. Sealed class is the last class in the hierarchy.

3. Sealed class can be a derived class but *can't* be a base class.
4. A sealed class *cannot* also be an abstract class. Because abstract class has to provide functionality and here we are restricting it to inherit.

The following is the **syntax** of a sealed class :

```
sealed class class_name
{ // data members
  // methods
}
```

Ex. sealed class A

```
{
}
class B : A
{
  //if we derive B from A we will get //compiler error : 'B': cannot
  //derive from sealed type 'A'
}
```

Sealed method in C# :

During method overriding, if we don't want an overridden method to be further overridden by another class, we can declare it as a sealed method.

This is a method that is declared with the keyword **sealed** and is always used with combination of **override** keyword.

Derived classes will not be able to override this method as it is sealed for overriding.

```
Ex. class Parent
{
  public virtual void Show() { }
}
class Child : Parent
{
  public sealed override void Show() { }
}
class GrandChild : Child
{
  //GrandChild.Show(): cannot override inherited member 'Child.Show()'
  //because it is sealed
}
```