UNIT – 2

Class and Inheritance,Property, Indexer, Pointers, Delegates,Event,Collections

Programming with C#

Code : CS-23

Presented By : Dhruvita Savaliya

What is OOP ?

- OOPs is a concept of modern programming language that allows programmers to organize entities and objects.
- Four key concepts of OOPs are abstraction, encapsulation, inheritance, and polymorphism.
- **OOP Features**
- Object Oriented Programming (OOP) is a programming model where programs are organized around objects and data rather than action and logic.
- OOP allows decomposing a problem into many entities called objects and then building data and functions around these objects.
- A class is the core of any modern object-oriented programming language such as C#.
- In OOP languages, creating a class for representing data is mandatory.
- A class is a blueprint of an object that contains variables for storing data and functions to perform operations on the data.
- A class will not occupy any memory space; hence, it is only a logical representation of data.

Main Piller's of Object-Oriented Programming :

- **Abstraction**: Show only the necessary things.
- **Polymorphism**: More than one form: An object can behave differently at different levels.
- **Inheritance**: Parent and Child Class relationships.
- **Encapsulation**: Hides the Complexity.

- Most programming languages provide the following basic building blocks to build object-oriented applications:
- **Classes:** A Class define the structure using methods and properties/fields that resemble real-world entity.
- **Methods:** A method represents a particular behavior. It performs some action and might return information about an object, or update an object's data.
- **Properties:** Properties hold the data temporarily during the execution of an application.
- **Objects:** Objects are instances of the class that holds different data in properties/fields and can interact with other objects.
- **Interfaces:** An interface is a contract that defines the set of rules for a particular functionality. They are used effectively with classes using OOP principles like inheritance and polymorphism to make applications more flexible.

What is Method(UDF) ?

- A method is a group of statements that together perform a task. Every C# program has at least one class with a method named Main.
- To use a method, you need to −
  - Define the method
  - Call the method
- **Defining Methods in C# :**
  - When you define a method, you basically declare the elements of its structure.
- **Syntax :**

```
<Access Specifier> <Return Type> <Method Name>(Parameter List)
{
        Method Body
}
```

- **Access Specifier** − This determines the visibility of a variable or a method from another class. (public, protected, private).
- **Return type** − A method may return a value. The return type is the data type of the value the method returns. If the method is not returning any values, then the return type is **void**. (int, float, void, double, char, string, etc..)
- **Method name** − Method name is a unique identifier and it is case sensitive. It cannot be same as any other identifier declared in the class.
- **Parameter list** − Enclosed between parentheses, the parameters are used to pass and receive data from a method. The parameter list refers to the type, order, and number of the parameters of a method. Parameters are optional; that is, a method may contain no parameters.(int a,float b,etc…)
- **Method body** − This contains the set of instructions needed to complete the required activity. Console.WriteLine("Hello");

Class & Objects :

- ***Class and Object*** are the basic concepts of Object-Oriented Programming which revolve around the real-life entities.
- A class is a user-defined blueprint or prototype from which objects are created.
- In other words ,A class is a type, or blueprint which contains the data members and methods ,An Object is an instance.
- Basically, a class combines the fields and methods(member function which defines actions) into a single unit.
- In C#, classes support polymorphism, inheritance and also provide the concept of derived classes and base classes.

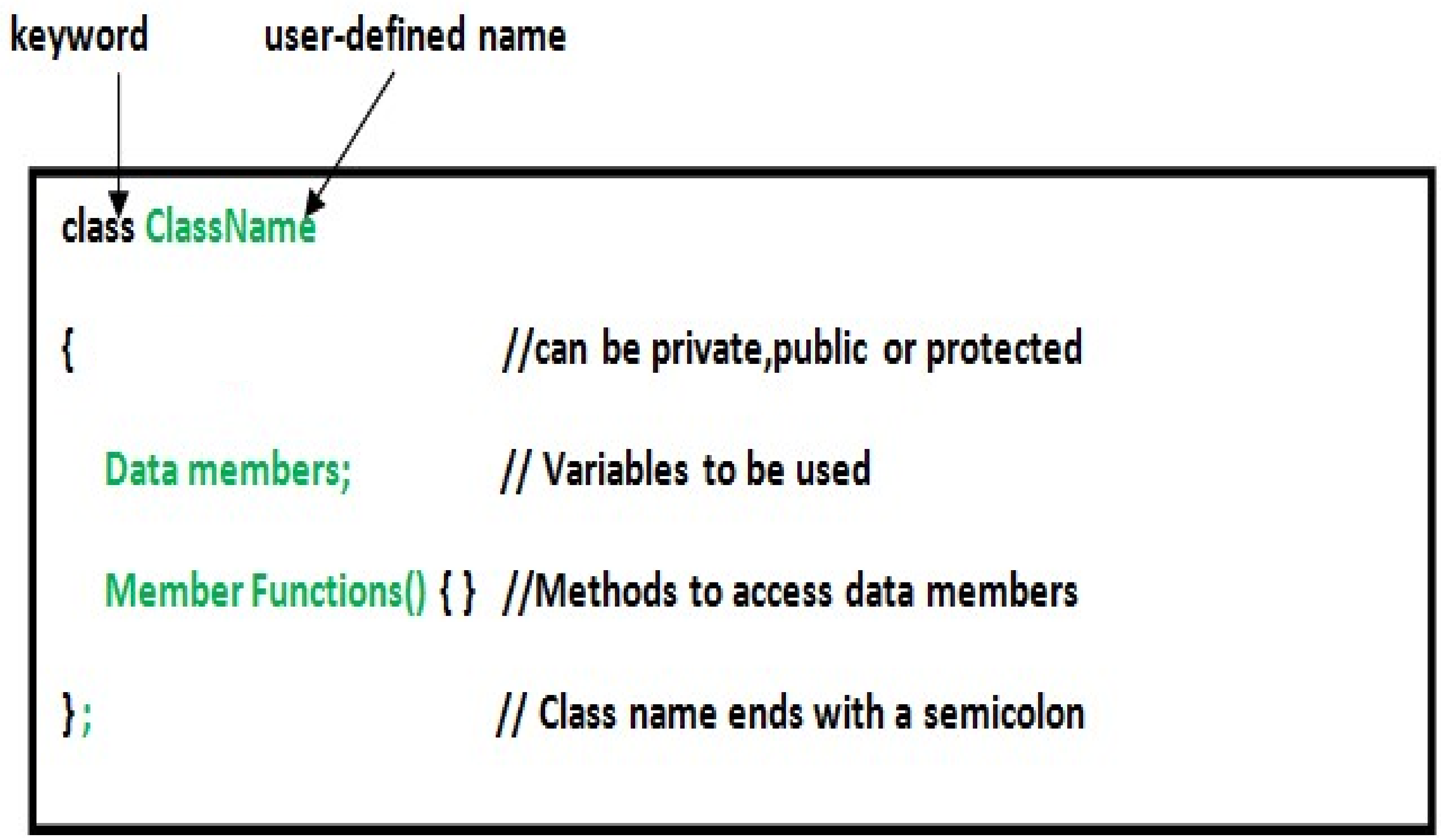
Declaration of class :

- Generally, a class declaration contains only a keyword **class**, followed by an **identifier(name)** of the class. But there are some optional attributes that can be used with class declaration according to the application requirement. In general, class declarations can include these components, in order:
- **Modifiers:** A class can be public or internal etc. By default modifier of the class is *internal*.
- **Keyword class:** A *class* keyword is used to declare the type class.
- **Class Identifier:** The variable of type class is provided. The identifier(or name of the class) should begin with an initial letter which should be capitalized by convention.
- **Base class or Super class:** The name of the class's parent (superclass), if any, preceded by the *: (colon)*. This is optional.
- **Interfaces:** A comma-separated list of interfaces implemented by the class, if any, preceded by the **: (colon)**. A class can implement more than one interface. This is optional.
- **Body:** The class body is surrounded by { } (curly braces).

Syntax :

```
<access specifier> class class_name
{
   // member variables
   <access specifier> <data type> variable1;
   <access specifier> <data type> variableN;

   // member methods
   <access specifier> <return type> method1(parameter_list)
   {
           // method body
   }
}
```



```
class ClassName

{                               //can be private,public or protected

   Data members;                // Variables to be used

   Member Functions() { }       //Methods to access data members

} ;                             // Class name ends with a semicolon
```

- Access specifiers specify the access rules for the members as well as the class itself.
- If not mentioned, then the default access specifier for a class type is **internal**.
- Default access for the members is **private**.
- Data type specifies the type of variable, and return type specifies the data type of the data the method returns, if any.
- To access the class members, you use the dot (.) operator.
- The dot operator links the name of an object with the name of a member.

Object :

- An **Object** is an instance of a Class.
- When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.
- Suppose, we have a class Color.
- Red , Pink , Purple , Green , White … are objects of color class.
- **Declaring Objects**
- When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.
- **Syntax :**

```
ClassName obj = new ClassName();
```

Example of Class & Object :

```
using System;
  public class Student
   {
      int id;//data member (also instance variable)
      string name;//data member(also instance variable)

   public static void Main(string[] args)
      {
         Student s1 = new Student();//creating an object of Student
         s1.id = 101;
         s1.name = "Sonoo Jaiswal";
         Console.WriteLine(s1.id);
         Console.WriteLine(s1.name);

      }
   }
```

new Operator :

- The new operator dynamically allocates memory for an object and returns a reference to it.
- This reference is then stored in a variable.
- Thus in c#, all class object must be dynamically allocated.
- Example :

Calculate c = new Calculate();

here, Calculate is class and c is object / instance of that class.

double[] points = new double[10];

here, points is name of array that has 10 size invoked by new operator.

Encapsulation :

- Encapsulation is the concept of wrapping data into a single unit.
- It collects data members and member functions into a single unit called class.
- The purpose of encapsulation is to prevent alteration of data from outside.
- This data can only be accessed by getter functions of the class.
- A fully encapsulated class has getter and setter functions that are used to read and write data.
- This class does not allow data access directly.
- **Must visit for access modifiers and encapsulation :**

https://www.tutorialspoint.com/csharp/csharp_encapsulation.htm

Polymorphism :

- The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word.
- The word **polymorphism** means having many forms. In object-oriented programming paradigm, polymorphism is often expressed as 'one interface, multiple functions'.
- Polymorphism can be static or dynamic. In **static polymorphism**, the response to a function is determined at the compile time. In **dynamic polymorphism**, it is decided at run-time.

Constructors :

- constructor is a special method which is invoked automatically at the time of object creation.
- It is used to initialize the data members of new object generally.
- A constructor also contains the collection of instructions that are executed at the time of Object creation.
- It is used to assign initial values to the **data members** of the same class.

Properties of Constructor :

- Constructor is a method that has same name as that class name.
- Constructor do not nave return type, so they can not have any return values.
- Access modifier can be used with constructor.
- It invoked when objects get created.
- It allocates the appropriate memory to objects.
- It initialize member variable of a class.
- We can declare more than one constructor in a class
- Constructor can be overloaded.

Syntax :

```
<Access_Modifier> class Class_name
{
         <Access_Modifier> Class_name()
        {
                 // This is the constructor method.
        }
        <Access_Modifier> Class_name(data_type  arg 1 ,
  data_type    arg 2, etc ..)
        {
                 // This is the constructor method.
        }
        // rest of the class members goes here.
}
```

Types of Constructor :

1. **Default Constructor**
2. **Parameterized Constructor**
3. **Copy Constructor**
4. **Private Constructor**
5. **Static Constructor**

1. Default Constructor :

- A constructor which has no argument is known as default constructor.
- It is invoked at the time of creating object.
- **Example :**

```
using System;
   public class Employee
    {
        public Employee()
        {
            Console.WriteLine("Default Constructor Invoked");
        }
    }
   class TestEmployee    {
       public static void Main(string[] args)
        {
            Employee e1 = new Employee();
            Employee e2 = new Employee();
        }
    }
```

Output :

```
Default Constructor Invoked
Default Constructor Invoked
```

2. Parameterized Constructor :

- A constructor which has parameters is called parameterized constructor.
- It is used to provide different values to distinct objects.

```
using System;
public class Employee     {
     public int id;
     public Employee(int i)   {
         id = i;
      }
     public void display()     {
         Console.WriteLine("Your Id is : "+id);
     }
}
class TestEmployee{
    public static void Main(string[] args)   {
         Employee  e1 = new Employee(101);
         Employee  e2 = new Employee(105);
         e1.display();
         e2.display();
     }
}
```

Output :

```
Your Id is : 101
Your Id is : 105
```

3. Copy Constructor :

- This constructor creates an object by copying variables from another object.
- Its main use is to initialize a new instance to the values of an existing instance.

```
using System;
namespace array
{
    class demo
    {
        int a, b;
        public demo(int aa,int bb)
        {
            a = aa; b = bb;
            Console.WriteLine("hello“+a+b);
        }
    public demo(demo d1)
    {
        a = d1.a; b = d1.b;
  Console.WriteLine("hellocopy"+a+b);
    }
 }
class Program
{
    static void Main(string[] args)
    {
         demo d = new demo(10,20);
        demo d1 = new demo(d);
    }
 }
}
```

4. 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 and also it's not possible to create an instance of this class.

```
Using System;
namespace  class1
{
   class Program
    {
    private Program()
    {
       Console.WriteLine("hello
program");
    }
    static void Main(string[] args)
    {
        Program p = new
Program();
    }
  }
}
```

5. Static Constructor :

- Static Constructor has to be invoked only once in the class and it has been invoked during the creation of the first reference to a static member in the class.
- A static constructor is initialized static fields or data of the class and to be executed only once.

```
Using System;
namespace  class1
{
   class Program
    {
          static int a;
        static Program()
        {
          a=10;
          Console.WriteLine("hello program");
        }
        static void Main(string[] args)
        {
           Program p = new Program();
        }
    }
}
```

Destructor :

- Destructors in C# are methods inside the class used to destroy instances of that class when they are no longer needed. The Destructor is called implicitly by the .NET Framework's Garbage collector and therefore programmer has no control as when to invoke the destructor. An instance variable or an object is eligible for destruction when it is no longer reachable.
- **Important Points:**
- A Destructor is unique to its class i.e. there cannot be more than one destructor in a class.
- A Destructor has no return type and has exactly the same name as the class name (Including the same case).
- It is distinguished apart from a constructor because of the *Tilde symbol (~)* prefixed to its name.
- A Destructor does not accept any parameters and modifiers.
- It cannot be defined in Structures. It is only used with classes.
- It cannot be overloaded or inherited.
- It is called when the program exits.
- Internally, Destructor called the Finalize method on the base class of object.

Syntax :

```
class class_name
{
        // Rest of the class
        // members and methods.
        // Destructor

        ~class_name()
        {
                // Your code
        }
}
```

Example :

```
using System;
namespace CsharpDestructor
{
    class Person
    {
            public Person()
            {
                Console.WriteLine("Constructor called.");
            }
            // destructor
            ~Person()
            {
                    Console.WriteLine("Destructor called.");
            }
            public static void Main(string [] args)
            {
                    //creates object of Person
                    Person p1 = new Person();
            }
    }
}
```

Inheritance :

- Inheritance is a fundamental concept in object-oriented programming that allows us to define a new class based on an existing class.
- The new class inherits the properties and methods of the existing class and can also add new properties and methods of its own.
- Inheritance promotes code reuse, simplifies code maintenance, and improves code organization.
- **Super Class:** The class whose features are inherited is known as super class(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as subclass(or a derived class, extended class, or child class).
- The subclass can add its own fields and methods in addition to the super class fields and methods.

Advantages of Inheritance:

1. **Code Reusability :** Inheritance allows us to reuse existing code by inheriting properties and methods from an existing class.
2. **Code Maintenance :** Inheritance makes code maintenance easier by allowing us to modify the base class and have the changes automatically reflected in the derived classes.
3. **Code Organization:** Inheritance improves code organization by grouping related classes together in a hierarchical structure.

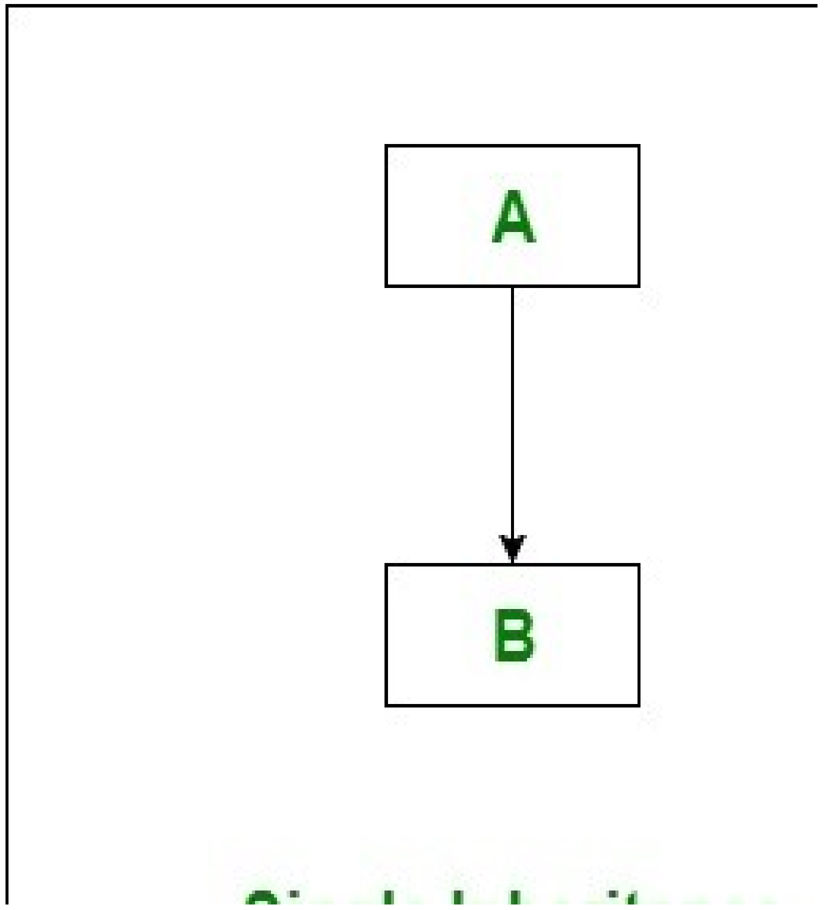
Disadvantages of Inheritance:

1. **Tight Coupling :** Inheritance creates a tight coupling between the base class and the derived class, which can make the code more difficult to maintain.
2. **Complexity :** Inheritance can increase the complexity of the code by introducing additional levels of abstraction.

- **Types of Inheritance in C#**

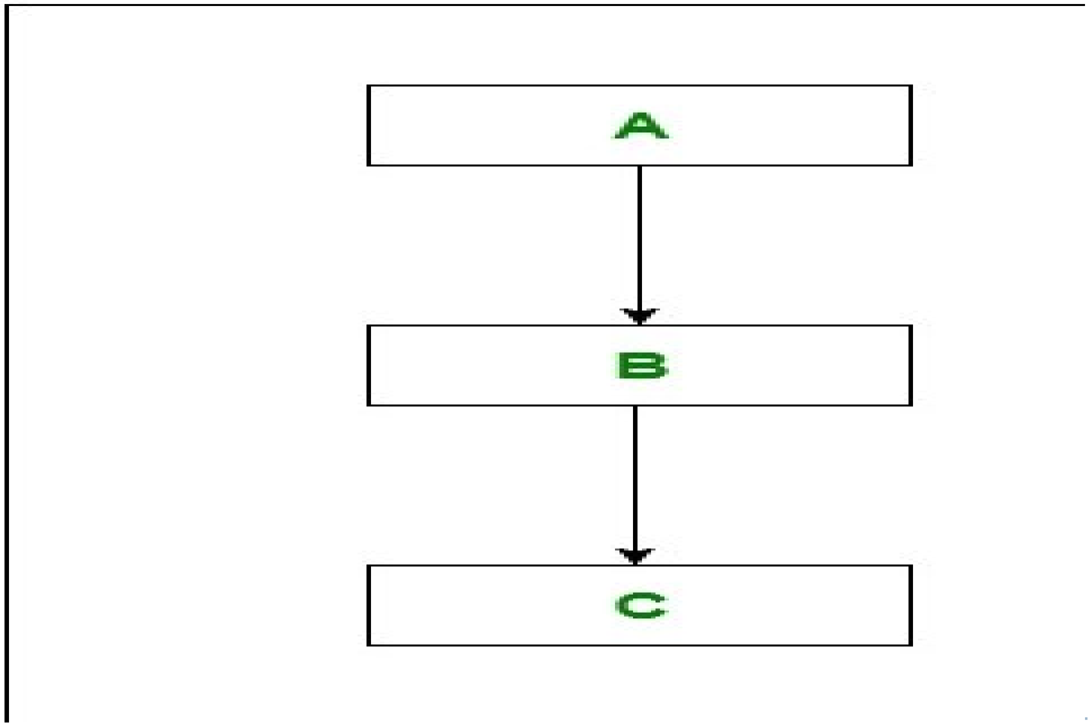
1. **Single Inheritance:** In single inheritance, subclasses inherit the features of one superclass.

   In image below, the class A serves as a base class for the derived class B.



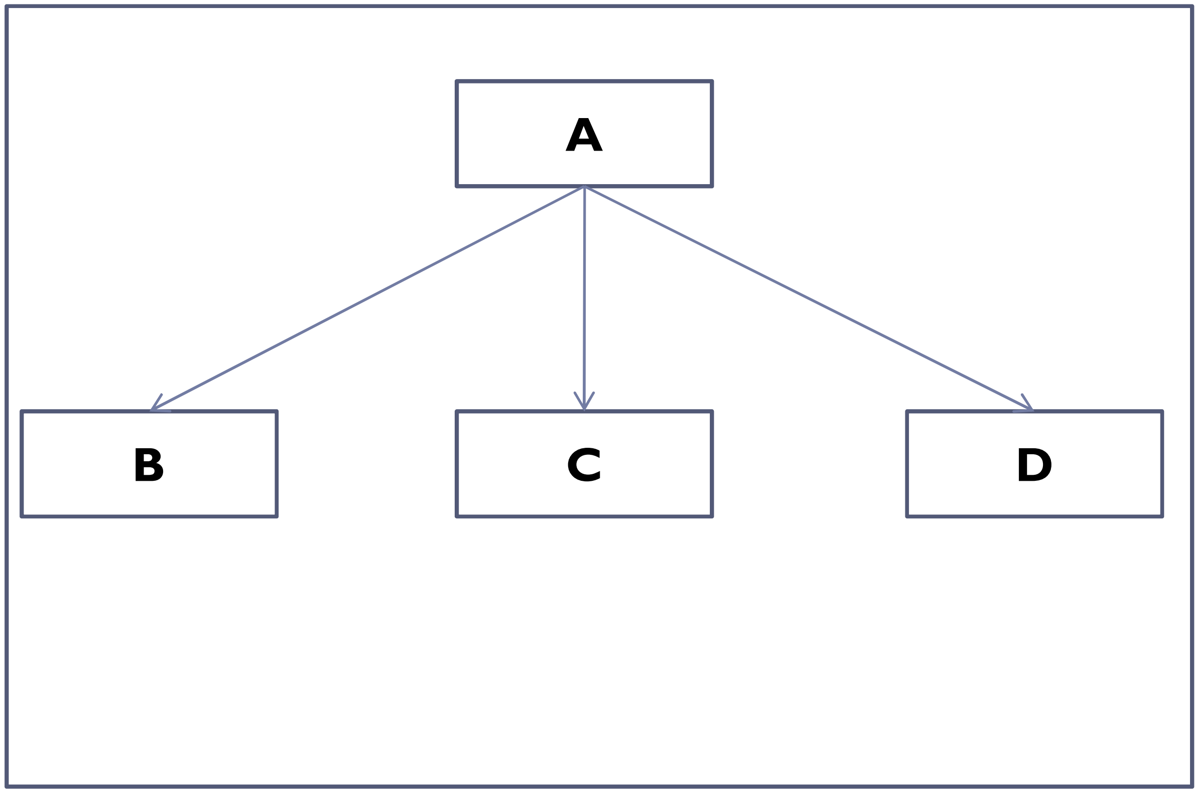
2. **Multilevel Inheritance:** In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to other class.

In below image, 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.



3. Hierarchical Inheritance: In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass.

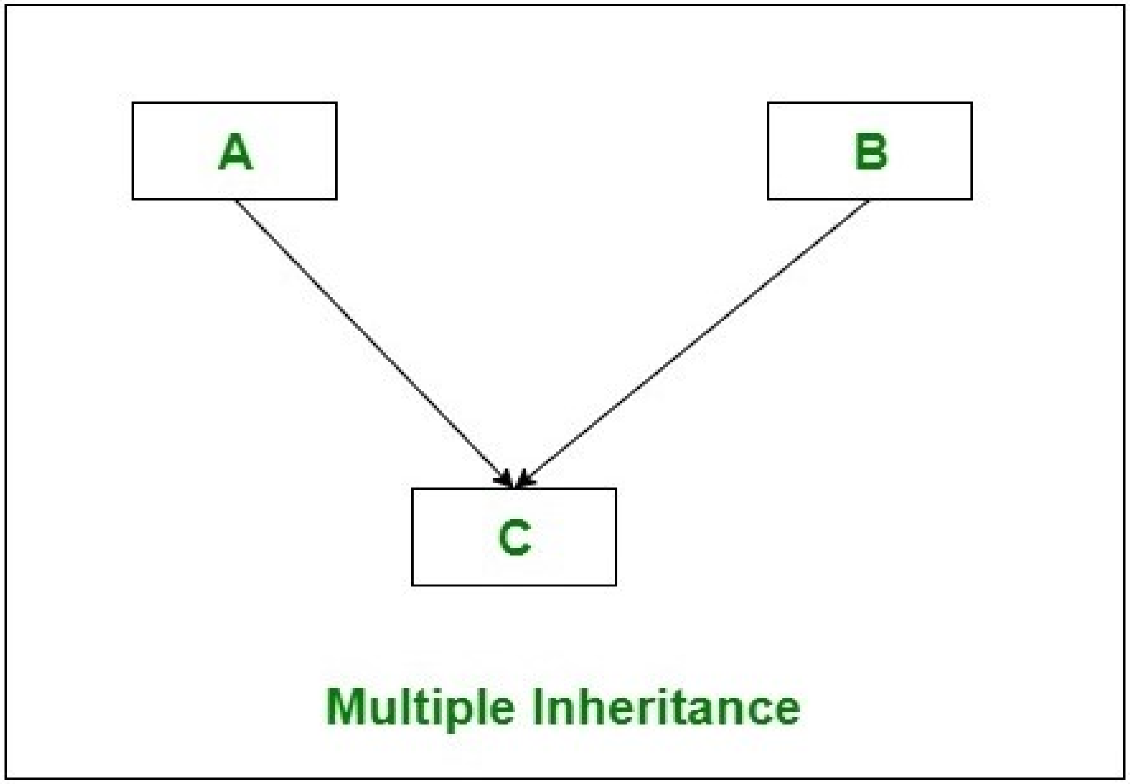
In below image, class A serves as a base class for the derived class B, C, and D.



4. Multiple Inheritance(Through Interfaces): In Multiple inheritance, one class can have more than one superclass and inherit features from all parent classes.

Please note that **C# does not support multiple inheritance** with classes.

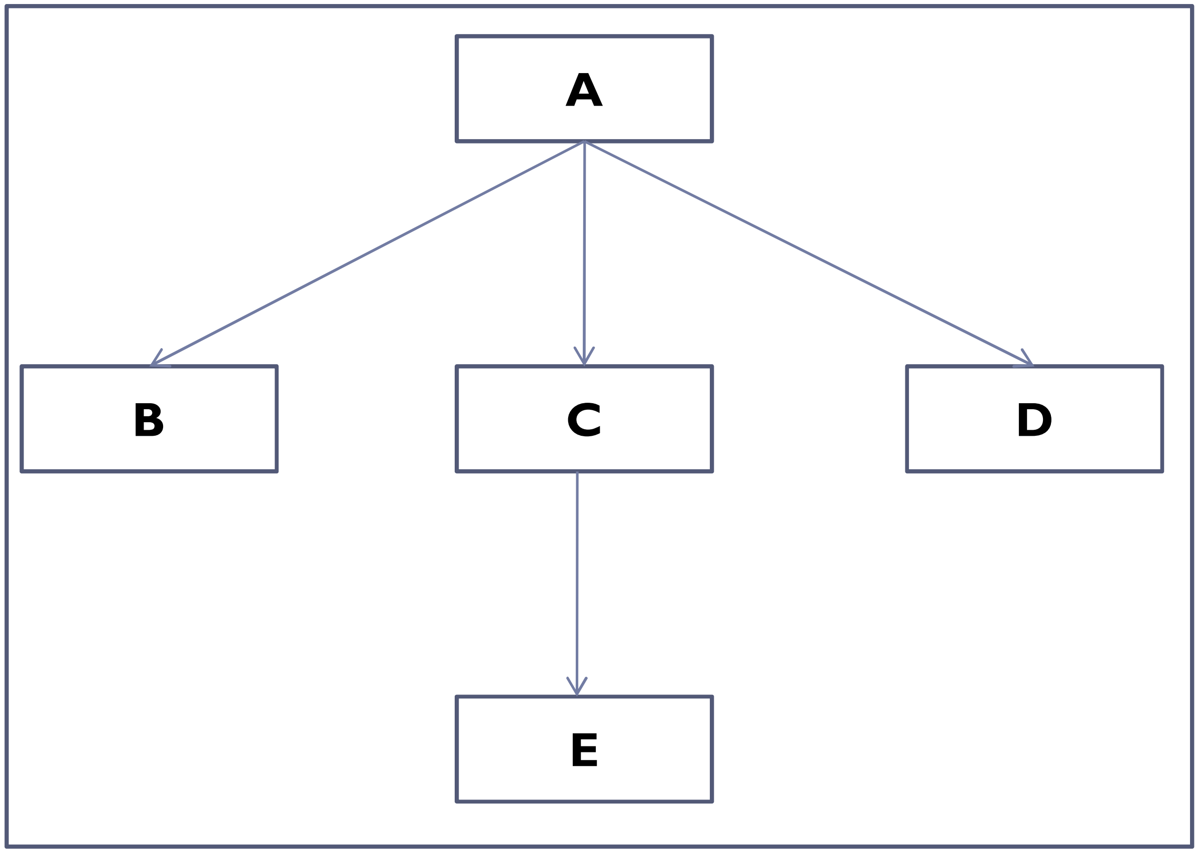
In C#, we can achieve multiple inheritance only through Interfaces.



5. Hybrid Inheritance(Through Interfaces): It is a mix of two or more of the above types of inheritance.

Since C# doesn't support multiple inheritance with classes, the hybrid inheritance is also not possible with classes.

In C#, we can achieve hybrid inheritance only through Interfaces.



- **How to use inheritance**
- The symbol used for inheritance is **:**.

Syntax:

```
class base_class
{
        // methods and fields . .
}
class derived_class : base_class
{
        // methods and fields . .
}
```

```
using System;
namespace inher
{
    class par       {
        public void abc()
        {
            Console.WriteLine("par ABC");
            a();
        }
        protected void a()
        {
            Console.WriteLine("par A");
        }
    }
    class child : par    {
        public void abc1()
        {
            a();
            Console.WriteLine("ch ABC1");
        }
    }
    class Program     {
        static void Main(string[] args)        {
            child c = new child();
            c.abc();//p
            c.abc1();//c
            Console.ReadLine();
        }
    }
}
```

Method overloading :

- ***Method Overloading*** is the common way of implementing polymorphism.
- It is the ability to redefine a function in more than one form.
- A user can implement function overloading by defining two or more functions in a class sharing the same name.
- C# can distinguish the methods with **different method signatures**. i.e.
- The methods can have the same name but with different parameters list (i.e. the number of the parameters, order of the parameters, and data types of the parameters) within the same class.
- **Rules :**
- All the method should have same name.
- All the method should be in same class.
- But method should have different parameters.
  - Different type of parameters
  - Different number of parameters
  - Different sequence of parameters

- **Example :**

```
using System;
class cls_1
    {
        public void display()
        {
            Console.WriteLine("First Method");
        }
              public void display(int a)
        {
            Console.WriteLine("Second 
    Method"+a);
        }
    }
class Program
{
    static void Main(string[] args)
        {
            cls_1 obj1 = new cls_1();
                obj1.display();
                obj1.display(10);
            Console.ReadLine();
        }
    }
```

Method Override :

- 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.
- **Rules :**
- All the method should have same name.
- All the method should be in different class.
- But method should have same parameters.
  - Same type of parameters
  - Same number of parameters
  - Same sequence of parameters
- There should be inheritance between classes.
- Method overriding is one of the ways by which C# achieve *Run Time Polymorphism(Dynamic Polymorphism)*.

- **Example :**

```
using System;
class cls_1
{
    public virtual void display()
    {
            Console.WriteLine("Parent
   class Method");
    }
}
class cls_2 : cls_1
{
    public override void display()
    {
            Console.WriteLine("Child
   class Method");
    }
}
class Program
{
    static void Main(string[] args)
    {
        cls_2 obj1 = new cls_2();
        obj1.display();
       //call child class method
        Console.ReadLine();
    }
}
```

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. (a+b)
- 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
  }
  ```

- **Example :**

  public static Calculator **operator** -(Calculator c1)

  ```
  {
          ..........
          return c1;
  }
  ```

```
class op_overloading
  {
    private int a, b;
    public op_overloading(int x,int y)
    {
      a = x;
      b = y;
    }
    public static op_overloading operator +(op_overloading
   o1,op_overloading o2)
    {
      op_overloading o3 = new op_overloading(0,0);

      o3.a = o1.a + o2.a;
      o3.b = o1.b + o2.b;
      return o3;
    }
    public void display()
    {
      Console.WriteLine("value of A is " + a);
      Console.WriteLine("value of B is " + b);
      Console.WriteLine();
    }
  }
class Program
{
  static void Main(string[] args)
  {
    op_overloading obj1 = new op_overloading(10,30);
    op_overloading obj2 = new op_overloading(33,23);

    op_overloading obj_res = new op_overloading(0,0);
    obj_res = obj1 + obj2;
    Console.WriteLine("value of A & B using OBJ1");
    obj1.display();
    Console.WriteLine("value of A & B using OBJ2");
    obj2.display();
    Console.WriteLine("value of A & B using OBJ_RES");
    obj_res.display();
    Console.ReadLine();
  }
}
```

Ref and Out Parameter’s / Reference Parameter’s :

- Ref and out keywords in C# are used to pass arguments within a method or function.
- Both indicate that an argument/parameter is passed by reference.
- By default parameters are passed to a method by value.
- By using these keywords (ref and out) we can pass a **parameter by reference.**

ref Keyword :

- The ref keyword passes arguments by reference. It means any changes made to this argument in the method will be reflected in that variable when control returns to the calling method.

out Keyword :

- out keyword is used to pass arguments to method as a reference type and is primary used when a method has to return multiple values.

Example :

```
using System;
class cls_1
{
    public void display(ref int a,out int b)
    {
                a=56;
                b=88;
    }
}
class Program
{
    static void Main(string[] args)
    {
                int x=5,y;
        cls_1 obj = new cls_1();
                Console.WriteLine("Before Function Calling X "+x);
        obj.display(ref x,out y);
                Console.WriteLine("After Function Calling X {0}\n Y{1}",x,y);
        Console.ReadLine();
    }
}
```

Output :

```
Before Function Calling X 5
After Function Calling X 56
 Y88
```

Sealed class :

- Sealed classes are used to restrict the users from inheriting the class.
- A class can be sealed by using the ***sealed*** keyword.
- The keyword tells the compiler that the class is sealed, and therefore, cannot be extended.
- No class can be derived from a sealed class.
- It means it will not work like parent Class.
- **Syntax :**

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

- **Some points to remember of sealed class :**
  - A class, which restricts inheritance for security reason is declared sealed class.
  - Sealed class is the last class in the hierarchy.
  - Sealed class can be a derived class but can't be a base class.
  - A sealed class cannot also be an abstract class. Because abstract class has to provide
- **Sealed Methods :**
- Sealed method is used to define the overriding level of a virtual method.
- Sealed keyword is always used with override keyword.
- **Syntax :**

Access_modifier **sealed** method_name(…. para List…)

```
{
        // data members
        // methods . . .
}
```

Example of sealed **class** :

```
public class Animal
    {
        public  void run() {
    Console.WriteLine("running...");
    }
    }
    public sealed class Dog : Animal
    {
        public void eat() {
    Console.WriteLine("eating
    bread..."); }
    }
    //now not able to generate a
    child of god class
```

```
/* public class BabyDog : Dog {}
*/
class Program
{
    static void Main(string[] args)
    {
        Dog d = new Dog();
        d.eat();
        d.run();
        Console.ReadLine();
    }
}
```

Output :

```
eating bread...
running...
```

Example of sealed **Method** :

```
public class Animal
   {
      public virtual void eat() {
   Console.WriteLine("eating..."); }
      public virtual void run() {
   Console.WriteLine("running..."); }


   }
   public class Dog : Animal
   {
      public override void eat() {
   Console.WriteLine("eating bread..."); }
      public sealed override void run()
      {
          Console.WriteLine("running very
   fast...");
      }
   }
   public class BabyDog : Dog
{
   public override void eat() {
 Console.WriteLine("eating biscuits..."); }
   //public override void run() {
 Console.WriteLine("running slowly..."); }
}
class Program
{
   static void Main(string[] args)
   {
      BabyDog d = new BabyDog();
      d.eat();
      d.run();
      Console.ReadLine();
   }
}
```

Output :

```
running very fast...
```

Abstract class :

- Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
- Abstraction can be achieved with either **abstract classes** or **interfaces**.
- It means that only the required information is visible to the user and the rest of the information is hidden.
- Abstraction can be implemented using abstract classes in C#.
- Abstract classes are base classes with partial implementation.
- These classes contain abstract methods that are inherited by other classes that provide more functionality.
- The abstract keyword is used for classes and methods:
- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
  Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited from).

- **Some of the salient points about abstract classes are as follows:**
- The abstract class is created using the keyword abstract and some of the methods of the abstract class also contain the keyword abstract.
- No object can be created of the abstract class i.e.it cannot be instantiated.
- The abstract methods in the abstract class are implemented actually only in the derived classes.
- If all the methods in the abstract class contain the keyword abstract, then that class is known as pure Abstract class.
- **Advantages of Abstraction**
- It reduces the complexity of viewing things.
- Avoids code duplication and increases reusability.
- Helps to increase the security of an application or program as only important details are provided to the user.

```
abstract class abs_demo
{
    public  abstract void color();
    public void show()
    {
            Console.WriteLine(“hello abstraction");
    }
}
class child : abs_demo
{
    public override void color()
    {
            Console.WriteLine(“red”);
    }
}
```

```
class program{
public static void Main()
{
    child obj=new child();
    obj.color();
    obj.show();
}
}
```

abs_demo myObj = abs_demo();

// Will generate an error (Cannot create an instance of the abstract class or interface ‘abs_demo ')

Remember from the Inheritance chapter that we use the **:** symbol to inherit from a class, and that we use the **override** keyword to override the base class method.

- **Encapsulation** is data hiding(information hiding) while Abstraction is detail hiding(implementation hiding).
- While encapsulation groups together data and methods that act upon the data, data abstraction deal with exposing to the user and hiding the details of implementation.
- **Why Encapsulation?**
- Better control of class members (reduce the possibility of yourself (or others) to mess up the code).
- Fields can be made **read-only** (if you only use the get method), or **write-only** (if you only use the set method).
- **Flexible :** the programmer can change one part of the code without affecting other parts.
- Increased security of data.

Interface :

- Like a class, ***Interface*** can have methods, properties, events, and indexers as its members.
- But interfaces will contain only the declaration of the members.
- The implementation of the interface’s members will be given by class who implements the interface implicitly or explicitly.
  - Interfaces specify what a class must do and not how.
  - Interfaces can’t have private members.
  - By default all the members of Interface are **public and abstract**.
  - The interface will always defined with the help of keyword ‘*interface*‘.
  - Interface cannot contain fields because they represent a particular implementation of data.
  - *Multiple inheritance* is possible with the help of Interfaces but not with classes.

- **Syntax :**

```
interface <interface_name >
{
        // declare Events
        // declare indexers
        // declare methods
        // declare properties
}
```

- **Syntax for Implementing Interface:**

```
        class class_name : interface_name
```

- To declare an interface, use *interface* keyword. It is used to provide total abstraction. That means all the members in the interface are declared with the empty body and are public and abstract by default. A class that implements interface must implement all the methods declared in the interface.

Example of Interface :

```
using System;
interface intr1
{
    void abc();//public abstract
}
interface intr2
{
    void xyz();//public abstract
}
//multiple interface in single class
class demo :intr1,intr2
{
    public void fun()
    {
            Console.WriteLine("FUN");
    }
    public void abc()//from intr1
    {
            Console.WriteLine("ABC");
    }
    public void xyz()//from intr2
    {
            Console.WriteLine("XYZ");
    }
}
class main_cls
{
    public static void Main()
    {
            demo d=new demo();
            d.fun();
            d.abc();
            d.xyz();
    }
}
```

Property :

- Properties are the special type of class members that provides a flexible mechanism to read, write, or compute the value of a private field.
- Properties can be used as if they are public data members, but they are actually special methods called ***accessors***.
- This enables data to be accessed easily and helps to promote the flexibility and safety of methods. Encapsulation and hiding of information can also be achieved using properties.
- It uses pre-defined methods which are "get" and "set" methods which help to access and modify the properties.
- **Accessors:** The block of "set" and "get" is known as "Accessors".
- It is very essential to restrict the accessibility of property.
- There are two type of accessors i.e. **get accessors** and **set accessors**.
- There are different types of properties based on the "get" and "set" accessors:

- **Read and Write Properties:** When property contains both get and set methods.
- **Read-Only Properties:** When property contains only get method.
- **Write Only Properties:** When property contains only set method.
- **Auto Implemented Properties:** When there is no additional logic in the property accessors and it introduce in C# 3.0.
- **Syntax :**

```
<access_modifier> <return_type> <property_name>
{
        get
        {
                    // body
        }
        set
        {
                   // body
        }
}
```

- **Get Accessor:** It specifies that the value of a field can access publicly. It returns a single value and it specifies the *read-only property*.
- **Example:**

```
class demo
{
        private int roll_no;
        public int Roll
         {
              get {
                        return roll_no;
                   }
         }
}
```

- **Set Accessor:** It will specify the assignment of a value to a private field in a property. It returns a single value and it specifies the *write-only property*.
- **Example:**

```
class demo
{
        private int roll_no;
        public int Roll
         {
                 set {
                      roll_no=value;
                   }
         }
}
```

- **// read and write property**

```
using System;
public class Student {
   private string name;
   public string nm
   {
         get
         {
                  return name;
         }
         set
         {
                  name = value;
         }
   }
}
class TestStudent
{
   public static void Main(string[ ]
   args)
   {
         Student s = new
   Student();
         s.nm = “Hello";
   Console.WriteLine("Name: " +
   s.nm);
   }
}
```

- **Automatic Properties :**
- C# also provides a way to use short-hand / automatic properties, where you do not have to define the field for the property, and you only have to write get; and set; inside the property.
- **Example :**

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

Indexer :

- An indexer allows an instance/object of a class or struct to be indexed as an array.
- If the user will define an indexer for a class, then the class will behave like a virtual array.
- Array access operator i.e. (**[ ]**) is used to access the instance of the class which uses an indexer.
- Indexer is created with **this** keyword .
- A user can retrieve or set the indexed value without pointing an instance or a type member.
- Indexers are almost similar to the **Properties**.
- *The main difference between Indexers and Properties* is that the accessors of the Indexers will take parameters.

- **Syntax:**

```
[access_modifier] [return_type]  this[argument_list]
{
        set
        {
                // set block code
        }
        get
        {
                // get block code
        }

}
```

- **access_modifier:** It can be public, private, protected or internal.
- **return_type:** It can be any valid C# type.
- **this:** It is the keyword which points to the object of the current class.
- **argument_list:** This specifies the parameter list of the indexer.
- **get{ } and set { }:** These are the accessors.

Example :

```
using System;

class Indexer
{
    private string[ ] val = new string[3];

    public string this[int index]
    {
            get
            {
                        return val[index];
            }
            set
            {
                        val[index] = value;
            }
    }
}
```

```
class main {
    public static void Main() {
            Indexer ic = new Indexer();
            ic[0] = "C";
            ic[1] = "CPP";
            ic[2] = "CSHARP";
            Console.Write("Printing values
    stored in objects used as arrays\n");
            // printing values
            Console.WriteLine("First value =
    {0}", ic[0]);
            Console.WriteLine("Second value
    = {0}", ic[1]);
            Console.WriteLine("Third value =
    {0}", ic[2]);
    }
}
```

Unsafe code :

- Unsafe code in C# is the part of the program that runs outside the control of the Common Language Runtime (CLR) of the .NET frameworks.
- The CLR is responsible for all of the background tasks that the programmer doesn't have to worry about like memory allocation and release, managing stack etc.
- Using the keyword "unsafe" means telling the compiler that the management of this code will be done by the programmer.
- Making a code content unsafe introduces stability and security risks as there are no bound checks in cases of arrays, memory related errors can occur which might remain unchecked etc.
- A programmer can make the following sub-programs as unsafe:Code blocks
- Methods
- Types
- Class
- Struct

- **Need to use the unsafe code?**
  - When the program needs to implement pointers.
  - If native methods are used.
- **Syntax:**

  unsafe Context_declaration
- **Example :**

```
using System;

namespace ConsoleApplication
{

class Program {
   static void Main(string[] args)
   {
          unsafe
          {
                    int x = 10;
                    int* ptr;
                    ptr = &x;

          Console.WriteLine("Inside
    the unsafe code block");

          Console.WriteLine("The
    value of x is " + *ptr);
        } // end unsafe block

    Console.WriteLine("\nOutside
    the unsafe code block");
      } // end main
    }
}
```

- to compile a program named prog1.cs containing unsafe code, from command line, give the command −

```
csc /unsafe prog1.cs
```

- If you are using Visual Studio IDE then you need to enable use of unsafe code in the project properties.
- To do this −
- Open **project properties** by right clicking the properties node in the Solution Explorer.
- Click on the **Build** tab.
- Select the option "**Allow unsafe code**".

Delegate :

- A delegate is an object which refers to a method or you can say it is a reference type variable that can hold a reference to the methods.
- Delegates in C# are similar to the function pointer in C/C++.
- It provides a way which tells which method is to be called when an event is triggered.
- For example, if you click on a *Button* on a form (Windows Form application), the program would call a specific method.
- In simple words, it is a type that represents references to methods with a particular parameter list and return type and then calls the method in a program for execution when it is needed.

- **Important points of Delegates**
- Provides a good way to encapsulate the methods.
- Delegates are the library class in System namespace.
- These are the type-safe pointer of any method.
- Delegates are mainly used in implementing the call-back methods and events.
- Delegates can be chained together as two or more methods can be called on a single event.
- It doesn’t care about the class of the object that it references.
- Delegates can also be used in “anonymous methods” invocation.

- There are two types of delegates in C#, singlecast delegates and multiplecast delegates.
- **Singlecast delegate:** Singlecast delegate points to a single method at a time. The delegate is assigned to a single method at a time. They are derived from System.Delegate class.
- **Multicast Delegate:** When a delegate is wrapped with more than one method, that is known as a multicast delegate.

1. Single cast delegate :

- Singlecast delegate points to a single method at a time. The delegate is assigned to a single method at a time. They are derived from System.Delegate class.
- **Declaration of Delegates :**
  Delegate type can be declared using the **delegate** keyword. Once a delegate is declared, delegate instance will refer and call those methods whose return type and parameter-list matches with the delegate declaration.
- **Syntax:**

  [modifier] delegate [return_type] [delegate_name] ([parameter_list]);

- **modifier:** It is the required modifier which defines the access of delegate and it is optional to use.
  delegate: It is the keyword which is used to define the delegate.
  return_type: It is the type of value returned by the methods which the delegate will be going to call. It can be void. A method must have the same return type as the delegate.
  delegate_name: It is the user-defined name or identifier for the delegate.
  parameter_list: This contains the parameters which are required by the method when called through the delegate.

- **Note:** A delegate will call only a method which agrees with its signature and return type.
- A method can be a static method associated with a class or can be an instance method associated with an object, it doesn't matter.

Example of Single cast delegate :

```
using System;
public delegate void del_nm(int
   a,int b);
   class del_class    {
        public void sum(int a,int b)
        {
            Console.WriteLine("sum is
   :"+(a+b));
        }
   }
class Program   {
  static void Main(string[ ] args)
  {
   del_class d_c = new
  del_class();
   del_nm obj_del = new del_nm
  (d_c.sum);
  obj_del(10,20);
    //obj_del.Invoke(10,20);
    //d_c.sum(10,20);
    Console.ReadLine();
   }
 }
```

2. Multi cast delegate :

- When a delegate is wrapped with more than one method, that is known as a multicast delegate.
- Multicasting of delegate is an extension of the normal delegate(sometimes termed as Single Cast Delegate). It helps the user to point more than one method in a single call.
- When a delegate is wrapped with more than one method, that is known as a multicast delegate.
- In C#, delegates are multicast, meaning they can point to more than one function at a time. They are derived from System.MulticastDelegate class.

- **Properties:**

- Delegates are combined and when you call a delegate then a complete list of methods is called.
- All methods are called in First in First Out(FIFO) order.
- '+' or '+=' Operator is used to add the methods to delegates.
- '–' or '-=' Operator is used to remove the methods from the delegates list.
- **Note:** Remember, multicasting of delegate should have a return type of Void otherwise it will throw a runtime exception. Also, the multicasting of delegate will return the value only from the last method added in the multicast. Although, the other methods will be executed successfully.

Example of Multi cast delegate :

```
using System;
namespace ConsoleApplication4
{
 public delegate void mul_del(int a,int b);
    class mul_class
    {
        public void sum(int a,int b)
        {
            Console.WriteLine("sum is : "+(a+b));
        }
        public void sub(int a, int b)
        {
            Console.WriteLine("sub is : " + (a - b));
        }
        public void mul(int a, int b)
        {
            Console.WriteLine("mul is : " + (a * b));
        }
    }
class Program
    {
        static void Main(string[ ] args)
        {
            mul_class cls = new mul_class();
            mul_del del = new mul_del(cls.sum);

            del += cls.sub;
            del += cls.mul;
            del(20,20);
            Console.ReadLine();
        }
    }
}
```

Event :

- **Events** are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.
- **Using Delegates with Events :**
- The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class. The class containing the event is used to publish the event. This is called the **publisher** class. Some other class that accepts this event is called the **subscriber** class. Events use the **publisher-subscriber** model.
- A **publisher** is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.
- A **subscriber** is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

- **Declaring Events**
- To declare an event inside a class, first of all, you must declare a delegate type for the even as:
- public delegate return_type delegate_name(parameters);

  then, declare the event using the **event** keyword −
- event delegate_name   event_name;

```
using System;
namespace SampleApp
{
  public delegate string
  MyDel(string str);

  class EventProgram
  {
    event MyDel MyEvent;

    public EventProgram()
    {
      this.MyEvent = new
  MyDel(this.WelcomeUser);
    }
public string WelcomeUser(string
   username)
{
      return "Welcome " +
   username;
    }

    static void Main(string[] args) {
      EventProgram obj1 = new
   EventProgram();
      string result =
   obj1.MyEvent("Tutorials Point");
      Console.WriteLine(result);
    }
  }
}
```

Must visit for For More Information :

- **More About access modifiers and encapsulation :**

https://www.tutorialspoint.com/csharp/csharp_encapsulation.htm

- **More about Constructor :**

https://www.geeksforgeeks.org/c-sharp-constructors/

- **More About Operator Overloading :**

https://www.geeksforgeeks.org/c-sharp-operator-overloading/

- **More About Sealed Class & Method :**

https://www.c-sharpcorner.com/UploadFile/puranindia/what-are-sealed-classes-and-sealed-methods/

- **More About Interface :**

https://www.geeksforgeeks.org/c-sharp-interface/

- **More About abstraction :**

https://www.w3schools.com/cs/cs_abstract.php

- More About Properties :
- https://www.w3schools.com/cs/cs_properties.php
- https://www.geeksforgeeks.org/c-sharp-properties/
- More About Indexers :
- https://www.geeksforgeeks.org/c-sharp-indexers/
- More About delegates :
- https://www.c-sharpcorner.com/UploadFile/puranindia/C-Sharp-net-delegates-and-events/

Assignment Questions :

1. What is inheritance ? Explain hierarchical inheritance with example .
2. Explain ref and out parameters with example.
3. What is constructor? Explain any one type in detail.
4. Write down difference between interface and abstract class in detail.
5. What is sealed class ? Explain sealed method in detail.
6. What is c# property? Explain get and set accessor in detail.
7. What is indexer in detail.
8. What is unsafe code? Explain with Example.
9. What is delegate? Explain multicast with example.
10. Explain event in detail.