Unit – 3 OOP

3.1 Class and Objects:

- 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 functionality 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.

- Likewise, in object-oriented programming, a class defines some properties, fields, events, methods, etc. A class defines the kinds of data and the functionality their objects will have.
- When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Example :

```
using System;

// Define the class
class Person
{
   // Class attributes (fields)
   public string Name;
   public int Age;

   // Class constructor
   public Person(string name, int age)
   {
      Name = name;
      Age = age;
   }

   // Class method
   public void DisplayDetails()
   {
      Console.WriteLine("Name: " + Name);
      Console.WriteLine("Age: " + Age);
   }
}

class Program
{
   static void Main()
```

```
    {
        // Create an object of the Person class
        Person person1 = new Person("John Doe", 30);

        // Access the attributes using the object and display details
        person1.DisplayDetails();

        // You can also modify the attributes using the object
        person1.Name = "Jane Smith";
        person1.Age = 28;

        // Display updated details
        person1.DisplayDetails();
    }
}
```

3.2 C# Access Modifiers / Specifiers :

C# Access modifiers or specifiers are the keywords that are used to specify accessibility or scope of variables and functions in the C# application.
C# provides five types of access specifiers.

- Public
- Protected
- Internal
- Protected internal
- Private

We can choose any of these to protect our data. Public is not restricted and Private is most restricted. The following table describes about the accessibility of each.

Access 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 inherits 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:

```
using System;

public class AccessSpecifierExample
{
      public int publicField = 10;


   private int privateField = 20;

   protected int protectedField = 30;


    internal int internalField = 40;


   public void PublicMethod()
   {
      Console.WriteLine("This is a public method.");
   }

   private void PrivateMethod()
   {
      Console.WriteLine("This is a private method.");
   }


   protected void ProtectedMethod()
   {
      Console.WriteLine("This is a protected method.");
   }


   internal void InternalMethod()
   {
      Console.WriteLine("This is an internal method.");
   }
}

public class DerivedClass : AccessSpecifierExample
{
   public void AccessProtectedField()
   {

      Console.WriteLine("Accessing protected field from the derived class: " + protectedField);
   }
}

class Program
{
    static void Main()
    {
        AccessSpecifierExample exampleObj = new AccessSpecifierExample();


        Console.WriteLine("Public field value: " + exampleObj.publicField);
        exampleObj.PublicMethod();

        // The following lines will cause compilation errors since they attempt to access private and
protected members outside the class and its derived class.
        // Console.WriteLine("Private field value: " + exampleObj.privateField);
        // exampleObj.PrivateMethod();
        // Console.WriteLine("Protected field value: " + exampleObj.protectedField);
        // exampleObj.ProtectedMethod();

        // Accessing internal members from outside the assembly is not allowed.
        // Console.WriteLine("Internal field value: " + exampleObj.internalField);
        // exampleObj.InternalMethod();

        DerivedClass derivedObj = new DerivedClass();
        derivedObj.AccessProtectedField();
    }
}
```

3.3 Ref & Out Keyword:

Ref Keyword :

- ref keyword is used when a called method has to update the passed parameter.
- ref keyword is used to pass data in bi-directional way.
- Before passing a variable as ref, it is required to be initialized otherwise the compiler will
- throw an error.
- In the called method, it is not required to initialize the parameter passed as ref.

Out Keyword :

- out keyword is used to get data in uni-directional way.
- No need to initialize variable if out keyword is used.
- out keyword is used when a called method has to update multiple parameter passed.
- In called method, it is required to initialize the parameter passed as out.

Example :

```
using System;
public class Program {
   public static void update(out int a){
      a = 10;
```

```
  }
  public static void change(ref int d){
    d = 11;
  }
  public static void Main() {
    int b;
    int c = 9;
    Program p1 = new Program();
    update(out b);
    change(ref c);
    Console.WriteLine("Updated value is: {0}", b);
    Console.WriteLine("Changed value is: {0}", c);
  }
}
```

3.4 Encapsulation:

Encapsulation, in the context of C#, refers to an object's ability to hide data and behavior that are not necessary to its user. Encapsulation enables a group of properties, methods and other members to be considered a single unit or object.

The following are the benefits of encapsulation:

- Protection of data from accidental corruption
- Specification of the accessibility of each of the members of a class to the code outside the class
- Flexibility and extensibility of the code and reduction in complexity
- Lower coupling between objects and hence improvement in code maintainability
- Encapsulation is used to restrict access to the members of a class so as to prevent the user of a given class from manipulating objects in ways that are not intended by the designer. While encapsulation hides the internal implementation of the functionalities of class without affecting the overall functioning of the system, it allows the class to service a request for functionality and add or modify its internal structure (data or methods) to suit changing requirements.
- Encapsulation is also known as information hiding.

Example :

```
using System;

public class Person
{

   private string name;
   private int age;

   public Person(string name, int age)
   {
      this.name = name;
```

```
        this.age = age;
    }
    public void DisplayDetails()
    {
        Console.WriteLine("Name: " + name);
        Console.WriteLine("Age: " + age);
    }
    public void SetName(string newName)
    {
        name = newName;
    }
    public void SetAge(int newAge)
    {
        age = newAge;
    }
}

class Program
{
    static void Main()
    {
        Person person1 = new Person("John Doe", 30);

        person1.DisplayDetails();


        person1.SetName("Jane Smith");
        person1.SetAge(28);

        person1.DisplayDetails();
    }
}
```

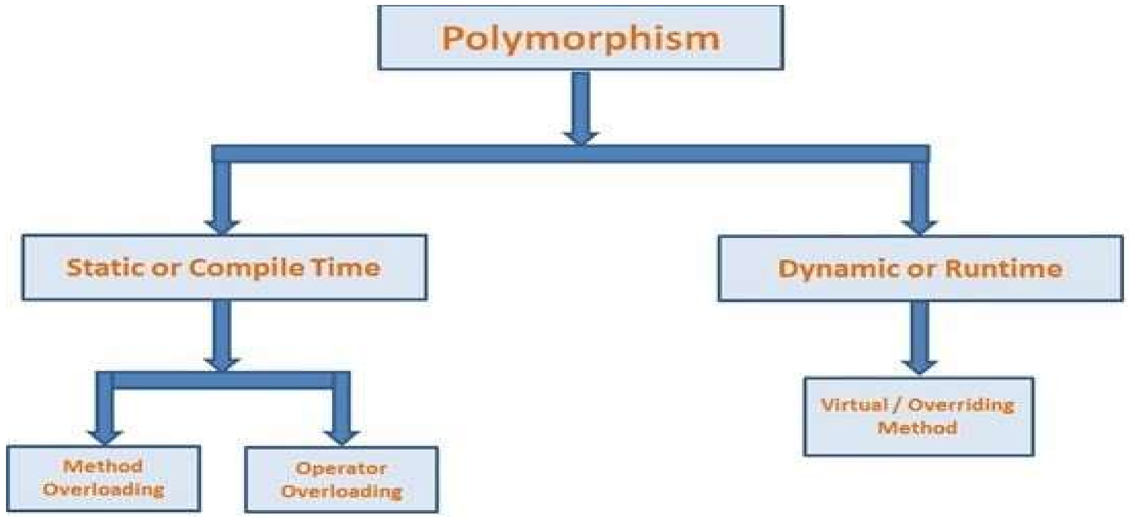
3.5 Polymorphism in C#:

Polymorphism is a Greek word meaning "one name many forms." "Poly" means many, and "morph" means forms. In other words, one object has many forms or has one name with multiple functionalities. Polymorphism allows a class to have multiple implementations with the same name. It is one of the core principles of Object Oriented Programming after encapsulation and inheritance. In this article, you'll learn what polymorphism is, how it works, and how to implement it in C#.

Types of Polymorphism

There are two types of polymorphism in C#:

- Static / Compile Time Polymorphism
- Dynamic / Runtime Polymorphism



Static or Compile Time Polymorphism :

- Method overloading is an example of Static polymorphism. overloading is the concept in which method names are the same with different parameters. The method/function has the same name but different signatures in overloading. It is also known as Early binding. It is also known as Compile Time Polymorphism because the decision of which method is to be called is made at compile time.
- Here C# the compiler checks the number of parameters passed and the parameter type, decides which method to call, and throws an error if no matching method is found.
- In the following example, a class has two methods with the same name, "Add," but with different input parameters ("the first method has three parameters and the second method has two parameters).

Example:

```
Public class TestData
{
   public int Add(int a, int b, int c)
   {
      return a + b + c;
   }
   public int Add(int a, int b)
   {
      return a + b;
   }
}
class Program
{
   static void Main(string[] args)
   {
      TestData dataClass = new TestData();
      int add2 = dataClass.Add(45, 34, 67);
      int add1 = dataClass.Add(23, 34);
} }
```

Dynamic / Runtime Polymorphism:

- Dynamic/runtime polymorphism is also known as late binding. Here, the method name and the method signature (the number of parameters and parameter type must be the same and may have a different implementation). Method overriding is an example of dynamic polymorphism.
- Method overriding can be done using inheritance. With method overriding, it is possible for the base class and derived class to have the same method name and the same something. The compiler would not be aware of the method available for overriding the functionality, so the compiler does not throw an error at compile time. The compiler will decide which way to call at runtime, and if no method is found, it throws an error.

Example :

```
using System;
public class Animal{
   public string color = "white";

}
public class Dog: Animal
{
   public string color = "black";
}
public class Test
{
   public static void Main()
   {
      Animal d = new Dog();
      Console.WriteLine(d.color);

   }
}
```

3.6 Sealed class:

A class from which it is not possible to create/derive a new class is known as a sealed class. In simple words, we can say that when we define the class using the sealed modifier, then it is known as a sealed class and a sealed class cannot be inherited by any other classes. To make any class a sealed class we need to use the keyword sealed. The keyword sealed tells the compiler that the class is sealed, and therefore, cannot be extended.

Points to Remember while working with Sealed Class in C#.

- A sealed class is completely the opposite of an abstract class.
- The sealed class cannot contain any abstract methods.
- It should be the bottom-most class within the inheritance hierarchy.
- A sealed class can never be used as a base class.

- The sealed class is specially used to avoid further inheritance.
- The keyword sealed can be used with classes, instance methods, and properties.

Note: Even if a sealed class cannot be inherited, we can still consume the class members from any other class by creating the object of the class.

Example :

```
using System;
namespace SealedDemo
{
   public class Printer
   {

      public virtual void Display()
      {
         Console.WriteLine("Display Dimension: 5x5");
      }
      public virtual void Print()
      {
         Console.WriteLine("Printer Printing...\n");
      }
   }

   public class LaserJet : Printer
   {

      public sealed override void Display()
      {
         Console.WriteLine("Display Dimension: 10x10");
      }


      public override void Print()
      {
         Console.WriteLine("LaserJet Printer Printing...\n");
      }
   }


   public sealed class InkJet : LaserJet
   {

      public override void Display()
     {
         Console.WriteLine("Some Different Display Dimension");
     }
      public override void Print()
```

```
        {
            Console.WriteLine("InkJet Printer Printing...");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Printer printer = new Printer();
            printer.Display();
            printer.Print();

            LaserJet laserJet = new LaserJet();
            laserJet.Display();
            laserJet.Print();

            InkJet inkJet = new InkJet();
            inkJet.Display();
            inkJet.Print();
            Console.ReadKey();
        }
    }
}
```

3.7 Abstract Class :

An abstract class is an incomplete class or special class that can't be instantiated. The purpose of an abstract class is to provide a blueprint for derived classes and set some rules on what the derived classes must implement when they inherit 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. After overriding the abstract method is in the non-Abstract class. We can derive this class in another class and again we can override the same abstract method with it.

C# Abstract Class Features

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

Example:
```
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();
  }
}
```

3.8 Interface :

- Interface in C# is a blueprint of a class. It is like an abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have a 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 full abstraction* because it cannot have a 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.

Note:

- An interface can contain declarations of methods, properties, indexers, and events.
- Default interface methods with implementation body are supported from C# 8.0.
- An interface cannot contain constructors and fields.
- Interface members are by default abstract and public.
- You cannot apply access modifiers to interface members. Although, C# 8.0 onwards, you may use private, protected, internal, public, virtual, abstract, sealed, static, extern, and partial modifiers on certain conditions.

Example :

```
using System;
namespace CsharpInterface {

  interface IPolygon {
    // method without body
    void calculateArea(int l, int b);

  }
 class Rectangle : IPolygon {

    // implementation of methods inside interface
    public void calculateArea(int l, int b) {

      int area = l * b;
      Console.WriteLine("Area of Rectangle: " + area);
    }
  }
class Program {
    static void Main (string [] args) {

      Rectangle r1 = new Rectangle();

      r1.calculateArea(100, 200);

    }
  }
}
```

3.9 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.

In C#, there are several types of inheritance:

- Single inheritance: A derived class that inherits from only single base class.
- Multi-level inheritance: A derived class that inherits from a base class and the derived class itself becomes the base class for another derived class.

- Hierarchical inheritance: A base class that serves as a parent class for two or more derived classes.

Example:

```
using System;

// single inheritance
class Animal {
	public void Eat() {
		Console.WriteLine("Animal is eating.");
	}
}

class Dog : Animal {
	public void Bark() {
		Console.WriteLine("Dog is barking.");
	}
}

// multi-level inheritance
class Mammal : Animal {
	public void Run() {
		Console.WriteLine("Mammal is running.");
	}
}

class Horse : Mammal {
	public void Gallop() {
		Console.WriteLine("Horse is galloping.");
	}
}

// hierarchical inheritance
class Bird : Animal {
	public void Fly() {
		Console.WriteLine("Bird is flying.");
	}
}

class Eagle : Bird {
	public void Hunt() {
		Console.WriteLine("Eagle is hunting.");
	}
}

class Penguin : Bird {
	public void Swim() {
```

```
                    Console.WriteLine("Penguin is swimming.");
          }
}

// multiple inheritance
interface I1 {
          void Method1();
}

}

// main program
class Program {
          static void Main(string[] args) {
                    // single inheritance
                    Dog dog = new Dog();
                    dog.Eat();
                    dog.Bark();

                    // multi-level inheritance
                    Horse horse = new Horse();
                    horse.Eat();
                    horse.Run();
                    horse.Gallop();

                    // hierarchical inheritance
                    Eagle eagle = new Eagle();
                    Penguin penguin = new Penguin();
                    eagle.Fly();
                    eagle.Hunt();
                    penguin.Fly();
                    penguin.Swim();



                              }
}
```
Advantages of Inheritance:
Code Reusability: Inheritance allows us to reuse existing code by inheriting properties and methods from an existing class.
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.
Code Organization: Inheritance improves code organization by grouping related classes together in a hierarchical structure.

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

Note :

- Inheritance is an important pillar of OOP(Object Oriented Programming). It is the mechanism in C# by which one class is allowed to inherit the features(fields and methods) of another class. Important terminology:
- 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 superclass fields and methods.
- Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

3.10 C# Properties:

What is a Property in C#?
A Property in C# is a member of a class that is used to set and get the data from a data field (i.e. variable) of a class. The most important point that you need to remember is that a property in C# is never used to store any data, it just acts as an interface or medium to transfer the data. We use the Properties as they are public data members of a class, but they are actually special methods called assessors.
What are Accessors in C#?
The Assessors are nothing but special methods which are used to set and get the values from the underlying data member (i.e. variable) of a class. Assessors are of two types. They are as follows:

- Set Accessor
- Get Accessor

What is a Set Accessor?
The **set** accessor is used to set the data (i.e. value) into a data field i.e. a variable of a class. This set accessor contains a fixed variable named **value**. Whenever we call the property to set the data, whatever data (value) we are supplying will come and store inside the variable called **value** by default. Using a set accessor, we cannot get the data.

Syntax: set { Data_Field_Name = value; }
What is Get Accessor?
The get accessor is used to get the data from the data field i.e. variable of a class. Using the get accessor, we can only get the data, we cannot set the data.
Syntax: get {return Data_Field_Name;}

What are the Different types of Properties Supported by C#.NET?
The C#.NET supports four types of properties. They are as follows

- Read-Only Property
- Write Only Property
- Read Write Property
- Auto-Implemented Property

What is Read-only Property in C#?
The Read-Only Property is used to read the data from the data field i.e. read the data of a variable. Using this Read-Only Property, we cannot set the data into the data field. This property will contain only one accessor i.e. get accessor.
Syntax:
```
AccessModifier Datatype PropertyName
{
    get {return DataFieldName;}
}
```
What is Write only Property in C#?
The Write-Only Property is used to write the data into the data field i.e. write the data to a variable of a class. Using this Write-Only Property, we cannot read the data from the data field. This property will contain only one accessor i.e. set accessor.
Syntax:
```
AccessModifier Datatype PropertyName
{
     set {DataFieldName = value;}
}
```
What is Read Write Property in C#?
The Read-Write Property is used for both reading the data from the data field as well as writing the data into the data field of a class. This property will contain two accessors i.e. set and get. The set accessor is used to set or write the value to a data field and the get accessor is read the data from a variable.
Syntax:
```
AccessModifier Datatype PropertyName
{
    set {DataFieldName = value;}
    get {return DataFieldName;}
}
```
Note: Whenever we create a property for a variable, the data type of the property must be the same as the data type of the variable. A property can never accept any argument.

Why do we need Properties in C#?
In order to encapsulate and protect the data members (i.e. fields or variables) of a class, we use properties in C#. The Properties in C# are used as a mechanism to set and get the values of data members of a class outside of that class. If a class contains any values in it and if we want to access those values outside of that class, then we can provide access to those values in 2 different ways. They are as follows:

- By storing the value under a public variable, we can give direct access to the value outside of the class.
- By storing that value in a private variable, we can also give access to that value outside of the class by defining a property for that variable.

Example :

```
using System;
  public class Employee
   {
      private string name;

      public string Name
      {
         get
         {
            return name;
         }
         set
         {
            name = value;
         }
      }
  }
  class TestEmployee{
      public static void Main(string[] args)
      {
         Employee e1 = new Employee();
         e1.Name = "Sonoo Jaiswal";
         Console.WriteLine("Employee Name: " + e1.Name);

      }
   }
```

3.11 Indexer in c# :

An indexer allows an instance 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. 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]
{
 get
 {
   // get block code
 }
 set
 {
   // set block code
 }
}
```

In the above syntax:

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 { }:

Implementing Indexers:

1. **Multiple Index Parameters:** Indexers can have multiple parameters, allowing for more complex indexing scenarios. You can define an indexer with multiple parameters to access elements based on different criteria.
2. **Indexer Overloading:** Similar to methods, indexers can be overloaded in C#. This means you can define multiple indexers with different parameter types or counts, providing different ways to access elements in the class or struct.
3. **Read-Only Indexers:** By omitting the set accessor in the indexer declaration, you can create read-only indexers. This restricts the ability to modify elements through the indexer, allowing only the retrieval of values.
4. **Implicit vs. Explicit Interface Implementation:** Indexers can be implemented implicitly or explicitly when defined as part of an interface. Implicit implementation is used when the indexer is defined within the class itself, while explicit implementation is used when the indexer is implemented explicitly to resolve any naming conflicts.
5. Indexers in Collections: Indexers are commonly used in collection classes, such as dictionaries, lists, and arrays. They provide a convenient way to access and manipulate elements within these collections based on an index or key.
6. **Indexers in Custom Classes:** Indexers can be implemented in custom classes to provide customized access to class members based on an index. This allows for more intuitive and expressive interaction with instances of the class.

Example :

```
using System;
class IndexerCreation
{
        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() {
        IndexerCreation ic = new IndexerCreation();

                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]);
        }
}
```

3.12 Delegate in c# :

- 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 returns type and then calls the method in a program for execution when it is needed.

Important Points about 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.
- Anonymous Methods(C# 2.0) and Lambda expressions(C# 3.0) are compiled to delegate types in certain contexts. Sometimes, these features together are known as anonymous functions.

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.
Where do we use Delegates in C#?

- Delegates are used in the following cases:
- Event Handlers
- Callbacks
- Passing Methods as Method Parameters
- LINQ
- Multithreading

Types of delegates :

1. **Single caste delegate :**

Single cast delegates only hold the reference of a single method. A single cast delegate derives from the System. Delegate class. It's used to invoke a single method at a time.
Example :
```
using System;
namespace Delegates
{
   public delegate void MyDelegate(string text);
   class Program
   {
      public static void ShowText(string text)
      {
         Console.WriteLine(text);
      }
      static void Main()
      {
         MyDelegate d = new MyDelegate(ShowText);
         d("Hello World...");
         Console.ReadLine();
      }
   }
}
```

2. **Multi-caste delegates :**

- One method reference, which is encapsulated in the delegate, is the only one that it is possible for the delegate to call. These Delegates are known as Multicast Delegates because they have the capacity to hold and invoke multiple methods.
- Multicast Delegates, also referred to as Combinable Delegates, are required to meet requirements such as the Delegate's return type having to be void. No Delegate type parameter that is declared as an output parameter using no keywords is a Delegate type parameter. Concatenating the invocation lists of the two operands of the addition operation yields a Multicast Delegate instance by joining two Delegates. Delegates are used in the order that they are added.

Key factors relating to Multicast delegate:

- Every method is called using the FIFO (First in, First out) principle.
- Methods are added to delegates using the + or += operator.
- Methods can be eliminated from the delegates list using the – or -= operator.

Example:

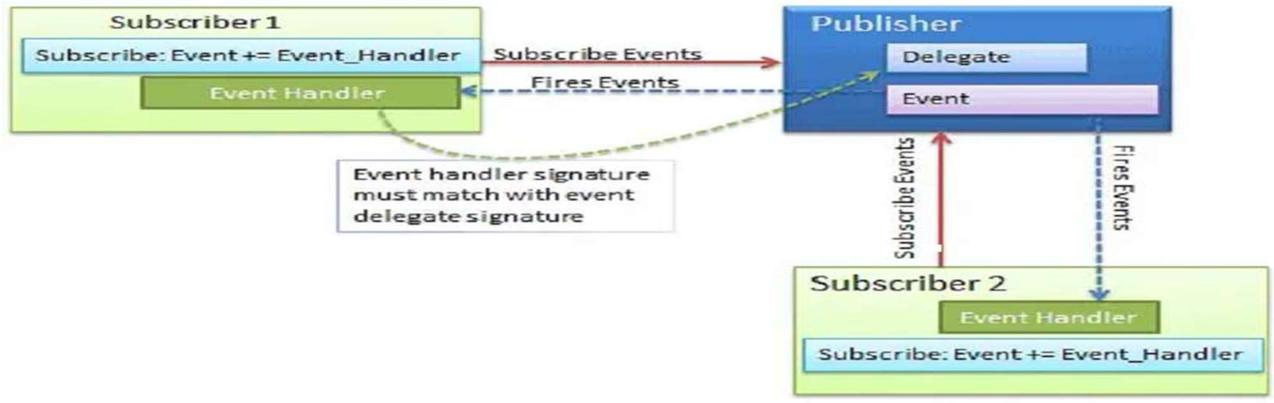
```
using System;
namespace Delegates
{
   public delegate void MyDelegate();
   class Program
   {
      public static void Method1()
      {
         Console.WriteLine("Inside Method1...");
      }
      public static void Method2()
      {
         Console.WriteLine("Inside Method2...");
      }
      static void Main()
      {
         MyDelegate d = null;
         d += Method1;
         d += Method2;
         d.Invoke();
         Console.ReadLine();
      }
   }
}
```

3.13 C# Events :

- An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the observer design pattern.
- The class who raises events is called Publisher, and the class who receives the notification is called Subscriber. There can be multiple subscribers of a single event. Typically, a publisher raises an event when some action occurred. The subscribers, who are interested in getting a notification when an action occurred, should register with an event and handle it.
- In C#, an event is an encapsulated delegate. It is dependent on the delegate. The delegate defines the signature for the event handler method of the subscriber class.

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.


Subscriber 1
Subscribe: Event += Event_Handler
Event Handler
Subscribe Events
Fires Events
Publisher
Delegate
Event
Event handler signature must match with event delegate signature
Subscribe Events
Fires Events
Subscriber 2
Event Handler
Subscribe: Event += Event_Handler

Example :

```
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("Hello world");
Console.WriteLine(result);
    }
  }
}
```

Note: Reference books for learning for oop based concepts:

- C# 9.0 in a Nutshell" by Joseph Albahari and Ben Albahari: This book provides a comprehensive guide to C# 9.0, including inheritance, interfaces, and other object-oriented programming concepts.
- "Head First C#" by Jennifer Greene and Andrew Stellman: This book is a beginner-friendly introduction to C#, including inheritance and other object-oriented programming concepts.
- "Professional C# 7 and .NET Core 2.0" by Christian Nagel: This book is a comprehensive guide to C# 7 and .NET Core 2.0, including inheritance, polymorphism, and other object-oriented programming concepts.