C# .NET Notes

❖ Introduction to .NET platform:

- NET stands for Network Enabled Technology. In .NET, dot (.) refers to object-oriented and NET refers to the internet. So, the complete .NET means through object-oriented we can implement internet-based applications.
- The DOTNET Framework provides two things are as follows
  - BCL (Base Class Libraries)
  - CLR (Common Language Runtime)
- JIT stands for the Just-in-Time compiler. It is the component of CLR which is responsible for converting MSIL code into Native Code. This Native code is directly understandable by the operating system.

❖ CLR (Common Language Runtime):

- CLR is the heart of .NET Framework and it contains the following components:
  Security Manager
  JIT Compiler
  Memory Manager
  Garbage Collector
  Exception Manager
  Common Language Specification (CLS)
  Common Type System (CTS)
- There are basically two components to manage security. They are as follows:
- CAS (Code Access Security)
- CV (Code Verification)
- These two components are basically used to check privileges of the current user that the user is allowed to access the assembly or not. The CLR also sees what kind of rights or what kind of authorities this code has and whether it is safe to be executed by the operating system. So, basically these types of checks are maintained by the Security Manager.

❖ Common Type System (CTS):

- CLR will execute all programming language's data types; this is possible because CLR will contain its own data types which are common to all programming languages.
- At the time of compilation, all language-specific data types are converted into CLR's data type. This data type system of CLR which is common to all programming languages of .NET is known as the Common Type System (CTS).

❖ CLS (Common Language Specification):

- One programming language cannot understand other programming language syntactical rules (language specification). But CLR will execute all programming languages code. This is possible

because CLR cannot understand any programming language specification rather CLR has its own language specification (syntactical rules) for its MSIL.

- Every language compiler should follow this language specification of CLR at the time of compilation and should generate MSIL; CLR's JIT compiler will generate native code from MSIL. This language specification of CLR is common for all programming languages code execution of .NET and is known as CLS.

❖ What is Manifest?

- Manifest contains metadata about the assembly like the name of the assembly, the version number of the assembly, culture and strong name information.

❖ Garbage Collector:

- Garbage Collector is nothing but, it is a feature provided by CLR which helps us to clean or destroy the unused managed objects.
- By cleaning or destroying those unused managed objects, it basically reclaims the memory.
- When a .NET application runs, it creates several objects and at a given moment of time, it may possible that the application may not use some of those objects.
- So, for those objects, Garbage Collector runs continuously as a background thread and at specific interval time, it checks whether there are any unused managed objects and if it finds it simply clean those objects and reclaims the memory.
- There are three generations of garbage collector in .NET. They are:
  Generation 0
  Generation 1
  Generation 2.
- Normally, when we are working with big applications, they can create thousands of objects.
- So, each of these objects, if garbage collector goes and checks if they are needed or not, it's really pain or it's a bulky process.
- By creating such generations what it means if an object in Generation 2 buckets it means the Garbage Collector will do fewer visits to this bucket.
- The reason is, if an object moves to Generation 2, it means it will stay more time in the memory. It's no point going and checking them again and again.
- So, in simple words, we can say that Generations 0, 1, and 2 will helps to increase the performance of the Garbage Collector.
- The more the objects in Gen 0, the better the performance and the more the memory will be utilized in an optimal manner.

❖ Managed and Unmanaged code in .NET:

- The codes which run under the complete control of CLR are called as Managed Code in .NET.
- These kinds of code (Managed code in C#) are run by dot net runtime environment. If the .NET framework is not installed or if .NET runtime is not available, then these kinds of codes are not going to be executed.
- CLR will provide all the facilities and features of .NET to the managed code execution like Language Interoperability, Automatic memory management, Exception handling mechanism, code access security, etc.

- On the other hand, Skype, PowerPoint, Microsoft Excel does not require dot net runtime, they run under their own environment.
- So, in short, the code (exe, web app) which not run under the control of CLR is called unmanaged code in .NET.
- CLR will not provide any facilities and features of .NET to the unmanaged code in C# execution like Language Interoperability, Automatic memory management, Exception handling mechanism, code access security, etc.

❖ App Domain in .NET Framework:

- The App Domain (Application Domain) is a logically isolated container inside a process.
- In this logical isolation, you can load and run .NET Code in an isolated manner.
- The following are the advantages of using the App Domain:
- You can load and unload DLL inside these logical containers without one container affecting the other. So, if there are issues in one application domain you can unload that application domain and the other application domain work without issues.
- If you have a Third Party DLL and for some reason, you don't trust the third party code. You can run that DLL in an isolated app domain with fewer privileges.
- For example, you can say that the DLL cannot access to your "C:\" drive. And other DLLs that you trust you can run with full privilege in a different app domain.
- You can run different versions of DLL in every application domain.

❖ What is an Assembly in .NET?

- Assemblies are the building block of .NET Framework applications; they form the fundamental unit of deployment or an assembly is nothing but a single unit of deployment or it is a precompiled chunk of code that can be executed by CLR.
- In simple words, we can say that Assembly is nothing but a precompiled .NET Code that can be run by CLR.
- In the .NET Framework, there are two types of assemblies. They are as follows:
  - EXE (Executable)
  - DLL (Dynamic Link Library)
- When we compile a Console Application or a Windows Application, it generates EXE, whereas when we compile a Class Library Project or ASP.NET web application, then it generates DLL.
- In.NET framework, both EXE, and DLL are called assemblies.

❖ What is the difference between the DLL and the EXE in .NET Framework?

- The EXE is run in its own address space or in its own memory space. If you double click on the EXE file then you will get output. Now, this program is running in its own memory space.
- Without closing this window, again if you double click on the EXE application, again it will run and will display the same output. This is because now, both the EXE are running in their own memory space. The point that you need to remember is EXE is an executable file and can run by itself as an application.

- Coming to DLL, it **cannot be run by itself** like EXE. That means the MyClassLibrary.dll cannot be invoked or run by himself.
- It **needs a consumer** who is going to invoke it. So, a DLL is **run inside other memory space**.
- The other memory space can be a console, or windows applications or web application which should have its own memory space.
- Now, the question that should come to your mind why do we need DLLs as it is not invoked by itself. The reason behind the DLL is **reusability**. Suppose you want some class, or logic or something else in many applications, then simply put those classes, logic inside a DLL and refer that DLL wherever it is required.

❖ Understanding Assembly in .NET Framework

- In .NET Framework, the assemblies are broadly classified into 2 types. They are as follows:
  01. Weak Named Assemblies
  02. Strong Named Assemblies
- All the .NET Framework assemblies are installed in a special location called **GAC** (Global Assembly Cache).
- All the assemblies present in GAC are strongly typed. Later part of this article we will discuss what exactly a strongly type assembly and difference between a week and a strong type assembly in .NET. In .NET, an assembly is consists of 4 Parts:
  01. Simple textual name (i.e. the Assembly name).
  02. The Version number.
  03. Culture information (If provided, otherwise the assembly is language-neutral)
  04. Public key token
- **Assembly Name (Simple Textual Name):**
  - This is nothing but the project name.
- **Version Number:**
  - The default format of Version number is 1.0.0.0. That means the version number again consists of four parts as follows:
  - Major Version
  - Minor Version
  - Build Number
  - Revision Number
- **Assembly Culture:**
  - The AssemblyCulture attribute is used for specifying the culture of the assembly.
  - By default in .NET assemblies are language-neutral which means the AssemblyCulture attribute contains an empty string.
- **Public Key Token:**
  - If you go to the GAC, then you will see that every assembly is assigned with a public key token.
  - In order to get the public key token, you need to sign your assembly with a private and public key pair.

❖ What is the difference between Strong and Weak Assemblies in .NET Framework?

- If an assembly is not signed with the private/public key pair then the assembly is said to be a weak named assembly and it is not guaranteed to be unique and may cause the DLL hell.
- The Strong named assemblies are guaranteed to be unique and solve the DLL hell problem. Again, you cannot install an assembly into GAC unless the assembly is strongly named.

❖ Verbatim String:

- Verbatim Literal is a string with a @ symbol prefix, as in @"Hello". The Verbatim literals make escape sequences translate as normal printable characters to enhance readability.

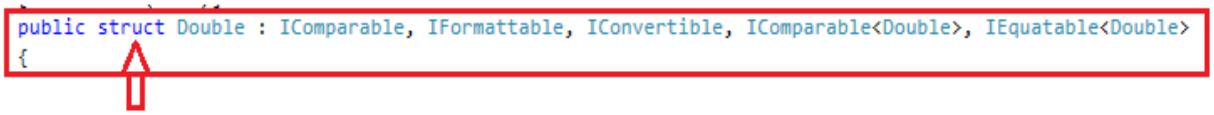
❖ Implicitly-Typed Variable:

- In C#, variables must be declared with the data type. These are called explicitly typed variables.
- C# 3.0 introduced var keyword to declare method level variables without specifying a data type explicitly.
- The compiler will infer the type of a variable from the expression on the right side of the = operator.
- var can be used to declare any built-in data type or a user-defined type or an anonymous type variable.
- Implicitly-typed variables must be initialized at the time of declaration; otherwise C# compiler would give an error: Implicitly-typed variables must be initialized.
- Multiple declarations of var variables in a single statement are not allowed.
- var cannot be used for function parameters.
- var can be used in for, and foreach loops.
- var can also be used with LINQ queries.

❖ Strings in C#:

```
int i = 10;
double d = 20.5;
```

- Then if you right click on the data type and go the definition then you will see that they are struct as shown in the below image. Struct means they are value type.




```
string str = "";
```

- Then if you right click on the string data type and click on go to definition then you will see that it is a class. Class means reference data type.

```
public sealed class String : IComparable, ICloneable, IConvertible, IEnumerable, IComparable<String>,
  IEnumerable<char>, IEquatable<String>
```

- So, the first point that you need to remember is strings are reference type while other primitive data types are struct type.
- The small string is actually an alias of String (Capital string). If you right click on the small string and if you go to the definition then you will see that the actual class name is capital string.
- Before understanding strings are immutable, first we need to understand two terms i.e. Mutable and Immutable.
- Mutable means can be changed whereas Immutable means cannot be changed.
- C# strings are immutable means C# strings cannot be changed.
- The String Interning in C# is a process which uses the same memory location if the value is same. When the loop executes for the first time, it will create a fresh object and assign the value ".Net Tutorials" to it.
- When the loop executes 2nd time, before creating a fresh object, it will check whether this ".Net Tutorials" value is already there in the memory, if yes then it simply uses that memory location else it will create a new memory location. This is nothing but C# string interning.
- So, if you are running a for loop and assigning the same value again and again, then it uses string interning to improve the performance.
- In this case rather than creating new object, it uses the same memory location.
- They made Strings as Immutable for Thread Safety. Think one situation where you have many threads and all the threads wants to manipulate the same string object as shown in the below image. If strings are mutable then we have thread safety issues.

❖ StringBuilder Class:

- StringBuilder is mutable.
- StringBuilder performs faster than string when appending multiple string values.
- Use StringBuilder when you need to append more than three or four strings.
- Use the Append() method to add or append strings to the StringBuilder object.
- Use the ToString() method to retrieve a string from the StringBuilder object.

❖ Anonymous Type:

- In C#, an anonymous type is a type (class) without any name that can contain public read-only properties only.
- It cannot contain other members, such as fields, methods, events, etc.
- We can create an anonymous type using the new operator with an object initializer syntax.
- The implicitly typed variable - var is used to hold the reference of anonymous types.

```
var student = new { Id = 1, FirstName = "James", LastName = "Bond"};
```

- The properties of anonymous types are read-only and cannot be initialized with a null, anonymous function, or a pointer type.
- The properties can be accessed using dot (.) notation, same as object properties.
- An anonymous type will always be local to the method where it is defined. It cannot be returned from the method.

❖ Dynamic Types:

- C# 4.0 (.NET 4.5) introduced a new type called dynamic that avoids compile-time type checking.
- A dynamic type escapes type checking at compile-time; instead, it resolves type at run time.
- A dynamic type variables are defined using the dynamic keyword.
- The compiler compiles dynamic types into object types in most cases. However, the actual type of a dynamic type variable would be resolved at run-time.

❖ Static Keyword in C#:

- We cannot applied any OOPs principles to the static class like inheritance, polymorphism, encapsulation and abstraction.
- But at the end, it is a class. And at least to use a class it has to be instantiated.
- If the static class not instantiated then we cannot invoke the methods and properties that are present in the static class.
- Now let us see how does the instantiation takes place internally of a static class i.e. in our example, it is the CommonTask class.
- The CLR (Common Language Runtime) will create only one instance of the CommonTask class irrespective of whether how many times they called from the Customer and CountryMaster class.
- Due to the single instance behaviour, the static class are also going to be used to share the common data.
- If you want a variable to have the same value throughout all instances of a class then you need to declare that variable as a static variable. So, the static variables are going to hold the application level data which is going to be the same for all the objects.

❖ Static and Non-Static Members in C#:

- In simple word, we can define that, the members of a class which does not require an instance for initialization or execution are known as the static member.
- On the other hand, the members which require an instance of a class for both initialization and execution are known as non-static members.
- The static variable gets initialized immediately once the execution of the class starts whereas the non-static variables are initialized only after creating the object of the class and that is too for each time the object of the class is created.
- A static variable gets initialized only once during the life cycle of a class whereas a non-static variable gets initialized either 0 or n number of times, depending on the number of objects created for that class.
- If you want to access the static members of a class, then you need to access them using the class name whereas you need an instance of a class to access the non-static members.

❖ Static and Non-Static methods in C#:

- If we declare a method using the static modifier then it is called as a static method else it is a non-static method.
- We cannot consume the non-static members directly within a static method.

- If we want to consume any non-static members with a static method then you need to create an object that and then through the object you can access the non-static members.
- On the other hand, you can directly consume the static members within a non-static method without any restriction.

❖ Rules to follow while working with static and non-static members in c#:

- **Non-static to static:** Can be consumed only by using the object of that class.
- **Static to static:** Can be consumed directly or by using the class name.
- **Static to non-static:** Can be consumed directly or by using the class name.
- **Non-static to non-static:** Can be consumed directly or by using the "this" keyword.

❖ Understanding Static and Non-Static Constructor in C#:

- If we create the constructor explicitly by the static modifier, then we call it a static constructor and the rest of the others are the non-static constructors.
- The most important point that you need to remember is the static constructor is the fast block of code which gets executes under a class.
- No matter how many numbers of objects you created for the class the static constructor is executed only once.
- On the other hand, a non-static constructor gets executed only when we created the object of the class and that is too for each and every object of the class.
- It is not possible to create a static constructor with parameters.
- This is because the static constructor is the first block of code which is going to execute under a class and this static constructor called implicitly, even if parameterized there is no chance of sending the parameter values.

❖ Understanding the Static class in C#:

- The class which is created by using the static modifier is called a static class.
- A static class can contain only static members in it.
- It is not possible to create an instance of a static class.
- This is because it contains only static members.
- And we know we can access the static members of a class by using the class name.

❖ Const and Read-Only in C#:

- **Constants** are the immutable values that are known at the time of program compilation and do not change their values for the lifetime of the program.
- The **Read-only** variables are also immutable values but these values are known at runtime and also do not change their values for the life of the program.
- Constants are static by default.
- It is mandatory to initialize a constant variable at the times of its declaration.
- The behaviour of a constant variable is same as the behaviour of static variable i.e. maintains only one copy in the life cycle of class and initialize immediately once the execution of the class start (object not required)

- The only difference between a static and constant variable is that the static variable value can be modified but a constant variable value can never be modified.
- The read-only variable's value cannot be modified once after its initialization.
- It is not mandatory or required to initialize the read-only variable at the time of its declaration like a constant.
- We can initialize the read-only variables under a constructor but the most important point is that once after initialization, you cannot modify the value.
- The behaviour of a read-only variable is similar to the behaviour of a non-static variable.
- That is, it maintains a separate copy for each object. The only difference between these two is non-static variables can be modified while the read-only variables cannot be modified.
- A constant variable is a fixed value for the complete class whereas a read-only variable is a fixed value but specific to one object of the class.

❖ Properties in C#:

- In order to encapsulate and protect the data members (i.e. fields), we use properties in C#.
- The Properties in C# are used as a mechanism to set and get the values of a class outside of that class.
- If a class contains any value in it and if we want to access those values outside of that class, then you can provide access to those values in 2 different ways:
  01. By storing the value under a public variable we can give access to the value outside of the class.
  02. 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.
- The most important point that you need to remember is a property in C# is never used to store data, it just acts as an interface to transfer the data.
- We use the Properties as they are the public data members of a class, but they are actually special methods called accessors.
- The set accessor is used to set the data (i.e. value) into a data field. This set accessor contains a fixed variable named "value". Whenever we call the property to set the data, whatever data (value) we are supplying that will come and store in the variable "value" by default.
- Syntax: set { Data Field Name = value; }
- The get accessor is used to get the data from the data field. Using this get accessor you cannot set the data.
- Syntax: get { return Data Field Name; }
- The C#.NET supports four types of properties. They are as follows
  01. Read-only property
  02. Write only property
  03. Read Write property
  04. Auto-implemented property
- Properties will provide the abstraction to the data fields.
- They also provide security to the data fields.
- Properties can also validate the data before storing into the data fields.
- The default accessibility modifier of the accessor is same as the accessibility modifier of property.

❖ Difference Between Convert.ToString() and ToString() Method in C#:

- Both these methods are used to convert a value to string.
- The difference is Convert.ToString() method handles null whereas the ToString() doesn't handle null in C#.
- In C# if you declare a string variable and if you don't assign any value to that variable, then by default that variable takes a null value.
- In such a case, if you use the ToString() method then your program will throw the null reference exception.
- On the other hand, if you use the Convert.ToString() method then your program will not throw an exception.

❖ Checked and unchecked keyword in C#:

- The checked keyword in C# is used to explicitly enable overflow checking for integral-type arithmetic operations and conversions.
- The unchecked keyword in C# is used to suppress overflow-checking for integral-type arithmetic operations and conversions.
- Here, overflow checking means when the value of any integral-type exceeds its range, it does not raises any exception, instead it will give us unexpected or garbage results.

==========

Object Oriented Programming (OOP's)

- Functional programming has the following problems.
  01. **Reusability**
  02. **Extensibility**
  03. **Simplicity**
  04. **Maintainability**
- The Object Oriented Programming (OOPs) in C# is a design approach where we think in terms of real-world objects rather than functions or methods.
- Unlike procedural programming language, here in oops, programs are organized around objects and data rather than action and logic.
- OOPs, provide 4 principles. They are:
  01. **Encapsulation**
  02. **Inheritance**
  03. **Polymorphism**
  04. **Abstraction**

❖ Class:

- A class is simply a user-defined data type that represents both state and behaviour. The state represents the properties and behaviour is the action that objects can perform.
- In other words, we can say that a class is the blueprint/plan/template that describes the details of an object.

- A class is a blueprint from which the individual objects are created. In C#, a Class is composed of three things i.e. a name, attributes, and operations.
- In C# we have below types of classes
  01. Abstract class
  02. Concrete class
  03. Sealed class
  04. Partial Class
  05. Static class

❖ Object:

- It is an instance of a class. A class is brought live by creating objects. An object can be considered as a thing that can perform activities. The set of activities that the object performs defines the object's behaviour.
- All the members of a class can be accessed through the object. To access the class members, we need to use the dot (.) operator. The dot operator links the name of an object with the name of a member of a class.

❖ Constructors in C#:

- We can define the constructors in C# are the special types of methods of a class that automatically executed whenever we create an instance (object) of that class.
- The Constructors are responsible for two things. First, object initialization and second one is memory allocation.
- Rules for constructor in c#:
  - The constructor name should be the same as the class name.
  - It should not contain return type even void also.
  - The constructor should not contain modifiers.
  - As part of the constructor body return statement with value is not allowed.
- It can have all five accessibility modifiers.
- The constructor can have parameters.
- It can have throws clause it means we can throw an exception from the constructor.
- The constructor can have logic, as part of logic it can have all C#.NET legal statements except return statement with value.
- We can place return; in the constructor.
- When we define multiple constructors within a class with different parameter types, number and order then it is called as constructor overloading.
- There are five types of constructors available in C#, they are as follows
  - Default Constructor
  - Parameterized Constructor
  - Copy Constructor
  - Static Constructor
  - Private Constructor
- In C#, it is also possible to create a constructor as private. The constructor whose accessibility is private is known as a private constructor. When a class contains a private constructor then we cannot create an object for the class outside of the class. So, private constructors are used to

creating an object for the class within the same class. Generally, private constructors are used in Remoting concept.

- In C#, it is also possible to create a constructor as static and when we do so, it is called Static Constructor. The static Constructor in C# will be invoked only once. There is no matter how many numbers of instances (objects) of the class are created, it is going to be invoked only once and that is during the creation of the first instance (object) of the class.
- The static constructor is used to initialize the static fields of the class. You can also write some code inside the static constructor which is going to be executed only once. The static data members in C# are created only once even though we created any number of objects.
- The static constructor should be without any parameter.
- There should not be any access modifier in static constructor definition.

❖ Access Specifiers in C#:

- The Access Specifiers in C# are also called access modifiers which are used to define the scope of the type as well as the scope of their members.
- That is who can access them and who cannot access them are defined by the Access Specifiers.
- C# supports 5 access specifiers, they are as follows
  - Public – access everywhere.
  - Private – only within the class.
  - Protected – within class and derived class.
  - Internal - anywhere within the containing assembly.
  - Protected internal - anywhere within the assembly in which it is declared or from within a derived class in another assembly.

❖ Encapsulation:

- The process of binding the data and functions together into a single unit (i.e. class) is called encapsulation in C#.
- Or you can say that the process of defining a class by hiding its internal data member direct access from outside the class and providing its access only through publicly exposed methods (setter and getter methods) or properties with proper validations and authentications is called encapsulation.
- The Data encapsulation is also called data hiding because by using this principle we can hide the internal data from outside the class.
- In C#, encapsulation is implemented by:
  - By declaring the variables as private (to restrict its direct access from outside the class)
  - By defining one pair of public setter and getter methods or properties to access private variables.

❖ Abstraction in C#:

- The process of defining a class by providing the necessary and essential details of an object to the outside world and hiding the unnecessary things is called abstraction in C#.
- It means we need to display what is necessary and compulsory and need to hide the unnecessary things to the outside world.
- In C# we can hide the member of a class by using private access modifiers.

- The necessary methods and properties are exposed by using the “public” access modifier whereas the unnecessary methods and properties hidden from outside the world by using the “private” access modifier.

❖ What is the difference between Abstraction and Encapsulation?

- The Encapsulation is the process of hiding irrelevant data from the user or you can say Encapsulation is used to protect the data. For example, whenever we buy a mobile, we can never see how the internal circuit board works. We are also not interested to know how the digital signal converts into the analogue signal and vice versa. So from a Mobile user’s point of view, these are some irrelevant information. This is the reason why they are encapsulated inside a cabinet.
- In C# programming, we will do the same thing. We will create a cabinet and keep all the irrelevant information which should not be exposed to the user of the class.
- In encapsulation, problems are solved at the implementation level.
- Encapsulation can be implemented using by access modifier i.e. private, protected and public.
- Coming to abstraction in C#, it is just the opposite of Encapsulation. What it means, it is a mechanism which will show only the relevant information to the user. If we consider the same mobile example. Whenever we buy a mobile phone, we can see and use many different types of functionalities such as a camera, calling function, recording function, mp3 player, multimedia etc. This is nothing but the example of abstraction in C#. The reason is we are only seeing the relevant information instead of their internal working.
- In abstraction, problems are solved at the design or interface level.
- We can implement abstraction using abstract class and interfaces.

❖ Inheritance in C#:

- The process of creating a new class from an existing class such that the new class acquires all the properties and behaviours of the existing class is called as inheritance.
- The properties (or behaviours) are transferred from which class is called the superclass / parent class / base class
- Whereas the class which derives the properties or behaviours from the superclass is known as a subclass / child class / derived class.
- C#.NET classified the inheritance into two categories:
  - **Implementation inheritance:** This is the commonly used inheritance. Whenever a class is derived from another class then it is known as implementation inheritance.
  - **Interface inheritance:** This type of inheritance is taken from Java. Whenever a class is derived from an interface then it is known as interface inheritance.
- Inheritance is classified into 5 types. They are as follows.
  - Single Inheritance
  - Hierarchical Inheritance
  - Multilevel Inheritance
  - Hybrid Inheritance
  - Multiple Inheritance

❖ Interface in C#:

- The interface is a fully un-implemented class used for declaring a set of operations of an object.
- So, we can define an interface as a pure abstract class which allows us to define only abstract methods. Abstract method means a method without body or implementation.
- By default the members of an interface are public.
- An interface can contain:
  Abstract methods, Properties, Indexes, Events
- An interface cannot contain:
  Non-abstract functions, Data fields, Constructors, Destructors
- Nested interfaces may be declared protected or private but not the interface methods.
- An interface cannot implement an abstract class. An interface may only extend a super interface. However, an abstract class can implement an interface because an abstract class can contain both abstract methods and concrete methods.
- It is not permitted to declare an interface as sealed.
- It is not necessary for a class that implements an interface to implement all its methods, but in this case, the class must be declared as abstract.

❖ How interface is different from a class?

- An interface is different from a class in the following ways:
- We cannot instantiate an interface.
- An interface does not contain any constructor or data fields or destructor, etc.
- All of the methods of an interface are abstract and public by default.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

❖ What are the similarities between the interface and abstract class?

- Both interface and the abstract class cannot be instantiated means we cannot create the object.
- But we can create a reference variable for both interface and abstract class.
- The subclass should implement all abstract methods.
- Both cannot be declared as sealed.

❖ What is the main difference between interface and abstract class?

- The interface is a fully un-implemented class used for declaring a set of operations of an object. The abstract class is a partially implemented class. It implements some of the operations of an object. These implemented operations are common for all next level subclasses. Remaining operations are implemented by the next level subclasses according to their requirement.
  - The interface allows us to develop multiple inheritances. So we must start object design with interface whereas abstract class does not support multiple inheritances so it always comes next to interface in object creation process.

❖ Rules to follow while working with the interface:

- The interface cannot have concrete methods, violation leads to CE: interface methods cannot have a body.

- We cannot declare interface members as private or protected members violation leads to CE:" modifier is not allowed here".
- An interface cannot be instantiated but its reference variable can be created for storing its subclass object reference.
- We cannot declare the interface as sealed it leads to CE: "illegal combination of modifier interface and final".
- The class derived from interface should implement all abstract methods of interface otherwise it should be declared as abstract else it leads to compile time error.
- The subclass should implement the interface method with public keyword because interface methods default accessibility modifier is public.
- In an interface, we cannot create fields variable violation leads to compile time error.

❖ Abstract Class in C#:

- A class that is declared by using the keyword abstract is called an abstract class. An abstract class is a partially implemented class used for developing some of the operations of an object which are common for all next level sub classes. So it contains both abstract methods, concrete methods including variables, properties, and indexers.
- It is always created as super class next to interface in object inheritance hierarchy for implementing common operations from an interface. An abstract class may or may not have abstract methods. But if a class contains an abstract method then it must be declared as abstract.
- An abstract class cannot be instantiated directly. It's compulsory to create/derive a subclass from the abstract class in order to provide functionality to its abstract functions.
- An abstract class is never usable to itself, because we cannot create the object of an abstract class. The members of an abstract class can be consumed only by the child class of the abstract class.

❖ Abstract Method:

- A method that does not have a body is called an abstract method.
- It is declared with the modifier abstract. It contains only Declaration/signature and does not contain implementation/body/definition of the method.
- An abstract function should be terminated with a semicolon. Overriding of an abstract function is compulsory.

❖ Points to Remember while working with abstract class in C#:

- An abstract class can contain both abstract methods and non-abstract (concrete) methods.
- It can contain both static and instance variables.
- The abstract class cannot be instantiated but its reference can be created.
- If any class contains abstract methods then it must be declared by using the keyword abstract.
- An abstract class can contain sealed methods but an abstract method or class cannot be declared as sealed.
- A subclass of an abstract class can only be instantiated if it implements all of the abstract methods of its super class. Such classes are called concrete classes to differentiate them from abstract classes.

❖ When should a class be declared as abstract?

- If the class has any abstract methods.
- If it does not provide implementation to any of the abstract methods it inherited.
- If it does not provide implementation to any of the methods of an interface.

❖ When to use the abstract method?

- Abstract methods are usually declared where two or more sub classes are expected to fulfil a similar role in different manner.
- Often the sub classes are required to fulfil an interface, so the abstract super class might provide several of the interface methods, but leave the sub classes to implement their own variations of the abstract methods.

❖ Why abstract class cannot be instantiated?

- Because it is not fully implemented the class as its abstract methods cannot be executed.
- If compiler allows us to create the object for an abstract class we can invoke the abstract method using that object which cannot be executed by CLR at runtime.
- Hence to restrict calling abstract methods, the compiler does not allow us to instantiate an abstract class.

❖ Differences between overriding methods and abstract methods in C#?

- The concept of abstract method is near similar to the concept of method overriding because in method overriding if a Parent class contains any virtual methods in it, then those methods can be re-implemented under the child class by using the override modifier.
- In a similar way, if a parent class contains any abstract methods in it, those abstract methods must be implemented under the child class by using the same override modifier.
- The main difference between method overriding and abstract method is in case of method overriding the child class re-implementing the method is optional but in case of the abstract method, the child class implementing the method is mandatory.

❖ Polymorphism in C#:

- The word polymorphism is derived from the Greek word, where Poly means many and morph means faces/ behaviours.
- Technically we can say that when a function shows different behaviours when we passed different types and number values, then it is called polymorphism.
- There are two types of polymorphism in C#
  - Static polymorphism / compile-time polymorphism / Early binding
  - Dynamic polymorphism / Run-time polymorphism / Late binding
- The polymorphism in C# can be implemented using the following three ways.
  - Overloading
  - Overriding
  - Hiding

❖ What is Compile-Time Polymorphism?

- In the case of compile-time polymorphism, the object of class recognizes which method to be executed for a particular method call at the time of program compilation and binds the method call with method definition.
- This happens in case of overloading because in case of overloading each method will have a different signature and based on the method call we can easily recognize the method which matches the method signature.
- It is also called as static polymorphism or early binding. Static polymorphism is achieved by using function overloading and operator overloading.

❖ What is Runtime Polymorphism?

- In the case of Runtime Polymorphism for a given method call, we can recognize which method has to be executed exactly at runtime but not in compilation time because in case of overriding we have multiple methods with the same signature.
- So, which method to be given preference and executed that is identified at Runtime and binds the method call with its suitable method. It is also called as dynamic polymorphism or late binding. Dynamic Polymorphism is achieved by using function overriding.

❖ How can we override a parent class method under child class?

- If you want to override a parent class method in its child class, first the method in the parent class must be declared as virtual by the using the keyword virtual, then only the child classes get the permission for overriding that method.
- Declaring the method as virtual is marking the method is overridable. If the child class wants to override the parent class virtual method then the child class can do it with the help of the override modifier. But overriding the method under child class is not mandatory for the child classes.
- Syntax:

  Class1:

  Public virtual void show(){} //virtual function (overridable)

  Class2: Class1

  Public override void show(){} //overriding

❖ Method Overloading:

- It is a process of creating multiple methods in a class with the same name but with a different signature.
- It is also possible to overload the methods in the derived classes, it means, it allows us to create a method in the derived class with the same name as the method name defined in the base class.
- The point that you need to keep in mind is the signature of a method does not include the return type and the params modifiers. So it is not possible to overload a method just based on the return type and params modifier.

- If two methods have the same method name those methods are considered as overloaded methods.
- Then the rule we should check is both methods must have different parameter types/number/order.
- But there is no rule on return type, non-accessibility modifier and accessibility modifier means overloading methods can have its own return type, non-accessibility modifier, and accessibility modifier because overloading methods are different methods.

❖ Method Overriding:

- The process of re-implementing the super class non-static method in the subclass with the same prototype (same signature defined in the super class) is called Function Overriding or Method Overriding in C#. The implementation of the sub-class overrides (i.e. replaces) the implementation of super class methods.
- The point that you need to keep in mind is the overriding method are always going to be executed from the current class object. Super class method is called the overridden method and sub-class method is called as the overriding method.
- If the super class method logic is not fulfilling the sub-class business requirements, then the subclass needs to override that method with required business logic. Usually, in most of the real-time applications, the super class methods are implemented with generic logic which is common for all the next level sub-classes.
- If a method in sub-class contains the same signature as the super class non-private non-static method, then the sub-class method is treated as the overriding method and the super class method is treated as the overridden method.

❖ Method Hiding:

- When we use the new keyword to hide a base class member, then it is called as Method Hiding in C#. We will get a compiler warning if we miss the new keyword. This is also used for re-implementing a parent class method under child class. Re-implementing parent class methods under child classes can be done using two different approaches, such as
  - **Method overriding**
  - **Method hiding**
- In the first case, we re-implement the parent class methods under child classes with the permission of parent class because here in parent class the method is declared as virtual giving the permission to the child classes for overriding the methods.
- In the 2nd approach, we re-implement the method of parent class even if those methods are not declared as virtual that is without parent permission we are re-implementing the methods.
- Syntax:

```
class1:
public void display()

class2 : class1
public new void display()
```

❖ Partial Classes:

- Partial Classes are the new feature that has been added in C# 2.0 which allows us to define a class on multiple files i.e. we can physically split the content of the class into different files but even physically they are divided but logically it is one single unit only.
- A class which code can be written in two or more files is known as partial class. To make any class as partial we need to use the keyword partial.
- Partial classes allow us to split a class definition into 2 or more files. It is also possible to split the definition of a struct or an interface over two or more source files. Each source file will contain a section of the class definition, and all parts are combined into a single class when the application is compiled.
- There are several situations when splitting a class definition is desirable:
  - When working on large projects, splitting a class over separate files allows multiple programmers to work on it simultaneously.
  - When working with automatically generated source code, the code can be added to the class without having to recreate the source file. Visual Studio uses this approach when creating windows form, web service wrapper code and so on.
- All the parts spread across different class files, must use the partial keyword. Otherwise, a compiler error is raised. Missing partial modifier. Another partial declaration of this type exists.
- All the parts spread across different files, must have the same access modifiers. Otherwise, a compiler error is raised. Partial declarations have conflicting accessibility modifiers.
- If any of the parts are declared as abstract, then the entire type is considered as abstract or if any of the parts are declared as sealed, then the entire type is considered as sealed or if any of the parts inherit a class, then the entire type inherits that class.
- C# does not support multiple class inheritance. Different parts of the partial class must not specify different base classes. The following code will raise a compiler error stating – Partial declarations must not specify different base classes.
- Different parts of the partial class can specify different base interfaces and the final type implements all of the interfaces listed by all of the partial declarations.

❖ Partial Methods:

- A partial class or a struct can contain partial methods.
- A partial method is created using the same partial keyword.
- Notice that the PartialMethod() definition has the partial keyword and does not have a body(implementation) only the signature.
- The implementation of a partial method is optional. If we don't provide the implementation, the compiler removes the signature and all calls to the method.
- The implementation can be provided in the same physical file or in another physical file that contains the partial class.
- Partial methods are private by default and it is a compile-time error to include any access modifiers, including private. The following code will raise an error stating – A partial method cannot have access modifiers or the virtual, abstract, override, new, sealed, or extern modifiers.
- It is a compile-time error to include declaration and implementation at the same time for a partial method. The code below produces a compile-time error – No defining declaration found for implementing declaration of partial method 'PartialDemo.PartialClass.partialMethod()'.
- A partial method return type must be void. Including any other return type is a compile-time error – Partial methods must have a void return type.

- A partial method must be declared within a partial class or partial struct. A non-partial class or struct cannot include partial methods.
- Signature of the partial method declaration must match with the signature of the implementation.
- A partial method can be implemented only once. Trying to implement a partial method more than once raises a compile-time error – A partial method may not have multiple implementing declarations.

❖ Sealed Class:

- A class from which it is not possible to create/derive a new class is known as sealed class.
- In simple word, we can also define the class that is declared using the sealed modifier is known as the sealed class and a sealed class cannot be inherited by any other class.
- A sealed class is completely opposite to an abstract class.
- This sealed class cannot contain abstract methods.
- It should be the bottom most class within the inheritance hierarchy.
- A sealed class can never be used as a base class.
- This sealed class is specially used to avoid further inheritance.
- The keyword sealed can be used with classes, instance methods, and properties.
- In the below situations we must define the class as sealed
  - If we don't want to override all the methods of our class in sub-classes.
  - If we don't want to extend our class functionality.

❖ What is the difference between private and sealed method?

- The private method is not inherited whereas sealed method is inherited but cannot be overridden.
- So, a private method cannot be called from sub-classes whereas sealed method can be called from sub-classes. The same private method can be defined in sub-class and it does not lead to error.

❖ Extension Methods:

- It is a new feature that has been added in C# 3.0 which allows us to add new methods into a class without editing the source code of the class i.e. if a class consists of set of members in it and in the future if you want to add new methods into the class, you can add those methods without making any changes to the source code of the class.
- Extension methods can be used as an approach of extending the functionality of a class in the future if the source code of the class is not available or we don't have any permission in making changes to the class.
- Before extension methods, inheritance is an approach that used for the extending the functionality of a class i.e. if we want to add any new members into an existing class without making a modification to the class, we will define a child class to that existing class and then we add new members in the child class.
- In case of an extension method, we will extend the functionality of a class by defining the methods, we want to add into the class in a new class and then bind them to an existing class.

- Both these approaches can be used for extending the functionalities of an existing class whereas, in inheritance, we call the method defined in the old and new classes by using object of the new class whereas, in case of extension methods, we call the old and new methods by using object of the old class.
- Extension methods must be defined only under the static class.
- As an extension method is defined under a static class, compulsory that method should be defined as static whereas once the method is bound with another class, the method changes into non-static.
- The first parameter of an extension method is known as the binding parameter which should be the name of the class to which the method has to be bound and the binding parameter should be prefixed with this keyword.
- An extension method can have only one binding parameter and that should be defined in the first place of the parameter list.
- If required, an extension method can be defined with normal parameter also starting from the second place of the parameter list.

==========

C# 7.X new Features

❖ Out Variables:

- In C#, we generally use the out parameter to pass a method argument's reference.
- If you want to use an out parameter, then you need to explicitly specify the out keyword in both the calling method and method definition.
- Before C# 7, we need to split their declaration and usage into two parts i.e. first we need to declare a variable and then we need to pass that variable to the method using the out keyword.
- The Out Parameter never carries value into the method definition. So, it is not required to initialize the out parameter while declaring.
- If you want to ignore an out parameter then you need to use a wildcard called underscore ('_') as the name of the parameter.

❖ Digit Separators in C#:

- In reality, it's very difficult to read a very large number. To overcome this problem, C# 7 comes with a new feature called digit separators "_".
- Now, it is possible to use one or more Underscore (_) character as digit separators. Sometimes, it is required when we are going to represent a very big number.
- The separators make no difference in the value as we can see in the above output. You can place them wherever you like in the number, and in any quantity. And in case you're wondering, you're not restricted to using them with integers only; they also work with the other numeric types as well.
- It is also not mandatory to use a single underscore as a separator even though you can also use multiple separators.

❖ Tuples:

- If you want to return more than one value from a method then you need to use Tuples.

- It is a very common thing to return multiple values from a method.
- Tuples in C# 7, provides a better mechanism to return multiple values from a method.

❖ What are the different ways to return more than one value from a method?

- Using **Custom DataType:** You can return multiple values from a method by using a custom data type (i.e. class) as the return type of the method. But sometimes we don't need or don't want to use classes and objects because that's just too much for the given purpose.
- Using **Ref and Out variable**: You can also return more than one value from the method either by using the "out" or "ref" parameters. Using "out" or "ref" parameters quite difficult to understand and moreover, the "out" and "ref" parameters will not work with the async methods.
- Using **dynamic keyword:** You can also return multiple values from a method by using the dynamic keyword as the return type. The dynamic keyword was introduced in C# 4. But from a performance point of view, we probably don't want to use dynamic.

==========

Exception Handling in C#

- When we write and execute our code in the .NET framework then there is a possibility of two types of error occurrences they are:
  - Compilation errors
  - Runtime errors
- An error that occurs in a program at the time of compilation is known as compilation error (compile-time error). These errors occur due to the syntactical mistakes under the program.
- The errors which occurred at the time of program execution are called as the runtime error. These errors occurred when we entering wrong data into a variable, trying to open a file for which there is no permission, trying to connect to the database with wrong user id and password, the wrong implementation of logic, missing of required resources.
- Runtime errors are dangerous because whenever they occur in the program, the program terminates abnormally on the same line where the error gets occurred without executing the next line of code.
- A runtime error is known as an exception. The exception will cause the abnormal termination of the program execution. So these errors (exceptions) are very dangerous because whenever the exception occurs in the programs, the program gets terminated abnormally on the same line where the error gets occurred without executing the next line of code.
- These exception classes are predefined under BCL (Base Class Libraries) where a separate class is provided for each and every different type of exception like
  - IndexOutOfRangeException
  - FormatException
  - NullReferenceException
  - DivideByZeroException
  - FileNotFoundException
  - SQLException,
  - OverFlowException, etc.

- All the exception classes are responsible for abnormal termination of the program as well as after abnormal termination of the program they will be displaying an error message which specifies the reason for abnormal termination i.e. they provide an error message specific to that error.
- he process of catching the exception for converting the CLR given exception message to end-user understandable message or for stopping the abnormal termination of the program whenever runtime errors are occurring is called Exception Handling in C#.
- Once we handle an exception under a program we will be getting following advantages
- We can stop the abnormal termination
- We can perform any corrective action that may resolve the problem occurring due to abnormal termination.
- Displaying a user-friendly error message, so that the client can resolve the problem provided if it is under his control.
- Keywords: **try:**
  - The try keyword establishes a block in which we need to write the exception causing and its related statements. That means exception causing statements must be placed in the try block so that we can handle and catch that exception for stopping abnormal termination and to display end-user understandable messages.
- **catch:**
  - The catch block is used to catch the exception that is thrown from its corresponding try block. It has the logic to take necessary actions on that caught exception.
  - Catch block syntax is look like a constructor. It does not take accessibility modifier, normal modifier, return type. It takes the only single parameter of type Exception.
  - Inside catch block, we can write any statement which is legal in .NET including raising an exception.
- **finally:**
  - The keyword finally establishes a block that definitely executes statements placed in it.
  - Statements that are placed in finally block are always executed irrespective of the way the control is coming out from the try block either by completing normally or throwing an exception by catching or not catching.
- Once we use the try and catch blocks in our code the execution takes place as following
- If all the statements under try block are executed successfully, from the last statement of the try block, the control directly jumps to the first statement that is present after the catch block (after all catch blocks) without executing catch block (it means there is no runtime error in the code.
- Then if any of the statements in the try block causes an error, from that statement without executing any other statements in the try block, the control directly jumps to the catch blocks searching for a catch block to handle that exception.
- If a proper catch block is available that handles the exception, then the abnormal termination stops there, executes the code under the catch block and from there again it jumps to the first statement after all the catch blocks.
- Then if a matching catch is not available abnormal termination occurs again.

❖ Multiple Catch Blocks:

- When we implement multiple catch blocks in C#, then at any given point of time only one catch block going to be executed and other catch blocks will be ignored.

- We can catch all exceptions with a single catch block with parameter "Exception". We need to use this catch block only for stopping the abnormal termination irrespective of the exceptions thrown from its corresponding try block.
- It is always recommended to write a catch block with the exception parameter even though we are writing multiple catch blocks. It acts as a backup catch block.
- We need to write multiple catch blocks in C# for a single try block because of the following reasons
  - To print message specific to an exception or
  - To execute some logic specific to an exception

❖ Finally block in C#:

- The statements which are placed in the finally block are always executed irrespective of the way the control is coming out from the try block either by completing normally or throwing the exception by catching or not catching.
- As per the industry coding standard, within the "finally" block we need to write the resource releasing logic or clean up the code. Resource releasing logic means un-referencing objects that are created in the try block. Since the statements written in the try and catch block are not guaranteed to be executed we must place them in finally block.
- For example, if we want to close ADO.NET objects such as Connection object, Command object, etc. we must call the close() method in both the try as well as in the catch block to guarantee its execution.
- Instead of placing the same close() method call statements in multiple places if we can write it in the finally block which will be always executed irrespective of the exception raised or not raised.

❖ In how many ways we can use try-catch and finally block in C#?

- **Try and catch:** In this case, the exception will be handled and stop the abnormal termination.

- **Try, catch and finally:** In this case, the exception will be handled and stopping the abnormal termination along with the statements that are placed within the "finally" block gets executed at any cost.

- **Try and finally:** In this case, abnormal will not stop when a runtime error occurs because exceptions are not handled but even if an abnormal termination occurs also finally blocks get executed.

❖ Custom Exceptions in C#:

- If none of the already existing .NET exception classes serve our purpose then we need to go for custom exceptions in C#.
- We know that an exception is a class. So to create a Custom exception, create a class that derives from System.Exception class. As a convention, end the class name with Exception suffix. All .NET exceptions end with, exception suffix. If you don't do so, you won't get a compiler error, but you will be deviating from the guidelines for creating custom exceptions.

❖ Inner Exception in C#:

- The Inner Exception in C# is a property of an exception class. When there is a series of exceptions, then the most current exception obtains the previous exception details in the InnerException property.
- In order words, we can say that the InnerException property returns the original exception that caused the current exception.
- Sometimes, as a programmer, we are using exception handling mechanisms to implement programming logic which is bad, and this is called as **Exception Handling Abuse**.

==========

Delegates in C#

- What if we want to pass a function as a parameter? How does C# handles the callback functions or event handler? The answer is - **Delegate**.
- The delegates in C# are the **Type-Safe Function Pointer**. It means they hold the reference of a method or function and then calls that method for execution.
- we can call a method that is defined in a class in two ways as follows:
  01. We can call the method using the object of the class if it is a non-static method or we can call the method through class name if it is a static method.
  02. We can also call a method in C# by using delegates. Calling a C# method using delegate will be faster in execution as compared to the first process i.e. either by using an object or by using the class name.
- If you want to invoke or call a method using delegates then you need to follow three simple steps. The steps are as follows.
  - Defining a delegate
  - Instantiating a delegate
  - Invoking a delegate
- The point that you need to remember while working with C# Delegates is, the signature of the delegate and the method it points should be the same. So, when you create a delegate, then the access modifier, return type, number of arguments and their data types of the delegates must and should be same as the access modifier, return type, number of arguments and the data types of the function that the delegate wants to refer.
- You can define the delegates either within a class or just like other types we defined under a namespace.
- To consume the delegate you must first create an object of the delegate and while creating the object the method you want to execute using the delegate should be passed as a parameter.
- Generally we create the objects under the Main() method of a class. So while binding the method with delegate if the method is non-static refer to it as the object of the class and if it is static refer to it with the name of the class.
- Now call the delegate by supplying the required values to the parameters so that the methods get executed internally which is bound with the delegates.
- Delegate is used to declare an event and anonymous methods in C#.
- If a multicast delegate returns a value then it returns the value from the last assigned target method.

❖ Rules of using Delegates in C#:

- A delegate in C# is a user-defined type and hence before invoking a method using delegate, we must have to define that delegate first.
- The signature of the delegate must match the signature of the method, the delegate points to otherwise we will get a compiler error. This is the reason why delegates are called as type-safe function pointers.
- A Delegate is similar to a class. Means we can create an instance of a delegate and when we do so, we need to pass the method name as a parameter to the delegate constructor, and it is the function the delegate will point to
- Tip to remember delegate syntax: Delegates syntax look very much similar to a method with a delegate keyword.
- The Delegates in C# are classified into two types such as
  - Single cast delegate
  - Multicast delegate
- If a delegate is used for invoking a single method then it is called a single cast delegate or unicast delegate. In other words, we can say that the delegates that represent only a single function are known as single cast delegate.
- If a delegate is used for invoking multiple methods then it is known as the multicast delegate. Or the delegates that represent more than one function are called Multicast delegate.

❖ Multicast Delegate:

- A Multicast Delegate is a delegate that holds the references of more than one function.
- When we invoke the multicast delegate, then all the functions which are referenced by the delegate are going to be invoked.
- If you want to call multiple methods using a delegate then all the method signature should be the same.
- A multicast delegate invokes the methods in the invocation list, in the same order in which they are added.
- If the delegate has a return type other than void and if the delegate is a multicast delegate.
- Only the value of the last invoked method will be returned. Along the same lines, if the delegate has an out parameter, the value of the output parameter will be the value assigned by the last method.

❖ Anonymous Method in C#:

- As the name suggests, an anonymous method in C# is a method without having a name.
- The Anonymous methods in C# can be defined using the keyword delegate and can be assigned to a variable of the delegate type.
- It is useful when the user wants to create an inline method and also wants to pass parameter in the anonymous method like other methods.

❖ What are the Advantages of Using Anonymous Methods in C#?

- Lesser typing word. Generally, anonymous methods are suggested when the code volume is very less.

❖ What are the Limitations of Anonymous Methods in C#?

- An anonymous method in C# cannot contain any jump statement like goto, break or continue.
- It cannot access the ref or out parameter of an outer method.
- The Anonymous methods cannot have or access the unsafe code.

❖ Points to Remember while working with the Anonymous Methods in C#:

- The anonymous methods can be defined using the delegate keyword.
- An anonymous method must be assigned to a delegate type.
- This method can access outer variables or functions.
- An anonymous method can be passed as a parameter.
- This method can be used as event handlers.
- In anonymous methods, you are allowed to remove parameter-list, which means you can convert an anonymous method into a delegate.
- You can also use an anonymous method as an event handler.

❖ Generic Delegates in C#:

- The Generic Delegates in C# were introduced as part of .NET Framework 3.5 which doesn't require defining the delegate instance in order to invoke the methods.
- C# provides three built-in generic delegates, they are:
  - Func
  - Action
  - Predicate

❖ What is Func Generic Delegate in C#?

- The Func Generic Delegate in C# is present in the System namespace. This delegate takes one or more input parameters and returns one out parameter. The last parameter is considered as the return value.
- The Func Generic Delegate in C# can take up to 16 input parameters of different types. It must have one return type. The return type is mandatory but the input parameter is not.
- Whenever your delegate returns some value, whether by taking any input parameter or not, you need to use the Func Generic delegate in C#.

❖ What is Action Generic Delegate in C#?

- The Action Generic Delegate in C# is also present in the System namespace. It takes one or more input parameters and returns nothing.
- This delegate can take a maximum of 16 input parameters of the different or same type.
- Your delegate does not return any value, whether by taking any input parameter or not, then you need to use the Action Generic delegate in C#.

❖ What is Predicate Generic Delegate in C#?

- The Predicate Generic Delegate in C# is also present in the System namespace. This delegate is used to verify certain criteria of the method and returns the output as Boolean, either true or false.
- It takes one input parameter and always returns a Boolean value which is mandatory. This delegate can take a maximum of 1 input parameter and always return the value of the Boolean type.
- Whenever your delegate returns a Boolean value, by taking one input parameter, then you need to use the Predicate Generic delegate in C#.

❖ Points to remember while working with C# Generic Delegates:

- Func, Action, and Predicate are generic inbuilt delegates that are present in the System namespace which is introduced in C# 3.
- All these three delegates can be used with the method, anonymous method, and lambda expressions.
- The Func delegates can contain a maximum of 16 input parameters and must have one return type.
- Action delegate can contain a maximum of 16 input parameters and does not have any return type.
- The Predicate delegate should satisfy some criteria of a method and must have one input parameter and one Boolean return type either true or false which is by default. Means we should not have to pass that to the Predicate.

==========

Collection Framework

❖ Arrays:

- An array as a collection of similar type of values which are stored in sequential order i.e. they stored in a contiguous memory location.
- C# supports 2 types of arrays. They are as follows:
  - Single dimensional array
  - Multi-dimensional array
- Again the multi-dimensional arrays are of two types:
  - Jagged array: whose rows and columns are not equal
  - Rectangular array: whose rows and columns are equal
- We can access the values of an array using the index positions.
- The Arrays in C# are reference types that are derived from the System.Array class.
- The Array class is a predefined class that is defined inside the System namespaces.
- This class is working as the base class for all the arrays in C#.
- The Array class provides a set of members (methods and properties) to work with the arrays such as creating, manipulating, searching and sorting the elements of an array.

❖ What is the difference between for loop and for each loop in C# to access collection values?

- In the case of for loop in C#, the loop variable refers to the index of an array whereas, in the case of a for-each loop, the loop variable refers to the values of the array.

- Irrespective of the values stored in the array, the loop variable must be of type int in case of for loop. The reason for this is, here the loop variable is referring to the index position of the array. Coming to the for-each loop, the data type of the loop variable must be the same as the type of the values stored in the array. For example, if you have a string array then the loop variable must be of type string in case of the for-each loop in C#.
- The most important point that you need to keep in mind is that the for loop in C# can be used both for accessing values from an array as well as assigning values to an array whereas the for-each loop in C# can only be used for accessing the values from an array but not for assigning values into an array.

❖ Advantages of using an Array in C#:

- It is used to represent similar types of multiple data items using a single name.
- We can use arrays to implement other data structures such as linked lists, trees, graphs, stacks, queues, etc.
- The two-dimensional arrays in C# are used to represent matrices.
- The Arrays in C# are strongly typed. That means they are used to store similar types of multiple data items using a single name. As the arrays are strongly typed so we are getting two advantages. First, the performance of the application will be much better because boxing and unboxing will not happen. Secondly, runtime errors will be prevented because of a type mismatch. In this case, at compile time it will give you the error if there is a type mismatch.

❖ Disadvantages of using arrays in C#:

- The array size is fixed. So, we should know in advance how many elements are going to be stored in the array. Once the array is created, then we can never increase the size of an array. If you want then we can do it manually by creating a new array and copying the old array elements into the new array.
- As the array size is fixed, if we allocate more memory than the requirement then the extra memory will be wasted. On the other hand, if we allocate less memory than the requirement, then it will create the problem.
- As the elements of an array are stored in contiguous memory locations. So insertions and deletions of array elements are very difficult and also time-consuming. So the time complexity increase in insertion and deletion operation.
- We can never insert an element into the middle of an array. It is also not possible to delete or remove elements from the middle of an array.

❖ ArrayList:

- The ArrayList in C# is a collection class that works like an array but provides the facilities such as dynamic resizing, adding and deleting elements from the middle of a collection.
- It implements the System.Collections.IList interface using an array whose size is dynamically increased as required.
- The following are the methods and properties provided by the ArrayList collection class in C#:
  - **Add(object value):** This method is used to add an object to the end of the collection.
  - **Remove(object obj):** This method is used to remove the first occurrence of a specific object from the collection.

- **RemoveAt(int index):** This method takes the index position of the elements and remove that element from the collection.
- **Insert(int index, Object value):** This method is used to inserts an element into the collection at the specified index.
- **Capacity:** This property gives you the capacity of the collection means how many elements you can insert into the collection

- Advantages of ArrayList over Array:
  - Variable Length
  - Can insert into the middle of the collection
  - Can insert into the middle of the collection

❖ Hashtable:

- The Hashtable in C# is a collection that stores the element in the form of "Key-Value pairs".
- The data in the Hashtable are organized based on the hash code of the key.
- The key in the HashTable is defined by us and more importantly, that key can be of any data type.
- Once we created the Hashtable collection, then we can access the elements by using the keys.
- The Hashtable class comes under the System.Collections namespace.

❖ Stack:

- The Stack in C# is a non-generic collection class that works in the LIFO (Last In First Out) principle.
- So, we need to use the Stack Collection in C#, when we need last in first out access to the items of a collection.
- That means the item which is added last will be removed first.
- When we add an item into the stack, then it is called as pushing an item.
- Similarly when we remove an item from the stack then it is called popping an item.
- The Stack class belongs to the System.Collections namespace.
- The last plate which is added on the stack will be the first one to remove from the stack. It is not possible to remove a plate from the middle of the stack.
- The Stack Collection in C# allows both null and duplicate values.
- Push(): The push() method is used to Inserts an object on top of the Stack.
- Pop(): The pop() method is used to remove and return the object at the top of the Stack.
- Peek(): The peek() method is used to return the object from the top of the Stack without removing it.

❖ Queue:

- The Queue in C# is a non-generic collection class that works in the FIFO (First In First Out) principle.
- So, we need to use the Queue Collection in C#, when we need the first in first out access to the items of a collection.
- That means the item which is added first will be removed first from the collection.
- When we add an item into the queue collection, it is called as enqueuing an item.

- Similarly, when we remove an item from the queue collection then it is called dequeuing an item.
- The Queue class belongs to the System.Collections namespace.
- The non-generic Queue Collection class in C# allows both null and duplicate values.
- **Enqueue():** This method is used to add an item (or object) to the end of the Queue.
- **Dequeue():** The Dequeue() method of the Queue class is used to Remove and return the object from the beginning of the Queue.
- **Dequeue():** The Dequeue() method of the Queue class is used to Remove and return the object from the beginning of the Queue.

❖ Dictionary in C#:

- The Dictionary in C# is a Collection class same as HashTable i.e. used to store the data in the form of Key Value Pairs.
- But here while creating the dictionary object you need to specify the type for the keys as well as the type for values also.
- Points to Remember while working with Dictionary Generic Collection:
  - A dictionary is a collection of (key, value) pairs.
  - The Dictionary Generic Collection class is present in System.Collections.Generic namespace.
  - When creating a dictionary, we need to specify the type for the key and as well as for the value.
  - The fastest way to find a value in a dictionary is by using the keys.
  - Keys in a dictionary must be unique.

❖ Problems with non-generic Collection Classes in C#:

- Non-generic collection classes are not type-safe as they operate on object data type so they can store any type of value.
  - Array is type-safe.
  - Array List, HashTable, Stack, and Queue are not type-safe.
- So, the solution is Generic collections in C#.
  - Array: Type-safe but fixed length.
  - Collections: Auto Resizing but not type-safe.
  - Generic Collections: Typesafe and auto-resizing.

❖ Generics in C#:

- The Generics in C# allows us to define classes and methods which are decoupled from the data type.
- In other words, we can say that the Generics allow us to create classes using angular brackets for the data type of its members.
- At compilation time, these angular brackets are going to be replaced with some specific data types.
- Generics can be applied to the following:
  - Interface, Abstract class, Class, Method, Static method, Property, Event, Delegates, Operator
- **Advantages of Generics:**

- It Increases the reusability of the code.
- The Generics are type-safe. We will get the compile-time error if we try to use a different type of data rather than the one, we specified in the definition.
- We get better performance with Generics as it removes the possibilities of boxing and unboxing.

==========

Multithreading in C#

❖ How the operating system executes multiple applications at a time?

- To execute all the applications, the operating system internally makes use of processes.
- A process is a part of the operating system (or a component under the operating system) which is responsible for executing the program or application.
- So, to execute each and every program or application, there will be a process.
- Windows operating system is a multitasking operating system. It means it has the ability to run multiple applications at the same time
- So, we have an operating system and under the operating system, we have processes that running our applications. So under the process, an application runs. To run the code of an application the process will make use of a concept called Thread.

❖ What is Thread?

- Generally, a Thread is a lightweight process.
- In simple words, we can say that a Thread is a unit of a process that is responsible for executing the application code.
- So, every program or application has some logic or code and to execute that logic or code, Thread comes into the picture.
- By default, every process has at least one thread which is responsible for executing the application code and that thread is called as Main Thread.
- So, every application by default is a single-threaded application.
- All the threading related classes in C# belong to the System.Threading namespace.

❖ Join() Method of Thread class

- The Join method of Thread class in C# blocks the current thread and makes it wait until the child thread on which the Join method invoked completes its execution.
- There are three overloaded versions available for the Join Method in Thread class.

```
public void Join();

public bool Join(int millisecondsTimeout);

public bool Join(TimeSpan timeout);
```

- The first version of the Join method which does not take any parameter will block the calling thread (i.e. the Parent thread) until the thread (child thread) completes its execution. In this case, the calling thread is going to wait for indefinitely time until the thread on which the Join Method invoked is completed.
- The second version of the Join Method allows us to specify the time out. It means it will block the calling thread until the child thread terminates or the specified time elapses. This overloaded takes the time in milliseconds. This method returns true if the thread has terminated and returns false if the thread has not terminated after the amount of time specified by the millisecondsTimeout parameter has elapsed.
- The third overloaded version of this method is the same as the second overloaded version. The only difference is that here we need to use the TimeSpan to set the amount of time to wait for the thread to terminate.

❖ IsAlive Method of Thread Class:

- The IsAlive() method of Thread class returns true if the thread is still executing else returns false.

❖ Protecting Shared Resource in Multithreading Using Locking:

- In a multithreading application, it is very important for us to handle multiple threads for executing critical section code.
- if we have a shared resource and if multiple threads want to access the shared resource then we need to protect the shared resource from concurrent access otherwise we will get some inconsistency output.
- In C#, we can use **lock** and **Monitor** to provide thread safety in a multithreaded application.
- Both lock and monitor provides a mechanism which ensures that only one thread is executing the critical section code at any given point of time to avoid any functional breaking of code.
- The section or block or particular resource that you want to protect should be placed inside the lock block.

```
lock (_lockObject)
{
   Count++;
}
```

❖ Monitor class:

- We can also use Monitor class to protect shared resources in a multi-threaded environment.
- This can be done by acquiring an exclusive lock on the object so that only one thread can enter into the critical section at any given point of time.
- The Monitor is a static class and belongs to the System.Threading namespace.
- As a static class, it provides a collection of static methods.
- Using these static methods you can provide access to the monitor associated with a particular object.
- The lock is the shortcut for Monitor.Enter with try and finally. So, the lock provides the basic functionality to acquire an exclusive lock on a synchronized object. But, If you want more control

to implement advanced multithreading solutions using TryEnter() Wait(), Pulse(), and PulseAll() methods, then the Monitor class is your option.

❖ Mutex in C#:

- Mutex also works likes a lock i.e. acquired an exclusive lock on a shared resource from concurrent access, but it works across multiple processes.
- Exclusive locking is basically used to ensure that at any given point of time, only one thread can enter into the critical section.
- The Mutex class provides the WaitOne() method which we need to call to lock the resource and similarly it provides ReleaseMutex() which is used to unlock the resource.
- Mutex can only be released from the same thread which obtained it.

❖ Semaphore in C#:

- The Semaphore in C# is used to limit the number of threads that can have access to a shared resource concurrently.
- In other words, we can say that Semaphore allows one or more threads to enter into the critical section and execute the task concurrently with thread safety.
- So, in real-time, we need to use Semaphore when we have a limited number of resources and we want to limit the number of threads that can use it.
- The Semaphores are Int32 variables that are stored in operating system resources. When we initialize the semaphore object we initialize it with a number. This number basically used to limits the threads that can enter into the critical section.
- So, when a thread enters into the critical section, it decreases the value of the Int32 variable with 1 and when a thread exits from the critical section, it then increases the value of the Int32 variable with 1.
- The most important point that you need to remember is when the value of the Int32 variable is 0, then no thread can enter into the critical section.
- **Syntax:**

```
Semaphore semaphore = new Semaphore(2, 3);
```

- The InitialCount parameter sets the value for the Int32 variable.
- That defines the initial number of requests for the semaphore that can be granted concurrently.
- MaximumCount parameter defines the maximum number of requests for the semaphore that can be granted concurrently.
- The second parameter always must be equal or greater than the first parameter otherwise we will get an exception.

❖ Deadlock in C#:

- A deadlock in C# is a situation where two or more threads are unmoving or frozen in their execution because they are waiting for each other to finish.

❖ Thread Pool in C#:

- Thread pool in C# is nothing but a collection of threads that can be reused to perform no of tasks in the background.
- Now when a request comes, then it directly goes to the thread pool and checks whether there are any free threads available or not.
- If available, then it takes the thread object from the thread pool and executes the task.
- Once the thread completes its task then it again sent back to the thread pool so that it can reuse. This reusability avoids an application to create the number of threads and this enables less memory consumption.
- Thread pool gives better performance as compared to the Thread class object.

==========

Asynchronous Programming in C#

❖ Task-based Asynchronous Programming:

- A task in C# is used to implement Task-based Asynchronous Programming and was introduced with the .NET Framework 4. The Task object is typically executed asynchronously on a thread pool thread rather than synchronously on the main thread of the application.
- A task scheduler is responsible for starting the Task and also responsible for managing it. By default, the Task scheduler uses threads from the thread pool to execute the Task.
- Tasks in C# basically used to make your application more responsive. If the thread that manages the user interface offloads the works to other threads from the thread pool, then it can keep processing user events which will ensure that the application can still be used.

==========

AutoMapper, Indexer and Enums

❖ What is AutoMapper in C#?

- The AutoMapper in C# is a mapper between two objects. That is AutoMapper is an object-object mapper. It maps the properties of two different objects by transforming the input object of one type to the output object of another type.
- The AutoMapper is an open-source library present in GitHub.

❖ How to make Optional Parameters in C#:

- We can make the method parameters as optional in C# in many different ways as follows.
  - Using parameter array
  - Method overloading
  - Specify parameter defaults
  - Using OptionalAttribute

❖ Indexers in C#:

- The Indexers are the members of a class and if we define indexers in a class then the class behaves like a virtual array. So it's a member of a class which gives access to the values of a class just like an array.
- We cannot apply indexing directly to a class. We can do indexing on an array but we cannot do the same thing with a user-defined class like Employee. An array is a predefined class and all the logic's are implemented in that class for indexing so that we can access those using indexes. But Employee is a user-defined class and we have not implemented any logic to access the class like an array.
- If you want to access the class like an array then first you need to define an indexer in the class. Once you define an indexer in the class then you can start accessing the values of the class by using the index position.
- Syntax:

```
[<modifiers>] <type> this [int index or string name]
{
        [get {<statements>}]    //Get Accessor
        [set {<statements>}]    //Set Accessor
}
```

- The "this" keyword telling that we are defining an indexer on the current class.
- The int index or string name is used to specify whether you want to access the values by using its integer index position or by using string name.
- The get accessor is used for returning the value and the set accessor is used for assigning the value.
- But in real-time, we may have more number of properties and it's very difficult to access the values by using the index position.
- So in such cases, in most of the time we need to access the values by using the property name. To do so we need to use string accessor.

❖ Enums in C#:

- The Enums are strongly typed names constants.
- Points to Remember about C# Enums:
  - The Enums are enumerations.
  - Enums are strongly typed named constants. Hence, an explicit cast is needed to convert from the enum type to an integral type and vice versa. Also, an enum of one type cannot be implicitly assigned to an enum of another type even though the underlying value of their members is the same.
  - The default underlying type of an enum is int.
  - The default value for the first element of the enum is ZERO and gets incremented by 1.
  - It is also possible to customize the underlying type and values of enums.
  - The Enums are value types.
  - Enum keyword (all small letters) is used to create the enumerations, whereas the Enum class, contains static GetValues() and GetNames() methods which can be used to list Enum underlying type values and Names.

- The Enums are like classes, so we cannot define two members with the same name.