**.NET Core Overview:** .NET Core is a **new version of .NET Framework,** which is a **free, open-source, general-purpose development** platform maintained by **Microsoft**. It is a **cross-platform framework** that runs on Windows, macOS, and Linux operating systems. l**imitations** with the .NET Framework. For example, it **only runs on the Windows** platform.

The .NET Core Framework composed of the following parts:

**CLI Tool**s: A set of tooling for development and deployment. is a cross-platform toolchain for developing, building, running, and publishing .NET applications.

**Roslyn**: Language compiler for C# and Visual Basic

**CoreFX**: Set of framework libraries.

**CoreCLR**: A JIT based CLR (Command Language Runtime).

**.NET CLR**: manages the execution of programs written in any language that uses the .NET Framework, for example C#, VB.Net, F# and so on. Programmers write code in any language, including VB.Net, C# and F# yhen they compile their programs into an intermediate form of code called CLI in a portable execution file (PE) that can be managed and used by the CLR and then the CLR converts it into machine code to be will executed by the processor. The information about the environment, programming language, its version and what class libraries will be used for this code are stored in the form of metadata with the compiler that tells the CLR how to handle this code.

**Create and publish a NuGet package using Visual Studio:** NuGet package is a single ZIP file with the .nupkg extension that contains compiled code (DLLs),

To install learning API : click Solution -> Manage Nuget packages -> right Settings -> select local machine -> give the path for the folder for LearningApi. -> Install

To Create Nuget package : right click on project -> pack

Pack : dotnet pack - Packs the code into a NuGet package. command builds the project and creates NuGet packages.

**XML documentation comments**: the c# compiler will find all comment fields with XML tags in the source code and create an XML documentation file from those comments. Other tools can process that XML output to create human-readable documentation in the form of web pages or PDF files, for example.

**Namespaces:** NET uses namespaces to organize its many classes. declaring your own namespaces can help you control the scope of class and method names in larger programming projects.

**S.O.L.I.D. Principles: 5 design principles** of [object-oriented programs](https://www.educative.io/blog/object-oriented-programming) (OOP) that result in readable, adaptable, and scalable code. 5 principles of SOLID are:

**Single-responsibility principle**:  states that each class, module, or function in your program should only do one job. makes it easy to follow another well-respected principle of OOP, encapsulation. It is easy to hide data from the user when all data and methods for a job are within the same single-responsibility class.

**Open-closed principle**: “Software entities … should be open for extension, but closed for modification.” (OCP) calls for entities (class/function/module) that can be widely adapted but also remain unchanged. This leads us to create duplicate entities with specialized behavior through polymorphism. Through polymorphism, we can extend our parent entity to suit the needs of the child entity while leaving the parent intact. it minimizes program risk when you add new uses for an entity.

**Liskov substitution principle**: any class must be directly replaceable by any of its subclasses without error. each subclass must maintain all behavior from the base class along with any new behaviors unique to the subclass. he child class must be able to process all the same requests and complete all the same tasks as its parent class.  LSP involve polymorphism to create class-specific behavior for the same calls. For example: A abstract class fruit is a base class, any subclass(Apple, Orange, Banaba) can use the functionality of the fruit class also can add their own functionality

**Interface segregation principle**: Many client-specific interfaces are better than one general-purpose interface.” The advantage of ISP is that it splits large methods into smaller, more specific methods.This makes the program easier to debug for three reasons: less code carried between classes. Less code means fewer bugs.

A single method is responsible for a smaller variety of behaviors. If there is a problem with a behavior, you only need to look over the smaller methods.

If a general method with multiple behaviors is passed to a class that doesn’t support all behaviors (such as calling for a property that the class doesn’t have), there will be a bug if the class tries to use the unsupported behavior.

**Dependency inversion principle** dependency inversion principle (DIP) has two parts:

High-level modules should not depend on low-level modules. Instead, both should depend on abstractions (interfaces)

Abstractions should not depend on details. Details (like concrete implementations) should depend on abstractions.

**type safety in C# .net:** Type safety in .NET has been introduced to prevent the objects of one type from peeking into the memory assigned for the other object. type safety check happens both at Compile time and at runtime. The compiler will do the compile-time type safety check and CLR will do the runtime safety check. If Employee and Student are two incompatible class. We cannot assign an object of employee class to Student class variable.

**Scrum**

Scrum one of the Agile frameworks that helps people, teams and organizations followed to complete challenging projects

Standards of Scrum: • Scrum consists of 3 roles: product owner, ScrumMaster, and development team/engineering team. • Scrum uses fixed-length iterations called sprints ranging from one Week four weeks (or 30 days long). • Scrum teams should consist of 7 +/- 2 people. • Sprint cannot exceed more than 30 days. • Sprint is time bound. • Scrum team aims to build a potentially shippable product by end of each sprint. • Daily Scrum / Stand-up meeting should be only for 15 mins.

The Sprint: They are fixed length events of one month or less. A new Sprint starts immediately after the conclusion of the previous Sprint.

Daily Scrum: The purpose of the Daily Scrum is to inspect progress toward the Sprint Goal .

Scrum Team: consists of one Scrum Master, one Product Owner, and Developers. typically 10 or fewer people.

Product owner: •Responsible person for release burn-down chart. • Responsible for the product vision. • Person who maintains the product backlog • Only person who can make a decision whether the product is ready to ship. • Person to constantly reprioritize the product backlog.

ScrumMaster • Facilitator of the Scrum process. • Helps resolving any issues / impediments faced by the Scrum team. • Shields the team from external interferences and distractions. • Creates a positive environment for the team to be self-organizing. • Person who maintains sprint burn-down / burn-up charts. • Person who facilitates the required meetings

Development Team • Should be a self-organizing and cross-functional team (consists of the members with testing, development, business analyst, domain expertise, etc., skills). • Works with product owner in reprioritizing the product backlog items. • Consists of 7+/- 2 members. • Responsible to complete the committed task for the sprint.

**Sprint Phases Product Backlog** is the list of functionalities with the short descriptions derived from the requirements of the project

**Sprint planning** meeting is attended by the product owner, ScrumMaster, and the development team. what should be the duration of the sprint and into how many sprints the project be divided into to address the requirements in the product backlog.

**Sprint Backlog**: is the list of the product backlog item for which the Development team commits to deliver that particular sprint.

**Sprint Velocity** is the capability of the development team to deliver the number of user stories based on the estimated story points.

**Sprint** Defined period of time in which the specific committed work has to be completed.

**Daily Scrum** :on each day of the sprint, the team holds a meeting called the Daily Scrum. Usually this meeting is held at the same location and at the same time every day. This is a strictly timeboxed meeting of 15 minutes. In this meeting the following 3 things will be discussed/ answered by each team member. a. What I did yesterday. b. What I am going to do today c. Impediments I may have. All team members are required to attend, and the product owner and ScrumMaster are expected to attend the meeting.

**Creating a Class Object:** Whenever the compiler comes across the keyword new, it realizes that an object is being created and allocates a separate memory location for the object being created.

**static class**: is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated. In other words, you cannot use the new operator to create a variable of the class type you access the members of a static class by using the class name itself.

In classes, interfaces, and structs, you may add the static modifier to fields, methods, properties, operators, events, and constructors. A static member can't be referenced through an instance. It's referenced through the type name. example: public class MyBaseC

{

public struct MyStruct

{

public static int x = 100;

}

}

To refer to the static member x, use the fully qualified name, MyBaseC.MyStruct.x, unless the member is accessible from the same scope:

Console.WriteLine(MyBaseC.MyStruct.x);

While an instance of a class contains a separate copy of all instance fields of the class, there's only one copy of each static field.

It isn't possible to use this to reference static methods or property accessors.

If the static keyword is applied to a class, all the members of the class must be static.

A DatabaseConfig class can be a static class that may have members such as database name, server name, port number, and even a connection string. We know that these values will not change for objects.

**Structs** are light versions of classes. Structs are value types.

-Structs are value types while classes are reference types. --Structs can be instantiated without using a new operator. -A struct cannot inherit from another struct or class, and it cannot be the base of a class. All structs inherit directly from System.ValueType, which inherits from System.Object. Structs are value types while classes are reference types. -Structs can be instantiated without using a new operator. -A struct cannot inherit from another struct or class, and it cannot be the base of a class. All structs inherit directly from System.ValueType, which inherits from System.Object.

There are **two kinds of types** in C#: **reference types and value types**. Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations on one variable to affect the other (except in the case of in, ref and out parameter variables; see in, ref and out parameter modifier).

**Lambda expressions**: to create an anonymous function. To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

Example: Func<int, int> square = x => x \* x; Console.WriteLine(square(5)); // Output: 25

**Delegates:** A delegate is a type that safely encapsulates a method, similar to a function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type safe, and secure. A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate its instance with any method with a compatible signature and return type. You can invoke (or call) the method through the delegate instance. Delegates are used to pass methods as arguments to other methods. // Create a method for a delegate. public static void DelegateMethod(string message) { Console.WriteLine(message); } / Instantiate the delegate. Del handler = DelegateMethod; // Call the delegate. handler("Hello World"); public static void MethodWithCallback(int param1, int param2, Del callback) { callback("The number is: " + (param1 + param2).ToString()); } MethodWithCallback(1, 2, handler); Output: The number is: 3

**Language Integrated Query (LINQ):** "language-integrated" part of LINQ is the query expression. Query expressions are written in a declarative query syntax. By using query syntax, you can perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same basic query expression patterns to query and transform data in SQL databases, ADO .NET Datasets, XML documents and streams, and .NET collections. class LINQQueryExpressions { static void Main() { // Specify the data source. int[] scores = new int[] { 97, 92, 81, 60 }; // Define the query expression. IEnumerable<int> scoreQuery = from score in scores where score > 80 select score; // Execute the query. foreach (int i in scoreQuery) { Console.Write(i + " "); } } }// Output: 97 92 81

**interface (C**# Reference): An interface defines a contract. Any [class](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/class) or [struct](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct) that implements that contract must provide an implementation of the members defined in the interface.

**Local functions:** C# 7.0,supports local functions. Local functions are private methods of a type that are nested in another member. They can only be called from their containing member. Local functions can be declared in and called from:Methods, especially iterator methods and async methods,Constructors.

**Statement keywords**: Statements are program instructions.

Category C# keywords

Selection statements if, switch

Iteration statements do, for, foreach, while

Jump statements break, continue, goto, return, yield

Exception handling statements throw, try-catch, try-finally, try-catch-finally

Checked and unchecked checked, unchecked

fixed statement fixed

lock statement lock

**try-catch**: The try block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully. When an exception is thrown, the common language runtime (CLR) looks for the catch statement that handles this exception. If the currently executing method does not contain such a catch block, the CLR looks at the method that called the current method, and so on up the call stack. If no catch block is found, then the CLR displays an unhandled exception message to the user and stops execution of the program.

**try-catch-finally**: A common usage of catch and finally together is to obtain and use resources in a try block, deal with exceptional circumstances in a catch block, and release the resources in the finally block.

**Generic type:** Generics introduces the concept of type parameters to .NET, which make it possible to design classes and methods that defer one or more types until the class or method is declared and instantiated by client code.

**Member Overloading:** Member overloading means creating two or more members on the same type that differ only in the number or type of parameters but have the same name. Method overloading is an example of the polymorphism feature of an object oriented programming language. Example:

public class Methodoveloading

{

public int add(int a, int b) //two int type Parameters method

{

return a + b;

}

public int add(int a, int b,int c) //three int type Parameters with same method same as above

{

return a + b+c;

}

public float add(float a, float b,float c,float d) //four float type Parameters with same method same as above two method

{

return a + b+c+d;

}

**Method** **Overriding** means having two methods with same name and same signatures [parameters], one should be in the base class and other method should be in a derived class [child class]. You can override the functionality of a base class method to create a same name method with same signature in a derived class. Method **Overriding** is an example of the polymorphism feature. You can achieve method overriding using inheritance. Virtual and Override keywords are used to achieve method overriding. The override modifier is required to extend or modify the **abstract or virtual** implementation of an inherited method, property, indexer, or event.Points to be remembered,

Method cannot be private.

Only abstract or virtual method can be overridden.

Which method should be called is decided at run time.

**Abstract:** The abstract modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the abstract modifier in a class declaration to indicate that a class is intended only to be a base class of other classes, not instantiated on its own. Members marked as abstract must be implemented by non-abstract classes that derive from the abstract class. It is not possible to modify an abstract class with the sealed modifier because the two modifiers have opposite meanings.

**Sealed:** When applied to a class, the sealed modifier prevents other classes from inheriting from it.

**Virtual:** The virtual keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. When a virtual method is invoked, the run-time type of the object is checked for an overriding member. By default, methods are non-virtual. You cannot override a non-virtual method.You cannot use the virtual modifier with the static, abstract, private, or override modifiers.

C# **indexers** are usually known as smart arrays. A C# indexer is a class property that allows you to access a member variable of a class or struct using the features of an array. Indexers are implemented through get and set accessors for the [ ] operator.

**Abstract vs Virtual vs Interface**

Abstract methods do not provide an implementation and they force the derived classes to override the method. It is declared under abstract class. An abstract method only has the method definition

Virtual methods have an implementation, unlike the Abstract method and it can exist in the abstract and non-abstract class. It provides the derived classes with the option of overriding it Like to add something.

virtual or abstract members of a class cannot be declared as private.

An abstract class doesn't provide full abstraction but an interface does provide full abstraction; i.e. both a declaration and a definition is given in an abstract class but not so in an interface.

Note: Abstruct class and abstruct methods are different. In abstruct class, there are implementation of methods and those method can be inherited in derived class. But in Abstract method should modify the method by overriding.

**Code snippet:**

public abstract class EntityBase

{

public abstract override string ToString();

public abstract override bool Equals(object obj);

}

public class Customer : EntityBase

{

//Implementation code for the abstract methods

}

public class Base

{

public virtual void Test()

{

Console.WriteLine("This is the base version of the virtual method");

}

}

public class Derived : Base

{

public override void Test()

{

Console.WriteLine("This is the derived version of the virtual method");

interface MyInterface

{

void myMethod();

}

class MyClass : MyInterface

{

public static void Main()

{

MyClass cls = new MyClass();

cls.myMethod();

}

public void myMethod()

{

Console.WriteLine("welcome to MCN IT SOLUTION");

abstract class M1

{

public int add(int a, int b)

{

return (a + b);

}

}

class M2 :M1

{

public int mul(int a, int b)

{

return a \* b;

}

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

**Ref Keyword:**

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

public static string GetNextName(ref int id)

{

string returnText = "Next-" + id.ToString();

id += 1;

return returnText;

}

static void Main(string[] args)

{

int i = 1;

Console.WriteLine("Previous value of integer i:" + i.ToString());

string test = GetNextName(ref i);

Console.WriteLine("Current value of integer i:" + i.ToString());

}}// output: Previous value of integer i: 1

Current value of integer i:2

**Out Keyword**: The out keyword passes arguments by reference. This is very similar to the ref keyword.

1.Need not to be assigned before going into the function

2.Must be assigned before it comes out of the function

public static string GetNextNameByOut(out int id)

{

id = 1;

string returnText = "Next-" + id.ToString();

return returnText;

}

static void Main(string[] args)

{

int i = 0;

Console.WriteLine("Previous value of integer i:" + i.ToString());

string test = GetNextNameByOut(out i);

Console.WriteLine("Current value of integer i:" + i.ToString());

}// output: Previous value of integer i: 0

Current value of integer i:1

**C# Enumerations Type – Enum**: In C#, an enum (or enumeration type) is used to assign constant names to a group of numeric integer values. All the constant names can be declared inside the curly brackets and separated by a comma. If values are not assigned to enum members, then the compiler will assign integer values to each member starting with zero by default. enum WeekDays

{

Monday,

Tuesday,

Wednesday,

Thursday,

Friday,

Saturday,

Sunday

}

Console.WriteLine(WeekDays.Monday); // Monday

Console.WriteLine(WeekDays.Tuesday); // Tuesday

params :By using the params keyword, you can specify a method parameter that takes a variable number of arguments. The parameter type must be a single-dimensional array. No additional parameters are permitted after the params keyword in a method declaration, and only one params keyword is permitted in a method declaration. C# can be used to declare method that does not know the number of parameters.

For example, a Student class with a method, TotalMarks that returns the sum of all marks. Each grade may have a different number of subjects and their respective marks. The 3rd grade may have 3 subjects only. The 8th grade may have 4 subjects while a 9th grade may have 5 subjects. We can use params in this case and pass 3, 4, and 5 comma separated values.

**A thread** is defined as the execution path of a program. Each thread defines a unique flow of control. In C#, the System.Threading.Thread class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the main thread. a thread is a unit which executes the code under the program. So every program has logic and a thread is responsible for executing this logic. Every program by default carries one thread to executes the logic of the program and the thread is known as the Main Thread, so every program or application is by default single-threaded model. This single-threaded model has a drawback. The single thread runs all the process present in the program in synchronizing manner, means one after another. So, the second process waits until the first process completes its execution, it consumes more time in processing.

The life cycle of a thread starts when an object of the System.Threading.Thread class is created and ends when the thread is terminated or completes execution. various states in the life cycle of a thread −

The Unstarted State − It is the situation when the instance of the thread is created but the Start method is not called.

The Ready State − It is the situation when the thread is ready to run and waiting CPU cycle.

The Not Runnable State − A thread is not executable, when

Sleep method has been called

Wait method has been called

Blocked by I/O operations

The Dead State − It is the situation when the thread completes execution or is aborted.

Multitasking is the simultaneous execution of multiple tasks or processes over a certain time interval. Windows operating system is an example of multitasking because it is capable of running more than one process at a time like running Google Chrome, Notepad, VLC player, etc. at the same time.

Example of multithreading: For example, once process collects data in a thread, one process filters the data, and one process matches the data against a database. Each of these scenarios are common uses for multithreading and will significantly improve performance of similar applications running in a single thread.

**Lazy Loading In C# 4.0**: Lazy initialization of an object means that its creation is deferred until it is first used. Lazy

initialization is primarily used to improve performance, avoid wasteful computation, and reduce program memory requirements. When you have an object that is expensive to create, and the program might not use it. For example, assume that you have in memory a Customer object that has an Orders property that contains a large array of Order objects that, to be initialized, requires a database connection. If the user never asks to display the Orders or use the data in a computation, then there is no reason to use system memory or computing cycles to create it. By using Lazy<Orders> to declare the Orders object for lazy initialization, you can avoid wasting system resources when the object is not used. After the Lazy object is created, no instance of Orders is created until the Value property of the Lazy variable is accessed for the first time. On first access, the wrapped type is created and returned, and stored for any future access.

**Dependency injection in .NET:** The Dependency Injection Design Pattern in C# allows us to develop loosely coupled software components. In other words, we can say that Dependency Injection Design Pattern is used to reduce the tight coupling between the software components. As a result, we can easily manage future changes and other complexity in our application. Dependency Injection pattern involves 3 types of classes:

Client Class: The Client class (dependent class) is a class that depends on the service class.

Service Class: The Service class (dependency) is a class that provides service to the client class.

Injector Class: The Injector class injects the service class object into the client class.

**Design Patterns?** Design patterns are solutions to software design problems. Patterns are about reusable designs and interactions of objects. he C# source code for each pattern is provided in 2 forms: structural and real-world. Structural code uses type names as defined in the pattern definition and UML diagrams. Real-world code provides real-world programming situations where you may use these patterns.

**Singleton:** The Singleton design pattern ensures a class has only one instance and provide a global point of access to it.

**Abstract Factory**: The Abstract Factory design pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes.

**Factory Method**: The Factory Method design pattern defines an interface for creating an object, but let subclasses decide which class to instantiate. This pattern lets a class defer instantiation to subclasses.

**Observer:** The Observer design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.