LTI
Let's Solve

.NET Framework

A Larsen & Toubro Group Company

Introduction to .Net Framework

- What is the term “Framework” mean?
  - Let us understand with an example of building a
  - house.
  - In order to build a house, first a framework is built,
  - And everything we do, we do it inside that
  - framework only.
  - Like wise, Microsoft also provides a framework, and
  - any .Net based applications are developed within
  - this framework.

- We can define framework as “an arrangement in which software provides greater functionality that can be extended by which additional user written codes.
- Framework allow standard way of creating applications. It forces developers to work in a standard way.

Introduction to .Net Framework

- What is ".Net Framework"?
  - .NET Framework is a technology that supports building and running Windows apps and web services.

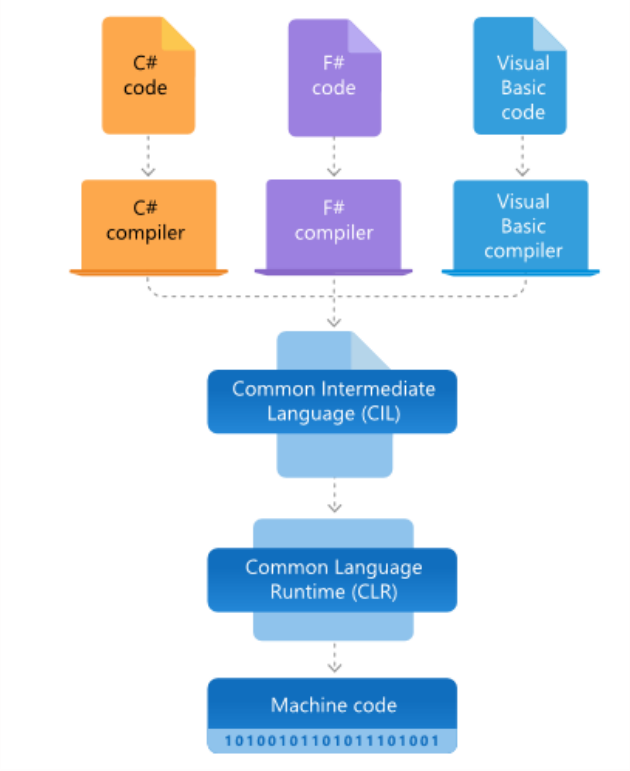
- .NET Framework is designed to fulfill the following objectives:
  - To provide a consistent object-oriented programming environment in which code can be executed locally or remotely.
  - To provide a code-execution environment that minimizes software deployment and versioning conflicts.
  - To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
  - To make the developer experience consistent across widely varying types of apps, such as Windows-based apps and Web-based apps.
  - To build all communication on industry standards to ensure that code based on .NET Framework integrates with any other code.

Introduction to .Net Framework

- .Net framework consists of:
  - **Common Language Runtime(CLR):**
    - CLR is a .Net runtime which manages code at execution time.
    - Providing core services such as memory management, thread management, and remoting.
    - Enforce strict type safety and other forms of code accuracy that promote security and robustness.
  - **.Net Framework Class Library, also called Base Class Library.**
    - The .NET Framework class library is a collection of reusable types that tightly integrate with the common language runtime.
    - The class library is object oriented, providing types from which your own managed code derives functionality.

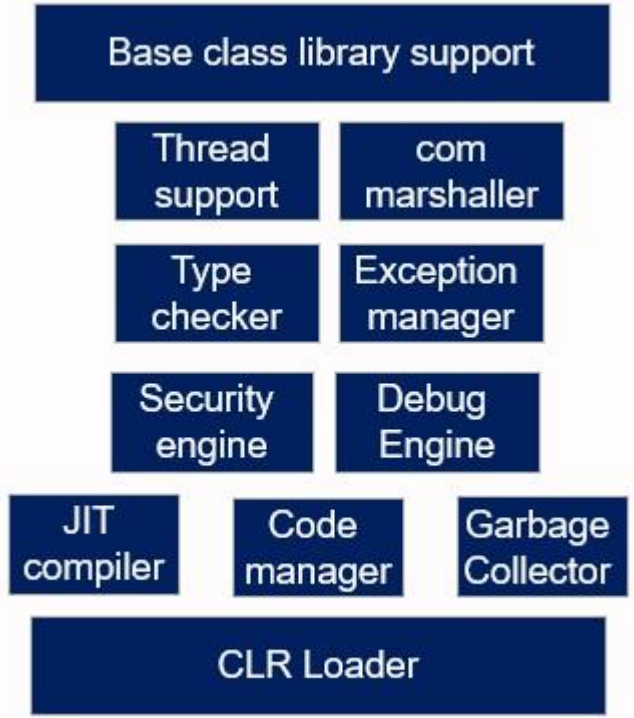
Introduction to .Net Framework

- **.Net Framework Architecture**
- .NET applications are written in the C#, F#, or Visual Basic programming language.
- Code is compiled into a language-agnostic Common Intermediate Language (CIL).
- Compiled code is stored in assemblies—files with a .dll or .exe file extension.
- When an app runs, the CLR takes the assembly and uses a just-in-time compiler (JIT) to turn it into machine code that can execute on the specific architecture of the computer it is running on.



Introduction to .Net Framework

- Components of CLR consists:
  - CLR consists of:
    - Common Type Specification(CTS)
    - Common language specification(CLS)
    - Garbage Collection
    - Just-In-Time Compiler



Introduction to .Net Framework

- **Common Type Specification(CTS):**
  - CTS describes the datatypes that can be used by managed code.
  - CTS defines how these types are declared, used and managed in the runtime.
  - It facilitates cross-language integration, type safety, and high-performance code execution.
  - The rules defined in CTS can be used to define your own classes and values.

For example, C# has an **int** data type and VB.NET has **Integer** data type. Hence a variable declared as an int in C# and Integer in VB.NET, finally after compilation, uses the same structure Int32 from CTS.

Introduction to .Net Framework

- **Common Language Specification(CLS):**
  - It defines a set of rules and restrictions that every language must follow which runs under the .NET framework.
  - The languages which follow these set of rules are said to be CLS Compliant.

For example, In C# every statement must have to end with a semicolon. it is also called a statement Terminator, but in VB.NET each statement should not end with a semicolon(;).

But CLR can understand all the language Syntax because in .NET each language is converted into MSIL code after compilation and the MSIL code is language specification of CLR.

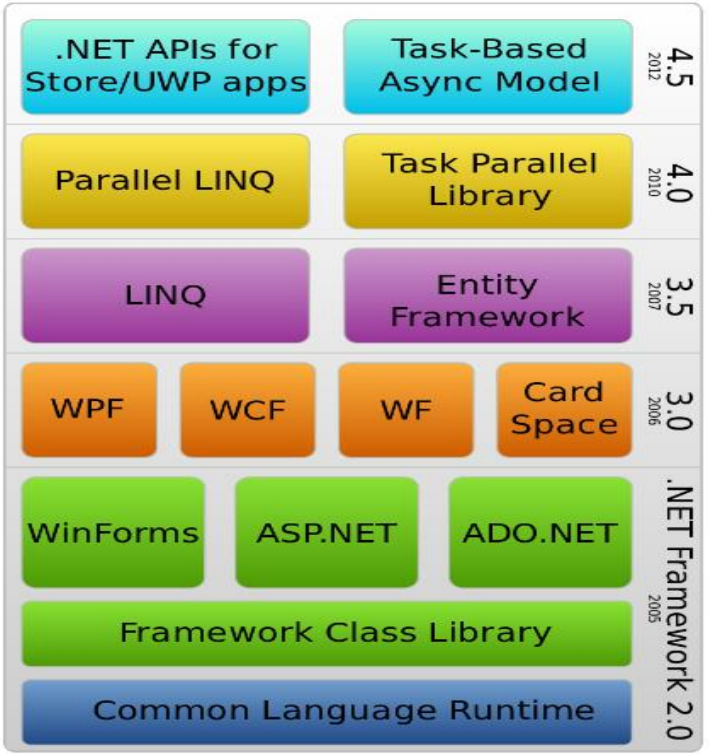
Introduction to .Net Framework

- **Garbage Collection:**
  - The garbage collector automatically releases the memory space after it is no longer required so that it can be reallocated.

- **JIT Compiler:**
  - The JIT compiler in the CLR converts the Microsoft Intermediate Language (MSIL) into the machine code that is specific to the computer environment that the JIT compiler runs on. The compiled MSIL is stored so that it is available for subsequent calls if required.

Introduction to .Net Framework

- .Net Framework Versions:
  - The first version of the .Net framework was released in the year 2002.
  - The version was called .Net framework 1.0.
  - The .Net framework has come a long way since then, and the current version is 4.7.1.



LTI
Let's Solve

.NET Framework

Quiz

Q: Which is an extensible set of classes used by any .NET compliant programming language?

- .NET class libraries
- Common Language Runtime
- Common Language Infrastructure
- Component Object Model

Q: Which of the following .NET components can be used to remove unused references from the managed heap?

- Common Language Infrastructure
- Common Language Runtime
- Garbage Collector
- Class Loader

LTI
Let's Solve

C# programming

A Larsen & Toubro Group Company

C# Programming

- What is C#?
  - It is an Object-Oriented and and type-safe programming language.
  - C# has its roots in the C family of languages and will be immediately familiar to C, C++, Java, and JavaScript programmers.
  - It can be used to create various types of applications, such as web, windows, console applications, or other types of applications
- Different Types of applications developed using C#.NET.
  - Windows applications
  - Web applications
  - Console applications
  - Class library

C# Versions:

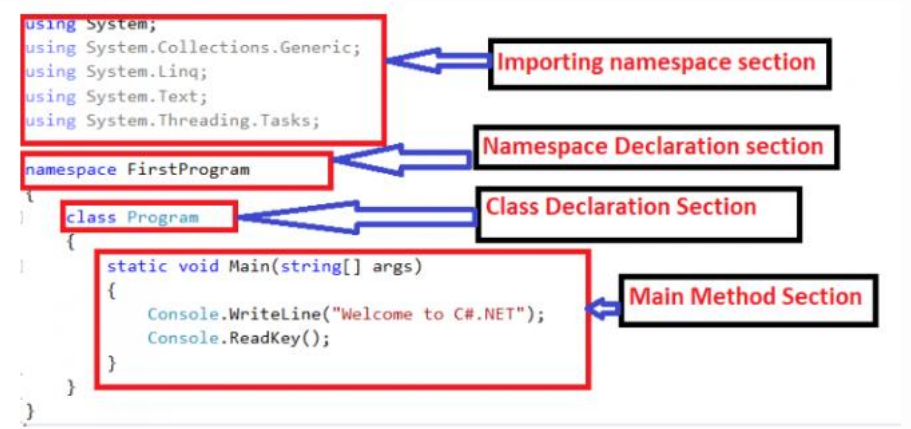
Version	.NET Framework	Visual Studio
C# 1.0	.NET Framework 1.0/1.1	Visual Studio .NET 2002
C# 2.0	.NET Framework 2.0	Visual Studio 2005
C# 3.0	.NET Framework 3.0\3.5	Visual Studio 2008
C# 4.0	.NET Framework 4.0	Visual Studio 2010
C# 5.0	.NET Framework 4.5	Visual Studio 2012/2013
C# 6.0	.NET Framework 4.6	Visual Studio 2013/2015
C# 7.0	.NET Core 2.0	Visual Studio 2017
C# 8.0	.NET Core 3.0	Visual Studio 2019

Tools for running C# programs:

- **Tools for running c# programs are:**
  - .Net Framework
  - Visual Studio Code
  - Visual Studio IDE (includes .Net Framework)

C# program structure:

- First Program:
  - **Namespace declaration:**
    - In .NET applications, all classes related to the project should be declared inside one namespace.
  - **Class declaration:**
    - This is a start-up class which contains Main method.
  - **Main method:**
    - The main() method is the starting execution point of the application. When the application is executed the main method will be executed first.



Data Types in C#:

- **Value types:**
  - **Simple Types**
    - Signed integral: sbyte, short, int, long
    - Unsigned integral: byte, ushort, uint, ulong
    - Unicode characters: char
    - IEEE binary floating-point: float, double
    - High-precision decimal floating-point: decimal
    - Boolean: bool
  - **Enum types:** User-defined types of the form enum E {...}
  - **Struct types:** User-defined types of the form struct S {...}
  - **Nullable value types:** Extensions of all other value types with a null value

Data Types in C#:

- **Reference types:**
  - **Class types**
    - Ultimate base class of all other types: object
    - Unicode strings: string
    - User-defined types of the form class C {...}
  - **Interface types**
    - User-defined types of the form interface I {...}
  - **Array types**
    - Single- and multi-dimensional, for example, int[] and int[,]
  - **Delegate types**
    - User-defined types of the form delegate int D(...)

Types and Variable declaration in C#:

- **Integral numeric types:** The *integral numeric types* represent integer numbers. All integral numeric types are value types.

```
int a = 123;
System.Int32 b = 123;
```

- **Floating-point numeric types:** The *floating-point numeric types* represent real numbers. All floating-point numeric types are value types.

```
double d = 0.42e2;
Console.WriteLine(d); // output 42;

float f = 134.45E-2f;
Console.WriteLine(f); // output: 1.3445

decimal m = 1.5E6m;
Console.WriteLine(m); // output: 1500000
```

Types and Variable declaration in C#:

- **bool type:**
  - The bool type keyword is an alias for the .NET System.Boolean structure type that represents a Boolean value, which can be either true or false.

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked"); // output: Checked
Console.WriteLine(false ? "Checked" : "Not checked"); // output: Not checked
```

- **char type:**
  - The char type keyword is an alias for the .NET System.Char structure type that represents a Unicode UTF-16 character.

```
char ch='A';
```

Types and Variable declaration in C#:

- **Enumeration type:**
  - An *enumeration type* (or *enum type*) is a value type defined by a set of named constants of the underlying integral numeric type.

```
enum Season
{
Spring,
Summer,
Autumn,
Winter
 }
```

Types and Variable declaration in C#:

- **Structure types:**
  - A structure type (or struct type) is a value type that can encapsulate data and related functionality. You use the struct keyword to define a structure type:

```
public struct Coords
{
public Coords(double x, double y)
{
X = x;
Y = y;
}
public double X { get; }
public double Y { get; }
public override string ToString() => $"({X}, {Y})";
}
```

Types and Variable declaration in C#:

- **Nullable value types:**
  - A nullable value type T? represents all values of its underlying value type T and an additional null value.
  - For example, you can assign any of the following three values to a bool? variable: true, false, or null.

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

Types and Variable declaration in C#:

- **Reference types:**
  - **void:** You use void as the return type of a method (or a local function) to specify that the method doesn't return a value.
  - **var:**
    - Beginning in Visual C# 3.0, variables that are declared at method scope can have an implicit "type" var.
    - An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type.

```
var i = 10; // Implicitly typed.
int i = 10; // Explicitly typed.
```

string[] words = { "apple", "strawberry", "grape", "peach", "banana" };

var wordQuery = from word in words where word[0] == 'g' select word;

Types and Variable declaration in C#:

- **Boxing:**
  - The process of Converting a **Value Type (char, int etc.) to a Reference Type(object)** is called **Boxing**.
  - Boxing is implicit conversion process in which object type (super type) is used.
  - The Value type is always stored in Stack. The Referenced Type is stored in Heap.

- **Unboxing:**
  - The process of converting **reference type into the value type** is known as **Unboxing**.
  - It is explicit conversion process.

```
int num = 23; // value type is int and assigned value 23
Object Obj = num; // Boxing
int i = (int)Obj; // Unboxing
```

Types and Variable declaration in C#:

- **Arrays:**
  - An array is a collection of same type of data.
  - An array has the following properties:
    - An array can be Single-Dimensional, Multidimensional or Jagged.
    - The number of dimensions and the length of each dimension are established when the array instance is created. These values can't be changed during the lifetime of the instance.
    - The default values of numeric array elements are set to zero, and reference elements are set to null.
    - Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
    - Array types are reference types derived from the abstract base type Array.
    - Since this type implements IEnumerable and IEnumerable<T>, you can use foreach iteration on all arrays in C#.

Types and Variable declaration in C#:

- Types of Arrays:
  - Single-Dimensional Arrays:

```
int[] array = new int[5];
string[] stringArray = new string[6];
int[] array1 = new int[] { 1, 3, 5, 7, 9 };
```

Multidimensional Arrays:

```
int[,] array = new int[4, 2];
int[, ,] array1 = new int[4, 2, 3];
int[,] array4 = { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Types and Variable declaration in C#:

- Jagged Arrays:
  - A jagged array is an arrays whose elements are arrays.
  - The elements of a jagged array can be of different dimensions and sizes.

```
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[5];
jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

- foreach: The foreach statement provides a simple, clean way to iterate through the elements of an array.

```
int[] numbers = { 4, 5, 6, 1, 2, 3, -2, -1, 0 };
foreach (int i in numbers)
{
System.Console.Write("{0} ", i);
}
```

Types and Variable declaration in C#:

- **class type:**
  - *Classes* are the most fundamental of C#'s types.
  - A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit.
  - A class provides a definition for dynamically created *instances* of the class, also known as *objects*.
  - Classes support *inheritance* and *polymorphism*, mechanisms whereby *derived classes* can extend and specialize *base classes*.

Types and Variable declaration in C#:

- **object type:**
  - The object type is an alias for System.Object in .NET.
  - In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from System.Object.
  - You can assign values of any type to variables of type object. Any object variable can be assigned to its default value using the literal null.

- **string type:**
  - The string type represents a sequence of zero or more Unicode characters.
  - string is an alias for System.String in .NET.
  - Although string is a reference type, the equality operators == and != are defined to compare the values of string objects, not references.
  - Strings are *immutable*--the contents of a string object cannot be changed after the object is created.

```
int i = 0; string s = "108"; bool result = int.TryParse(s, out i); // i now = 108
```

Types and Variable declaration in C#:

- dynamic type:
  - The dynamic type indicates that use of the variable and references to its members bypass compile-time type checking.
  - Instead, these operations are resolved at run time.
  - Type dynamic behaves like type object in most circumstances.
  - Any non-null expression can be converted to the dynamic type.
  - The dynamic type differs from object in that operations that contain expressions of type dynamic are not resolved or type checked by the compiler.
  - The compiler packages together information about the operation, and that information is later used to evaluate the operation at run time.
  - As part of the process, variables of type dynamic are compiled into variables of type object. Therefore, type dynamic exists only at compile time, not at run time.

Constants in C#

- **Constants:**
  - Constants are immutable values which are known at compile time and do not change for the life of the program.
  - Constants are declared with the const modifier.
  - Only the C# built-in types (excluding System.Object) may be declared as const.
  - User-defined types, including classes, structs, and arrays, cannot be const.
  - Use the readonly modifier to create a class, struct, or array that is initialized one time at runtime (for example in a constructor) and thereafter cannot be changed.
  - C# does not support const methods, properties, or events.
  - The static modifier is not allowed in a constant declaration.

```
public const double PI=3.14;
```

Constants in C#

- **readonly:** The “readonly” keyword is a modifier that can be used in four contexts:
  - **In a field declaration,** “readonly” indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class.
  - **In a “readonly struct type definition”,** “readonly” indicates that the structure type is immutable.
  - **In an instance member declaration within a structure type, “**readonly” indicates that an instance member doesn't modify the state of the structure.
  - **In a ref readonly method return,** the readonly modifier indicates that method returns a reference and writes aren't allowed to that reference.

```
class Age{
readonly int year;
Age(int year) {
this.year = year;
}
void ChangeYear() {
//year = 1967; // Compile error if
uncommented.
}
}
```

Difference between “const” and “readonly” in C#

- **const:**
  - A const field can only be initialized at the declaration of the field.
  - A const field can be initialized only once, hence cannot have different values.
  - A const field is a compile-time constant

- **readonly:**
  - A readonly field can be assigned multiple times in the field declaration and in any constructor.
  - A readonly fields can have different values depending on the constructor used.
  - A readonly field can be used for run-time constants

LTI
Let's Solve

Object-Oriented Programming in C#

A Larsen & Toubro Group Company

Functional Programming

- Issues in functional or procedural programming:
  - **Reusability:**
    - In Functional Programming, we need to write the same code or logic at multiple places which increases the code redundant.
    - Later if you want to change the logic, then you need to change at multiple places.
  - **Extensibility:**
    - It is not possible in functional programming to extend the features of a function.
    - You have to create a completely new function and then change the function as per your requirement.
  - **Simplicity:**
    - As extensibility and reusability are not possible in functional programming, usually we end up with lots of functions and lots of scattered code.
  - **Maintainability:**
    - As we don't have Reusability, Extensibility, and Simplicity in functional Programming, so it is very difficult to manage and maintain the application code.

What is an Object-Oriented Programming

- Object-Oriented Programming is a design approach where we think in terms of real-world objects rather than functions or methods.
- Programs are organized around objects and data rather than action and logic.

- **OOPs Principles:**
  - OOPs provide 4 principles. They are
    - Encapsulation
    - Inheritance
    - Polymorphism
    - Abstraction

What is an Object-Oriented Programming

- Object-Oriented Programming is a design approach where we think in terms of real-world objects rather than functions or methods.
- Programs are organized around objects and data rather than action and logic.

- **OOPs Principles:** OOPs provide 4 principles. They are
  - **Encapsulation:** The process of binding the data and functions together into a single unit (i.e. class) is called encapsulation.
  - **Inheritance:** *Inheritance* describes the ability to create new classes based on an existing class.
  - **Polymorphism:** *Polymorphism* means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways.
  - **Abstraction:** The process of representing the essential features without including the background details is called abstraction

Classes and Objects

- The terms *class* and *object* describe the *type* of objects, and the *instances* of classes, respectively.
- So, the act of creating an object is called *instantiation*.
- Using the blueprint analogy, a class is a blueprint, and an object is a building made from that blueprint.
  - To define a class:

```
class SampleClass
{
}
```

  - C# also provides types called *structures* that are useful when you don't need support for inheritance or polymorphism.
  - To define a structure:

```
struct SampleStruct
{
}
```

Let's Solve

Class Members

- **Class members:**
  - Each class can have different *class members* that include
    - **Properties** that describe class data,
    - **Methods** that define class behavior, and
    - **Events** that provide communication between different classes and objects.

- **Properties and fields**
  - Fields and properties represent information that an object contains. Fields are like variables because they can be read or set directly, subject to applicable access modifiers.
  - Properties have get and set accessors, which provide more control on how values are set or returned.

```
public class SampleClass{
    string sampleField;
    public int SampleProperty { get; set; }
}
```

Class Members

- **Methods:**
  - A method is a code block that contains a series of statements.
  - A program causes the statements to be executed by calling the method and specifying any required method arguments.

- **Method signatures:**
  - Methods are declared in a class, struct, or interface by specifying:
    - the access level such as public or private, optional modifiers such as abstract or sealed,
    - the return value,
    - the name of the method, and
    - any method parameters.

  <access-modifier> <optional-modifiers> <return-type> <method-name>(parameters-list)

- **Note:** A return type of a method is not part of the signature of the method for the purposes of method overloading.

Class Members

- **Static classes and members:**
  - **Static classes:**
    - A static class is basically the same as a non-static class, but there is one difference: a static class cannot be instantiated.
    - Because there is no instance variable, you access the members of a static class by using the class name itself.
    - Main features of a static class are as follows:
      - Contains only static members.
      - Cannot be instantiated.
      - Is sealed.
      - Cannot contain Instance Constructors.

Class Members

- Static classes and members:
  - Static members:
    - The static member is always accessed by the class name, not the instance name.
    - Only one copy of a static member exists, regardless of how many instances of the class are created.
    - A non-static class can contain static methods, fields, properties, or events.
    - Static methods and properties cannot access non-static fields and events in their containing type, and they cannot access an instance variable of any object unless it is explicitly passed in a method parameter.
    - Static methods can be overloaded but not overridden, because they belong to the class, and not to any instance of the class.
    - C# does not support static local variables (variables that are declared in method scope).

Class Members

- **Passing parameters to Methods:**
  - In C#, arguments can be passed to parameters either by value or by reference.
  - **Passing Value-Type Parameters**
    - Passing a value-type variable to a method by value means passing a copy of the variable to the method.
    - Any changes to the parameter that take place inside the method have no affect on the original data stored in the argument variable.
      - **Passing Value Types by Reference**
      - To pass by reference with the intent of avoiding copying but not changing the value, use the “in” modifier.
      - To pass a parameter by reference with the intent of changing the value, use the “ref”, or “out” keyword.

Class Members

- **Passing Reference-Type Parameters**
  - A variable of a reference type does not contain its data directly; it contains a reference to its data.
  - Hence, method parameters can be of following types:
  - **in** specifies that this parameter is passed by reference but is only read by the called method.
  - **out** specifies that this parameter is passed by reference and is written by the called method.
  - **params** specifies that this parameter may take a variable number of arguments.
  - **ref** specifies that this parameter is passed by reference and may be read or written by the called method.

Class Members

- **Access Modifiers:**
  - All types and type members have an accessibility level. The accessibility level controls whether they can be used from other code in your assembly or other assemblies.
  - Different types of access modifier available in C# are:
    - **public:** The type or member can be accessed by any other code in the same assembly or another assembly that references it.
    - **private:** The type or member can be accessed only by code in the same class or struct.
    - **protected:** The type or member can be accessed only by code in the same class, or in a class that is derived from that class.
    - **internal:** The type or member can be accessed by any code in the same assembly, but not from another assembly.
    - **protected internal:** The type or member can be accessed by any code in the assembly in which it's declared, or from within a derived class in another assembly.
    - **private protected:** The type or member can be accessed only within its declaring assembly, by code in the same class or in a type that is derived from that class.

Class Members

- **Default accessibility:**
- **Namespaces** implicitly have **public** declared accessibility. No access modifiers are allowed on namespace declarations.
- **Types** declared in compilation units or namespaces can have public or internal declared accessibility and default to **internal** declared accessibility.
- **Class** members can have any of the five kinds of declared accessibility and default to **private** declared accessibility.
- A **type** declared as a member of a **namespace** can have only **public** or **internal** declared accessibility.
- **Struct members** can have public, internal, or private declared accessibility and default to **private** declared accessibility because structs are implicitly sealed
- **Interface members** implicitly have **public** declared accessibility. No access modifiers are allowed on interface member declarations.
- **Enumeration members** implicitly have **public** declared accessibility. No access modifiers are allowed on enumeration member declarations.

Class Members

- **Constructor:**
  - A constructor is a method whose name is the same as the name of its type.
  - Its method signature includes only the method name and its parameter list; it does not include a return type.
  - Whenever a class or struct is created, its constructor is called.
  - A class or struct may have multiple constructors that take different arguments.
  - Constructors enable the programmer to set default values, limit instantiation, and write code that is flexible and easy to read.
  - **Types of constructors:**
    - Instance constructors
    - Static constructors

Class Members

- **Instance constructors:**
  - Instance constructors are used to create and initialize any instance member variables when you use the new expression to create an object of a class.
  - A constructor which takes no arguments, is called a *parameterless constructor*.
  - If a class does not have a constructor, a parameterless constructor is automatically generated and default values are used to initialize the object fields.

- **Private constructors:**
  - A private constructor is a special instance constructor.
  - It is generally used in classes that contain static members only.
  - Constructor are private by default.
  - Private constructors are used to prevent creating instances of a class when there are no instance fields or methods

Class Members

- **Static constructors:**
  - A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed once only.
  - It is called automatically before the first instance is created or any static members are referenced.
  - Static constructors have the following properties:
    - A static constructor does not take access modifiers or have parameters.
    - A class or struct can only have one static constructor.
    - Static constructors cannot be inherited or overloaded.
    - A static constructor cannot be called directly and is only meant to be called by the common language runtime (CLR). It is invoked automatically to initialize the class before the first instance is created or any static members are referenced.
    - If you don't provide a static constructor to initialize static fields, all static fields are initialized to their default value as listed in Default values of C# types.

Class Members

- Indexers:
  - Indexers allow instances of a class or struct to be indexed just like arrays.
  - The indexed value can be set or retrieved without explicitly specifying a type or instance member.
  - Indexers resemble properties except that their accessors take parameters.

```
class SampleCollection<T>
{
private T[] arr = new T[100];
// Define the indexer to allow client code to use [] notation.
public T this[int i] {
get { return arr[i]; }
set { arr[i] = value; }
}
}
```

Class Members

- Indexers:

Setting and getting Indexers:

```
class Program
{
static void Main()
{
var stringCollection = new SampleCollection<string>();
stringCollection[0] = "Hello, World";
Console.WriteLine(stringCollection[0]);
}
}
```

Inheritance

- *Inheritance* is one of the fundamental attributes of object-oriented programming.
- It allows you to define a child class that reuses (inherits), extends, or modifies the behavior of a parent class.
- The class whose members are inherited is called the *base class and t*he class that inherits the members of the base class is called the *derived class*.
- Inheritance is used to express an "is a" relationship between a base class and one or more derived classes.
- C# and .NET support *single inheritance* only.
- However, inheritance is transitive, which allows you to define an inheritance hierarchy for a set of types.
- Besides any types that they may inherit from through single inheritance, all types in the .NET type system implicitly inherit from Object or a type derived from it.

Inheritance

- A member's accessibility affects its visibility for derived classes as follows:
  - Private members are visible only in derived classes that are nested in their base class. Otherwise, they are not visible in derived classes.
  - Protected members are visible only in derived classes.
  - Internal members are visible only in derived classes that are located in the same assembly as the base class. They are not visible in derived classes located in a different assembly from the base class.
  - Public members are visible in derived classes and are part of the derived class' public interface. Public inherited members can be called just as if they are defined in the derived class.
- Derived classes can also **override** inherited members by providing an alternate implementation.
- In order to be able to override a member, the member in the base class must be marked with the **virtual** keyword.
- By default, base class members are not marked as virtual and cannot be overridden.

Inheritance

- **base keyword:**
  - The base keyword is used to access members of the base class from within a derived class:
    - Call a method on the base class that has been overridden by another method.
    - Specify which base-class constructor should be called when creating instances of the derived class.
- **new modifier:**
  - When used as a declaration modifier, the new keyword explicitly hides a member that is inherited from a base class.
  - When you hide an inherited member, the derived version of the member replaces the base class version.
  - Although you can hide members without using the new modifier, you get a compiler warning.
  - If you use new to explicitly hide a member, it suppresses this warning.
- **sealed:**
  - The sealed modifier prevents other classes from inheriting from it.

Polymorphism

- The word polymorphism is derived from the Greek word, where Poly means many and morph means faces/ behaviors. So, polymorphism means one thing having many (poly) form.
- When a function shows different behaviors when we passed different types and number values, then it is called polymorphism.
- **Types of Polymorphism in C#**
  - There are two types of polymorphism in C#
    - **Static polymorphism / compile-time polymorphism / Early binding**
    - **Dynamic polymorphism / Run-time polymorphism / Late binding**

Polymorphism

- **Static polymorphism:**
  - The object of class recognizes which method to be executed for a method call at the time of program compilation and binds the method call with method definition.
  - It is implemented using the concept of “function overloading” where each method will have a different signature.
  - Static polymorphism is achieved by using function overloading and operator overloading.

- **Runtime polymorphism:**
  - In 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.
  - Dynamic Polymorphism is achieved by using function overriding.

Polymorphism

- **Function overloading:**
  - It is a process of creating multiple methods in a class with the same name but with a different signature.
  - In C#, 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.
  - Function signature includes:
    - Number of parameters
    - Types of parameters, and
    - Sequence of parameters

Polymorphism

- **Function 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.
  - Super class method is called the overridden method and sub-class method is called as the overriding method.
  - 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.
  - 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.

Abstract Classes

- The abstract keyword enables you to create classes and class members that are incomplete and must be implemented in a derived class.
- Classes can be declared as abstract by putting the keyword abstract before the class definition.
- An abstract class cannot be instantiated.
- The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share.
- Abstract classes may also define abstract methods.
- This is accomplished by adding the keyword abstract before the return type of the method.

Interfaces

- An interface defines a contract.
- An interface contains definitions for a group of related functionalities that a non-abstract class or a struct must implement.
- Any class or struct that implements that contract must provide an implementation of the members defined in the interface.
- To implement an interface member, the corresponding member of the implementing class must be public, non-static, and have the same name and signature as the interface member.
- Interfaces can inherit from one or more interfaces. The derived interface inherits the members from its base interfaces. A class that implements a derived interface must implement all members in the derived interface, including all members of the derived interface's base interfaces.
- An interface may define static methods, which must have an implementation.
- Beginning with C# 8.0, an interface may define a default implementation for members.

Exception Handling

- A runtime error is known as an exception.
- The exception will cause the abnormal termination of the program execution.
- Exceptions can be generated by the common language runtime (CLR), by the .NET Framework or any third-party libraries, or by application code.
- Exception handling uses the try, catch, and finally keywords to try actions that may not succeed, to handle failures when you decide that it is reasonable to do so, and to clean up resources afterward.
- Exceptions can be generated by the common language runtime (CLR), by the .NET Framework or any third-party libraries, or by application code.
- Exceptions are created by using the throw keyword.

Exception Handling

- Exceptions have the following properties:
  - Exceptions are types that all ultimately derive from **System.Exception**.
  - Use a **try block** around the statements that might throw exceptions.
  - Once an exception occurs in the try block, the flow of control jumps to the first associated exception handler that is present anywhere in the call stack. In C#, the **catch keyword** is used to define an exception handler.
  - If no exception handler for a given exception is present, the program stops executing with an error message.
  - If a catch block defines an exception variable, you can use it to obtain more information about the type of exception that occurred.
  - Exceptions can be explicitly generated by a program by using the **throw** keyword.
  - Exception objects contain detailed information about the error, such as the state of the call stack and a text description of the error.
  - Code in a **finally block** is executed even if an exception is thrown. Use a finally block to release resources, for example to close any streams or files that were opened in the try block.

Exception Handling

- Some exceptions are thrown automatically by the .NET Framework's common language runtime (CLR) are as follows:
  - **ArithmeticException:** A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException.
  - **ArrayTypeMismatchException:** Thrown when an array cannot store a given element because the actual type of the element is incompatible with the actual type of the array.
  - **DivideByZeroException:** Thrown when an attempt is made to divide an integral value by zero.
  - **IndexOutOfRangeException:** Thrown when an attempt is made to index an array when the index is less than zero or outside the bounds of the array.
  - **InvalidCastException:** Thrown when an explicit conversion from a base type to an interface or to a derived type fails at runtime.
  - **NullReferenceException:** Thrown when an attempt is made to reference an object whose value is null.
  - **OutOfMemoryException:** Thrown when an attempt to allocate memory using the new operator fails. This indicates that the memory available to the common language runtime has been exhausted.

Exception Handling

- try-catch block:
  - The purpose of a try-catch block is to catch and handle an exception generated by working code. Some exceptions can be handled in a catch block and the problem solved.
- finally block:
  - The purpose of a finally statement is to ensure that the necessary cleanup of objects, usually objects that are holding external resources, occurs immediately, even if an exception is thrown.
- throw keyword:
  - The throw is a keyword and it is useful to throw an exception manually during the execution of the program and we can handle those thrown exceptions using try-catch blocks based on our requirements.
  - The throw keyword will raise only the exceptions that are derived from the Exception base class.

Exception Handling

- **Custom Exception:**
  - .NET provides a hierarchy of exception classes ultimately derived from the base class Exception.
  - However, if none of the predefined exceptions meets your needs, you can create your own exception classes by deriving from the Exception class.
  - In C#, the exceptions are divided into two types such as
    - **System exception:** An exception that is raised implicitly under a program by the exception manager.
    - **Application exception:** An exception that is raised explicitly under a program based on our own condition (i.e. user-defined condition) is known as an application exception.

Exception Handling

- Custom Exception:
  - When creating your own exceptions, end the class name of the user-defined exception with the word "Exception", and implement the three common constructors.

```
using System;
public class EmployeeListNotFoundException : Exception {
public EmployeeListNotFoundException() { }
public EmployeeListNotFoundException(string message) : base(message)
{
}
public EmployeeListNotFoundException(string message, Exception inner)
: base(message, inner)
{
}
}
```

Generics

- Generics introduce the concept of type parameters to the .NET Framework, which make it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code.
- Generic classes and methods combine reusability, type safety, and efficiency in a way that their non-generic counterparts cannot.
- Generics are most frequently used with collections and the methods that operate on them.
- The **System.Collections.Generic** namespace contains several generic-based collection classes.
- You can create your own generic interfaces, classes, methods, events, and delegates.

Generics

```
public class GenericList<T>{
public void Add(T input) { }
}
class TestGenericList{
private class ExampleClass { }
static void Main() {
GenericList<int> list1 = new GenericList<int>();
list1.Add(1);
GenericList<string> list2 = new GenericList<string>();
list2.Add("");
GenericList<ExampleClass> list3 = new GenericList<ExampleClass>();
list3.Add(new ExampleClass());
}
}
```

Generics

- **Generic Constraint:**
  - Constraints inform the compiler about the capabilities a type argument must have.
  - Constraints specify the capabilities and expectations of a type parameter.
  - Constraints are specified by using the where contextual keyword.

```
class EmployeeList<T> where T : Employee, IEmployee,System.IComparable<T>,
new()
{
// ...
}
```

Multiple constraints can be applied to the same type parameter, and the constraints themselves can be generic types.

Collections

- Collections provide a more flexible way to work with groups of objects.
- Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change.
- A collection is a class, so you must declare an instance of the class before you can add elements to that collection.
- If your collection contains elements of only one data type, you can use one of the classes in the System.Collections.Generic namespace.
- A generic collection enforces type safety so that no other data type can be added to it.
- Some of the common collection classes are as follows:
  - System.Collections.Generic classes
  - System.Collections classes

Collections

- System.Collections.Generic classes:
  - **Dictionary<TKey,TValue>:**
    - Represents a collection of key/value pairs that are organized based on the key.
  - **List<T>:**
    - Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
  - **Queue<T>:**
    - Represents a first in, first out (FIFO) collection of objects.
  - **SortedList<TKey,TValue>:**
    - Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
  - **Stack<T>:**
    - Represents a last in, first out (LIFO) collection of objects.

Collections

- System.Collections classes:
  - **ArrayList:**
    - Represents an array of objects whose size is dynamically increased as required.
  - **Hashtable:**
    - Represents a collection of key/value pairs that are organized based on the hash code of the key.
  - **Queue:**
    - Represents a first in, first out (FIFO) collection of objects.
  - **Stack:**
    - Represents a last in, first out (LIFO) collection of objects.

Collections

- **Iterators:**
  - An iterator is used to perform a custom iteration over a collection. An iterator can be a method or a get accessor.
  - You call an iterator by using a foreach statement.
  - Each iteration of the foreach loop calls the iterator.

- **IEnumerable:**
  - IEnumerable in C# is an interface that defines one method, GetEnumerator which returns an IEnumerator interface.
  - This allows readonly access to a collection then a collection that implements IEnumerable can be used with a for-each statement.

LTI
Let's Solve

Object Oriented Programming in C#

Quiz

A Larsen & Toubro Group Company

Q: Which feature of OOP indicates code reusability?

- Encapsulation
- Inheritance
- Abstraction
- Polymorphism

Q: Which of the following is NOT a .NET Exception class?

- Exception
- StackMemoryException
- DivideByZeroException
- OutOfMemoryException

Q: A class can be stopped to get inherited by declaring it as?

- not inheritable
- extends
- inherits
- sealed

Q: Which of the following statements is correct?

- Only one object can be created from an abstract class.
- By default methods are virtual.
- If a derived class does not provide its own version of virtual method then the one in the base class is used.
- Each derived class does not have its own version of a virtual method.

Q: Which of the following is an ordered collection class?

- Map
- Hashtable
- Queue
- Stack

LTI
Let's Solve

ADO.NET

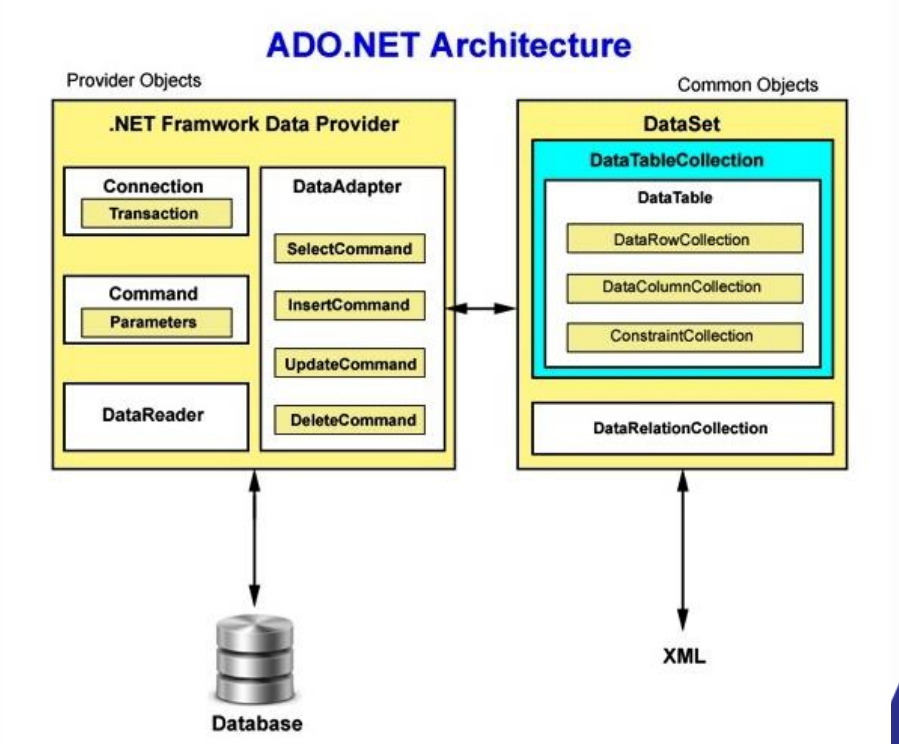
A Larsen & Toubro Group Company

Overview of ADO.NET

- ADO.NET is a set of classes that expose data access services for .NET Framework programmers.
- ADO.NET provides a rich set of components for creating distributed, data-sharing applications.
- It is an integral part of the .NET Framework, providing access to relational, XML, and application data.
- ADO.NET provides consistent access to data sources such as SQL Server and XML, and to data sources exposed through OLE DB and ODBC.
- ADO.NET separates data access from data manipulation into discrete components that can be used separately.
- ADO.NET includes .NET Framework data providers for connecting to a database, executing commands, and retrieving results.

ADO.NET architecture

- ADO.NET Components:
  - The two main components of ADO.NET for accessing and manipulating data are
    - .NET Framework data providers.
    - The DataSet

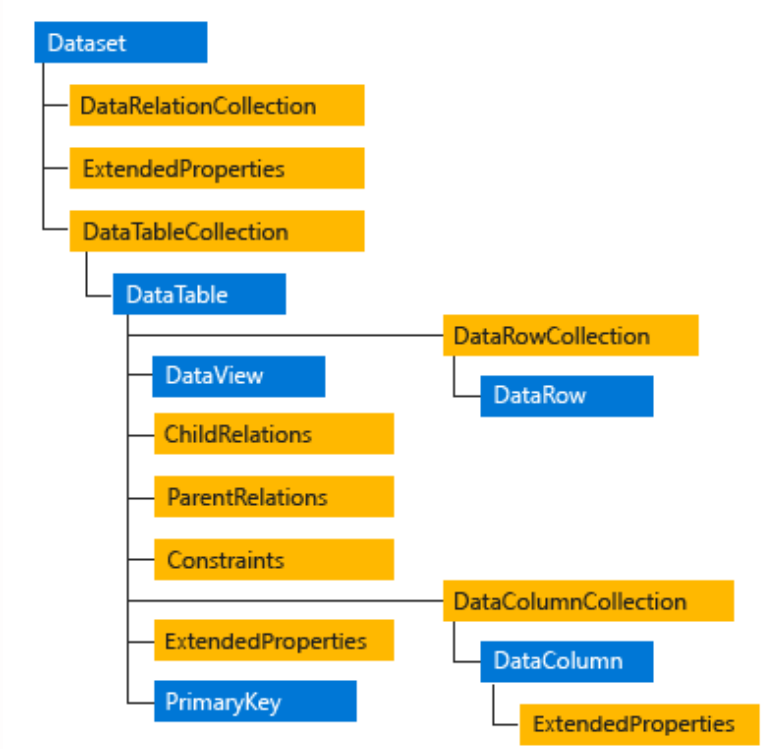


ADO.NET architecture

- **.NET Framework data providers:**
  - The .NET Framework Data Providers are components that have been explicitly designed for data manipulation and fast, forward-only, read-only access to data.
  - The **Connection** object provides connectivity to a data source.
  - The **Command** object enables access to database commands to return data, modify data, run stored procedures, and send or retrieve parameter information.
  - The **DataReader** provides a high-performance stream of data from the data source.
  - Finally, the **DataAdapter** provides the bridge between the DataSet object and the data source.
  - The **DataAdapter** uses Command objects to execute SQL commands at the data source to both load the DataSet with data and reconcile changes that were made to the data in the DataSet back to the data source.

ADO.NET architecture

- **The DataSet:**
  - The ADO.NET DataSet is explicitly designed for data access independent of any data source.
  - As a result, it can be used with multiple and differing data sources, used with XML data, or used to manage data local to the application.
  - The DataSet contains a collection of one or more DataTable objects consisting of rows and columns of data, and also primary key, foreign key, constraint, and relation information about the data in the DataTable objects.



ADO.NET architecture

- **DataSet:**
  - The ADO.NET DataSet is a memory-resident representation of data that provides a consistent relational programming model regardless of the source of the data it contains.
  - A DataSet represents a complete set of data including the tables that contain, order, and constrain the data, as well as the relationships between the tables.
  - There are several ways of working with a DataSet, which can be applied independently or in combination. You can:
    - Programmatically create a DataTable, DataRelation, and Constraint within a DataSet and populate the tables with data.
    - Populate the DataSet with tables of data from an existing relational data source using a DataAdapter.
    - Load and persist the DataSet contents using XML.

ADO.NET architecture

- Creating a DataSet:

```
DataSet customerOrders = new DataSet("CustomerOrders");
```

- Populating a DataSet from a DataAdapter:

```
string queryString = "SELECT CustomerID, CompanyName FROM dbo.Customers";
SqlDataAdapter adapter = new SqlDataAdapter(queryString, connection);
DataSet customers = new DataSet();
adapter.Fill(customers, "Customers");
```

ADO.NET architecture

- **DataTable:**
  - Represents one table of in-memory data.
  - ADO.NET enables you to create DataTable objects and add them to an existing DataSet.

```
DataSet customerOrders = new DataSet("CustomerOrders");
DataTable ordersTable = customerOrders.Tables.Add("Orders");
DataColumn pkOrderID =
ordersTable.Columns.Add("OrderID", typeof(Int32));
ordersTable.Columns.Add("OrderQuantity", typeof(Int32));
ordersTable.Columns.Add("CompanyName", typeof(string));
ordersTable.PrimaryKey = new DataColumn[] { pkOrderID };
```

ADO.NET architecture

- DataColumn:
  - Represents the schema of a column in a DataTable.

```
DataTable table = new DataTable("Product");
DataColumn column = new DataColumn();
column.DataType = System.Type.GetType("System.Decimal");
column.AllowDBNull = false;
column.Caption = "Price";
column.ColumnName = "Price";
column.DefaultValue = 25;
table.Columns.Add(column);
```

ADO.NET architecture

- DataRow:
  - Represents a row of data in a DataTable.

```
DataRow row;
for(int i = 0; i < 10; i++)
{
row = table.NewRow();
row["Price"] = i + 1;

// Be sure to add the new row to the
// DataRowCollection.
table.Rows.Add(row);
}
```

ADO.NET Transactions

- A transaction consists of a single command or a group of commands that execute as a package.
- Transactions allow you to combine multiple operations into a single unit of work.
- If a failure occurs at one point in the transaction, all the updates can be rolled back to their pre-transaction state.
- A transaction must conform to the ACID properties—atomicity, consistency, isolation, and durability.

ADO.NET Transactions

- **Local Transactions:**
  - A transaction is a local transaction when it is a single-phase transaction and is handled by the database directly. It is performed on a single connection.
  - Steps to perform a transaction:
    - Call the **BeginTransaction** method of the SqlConnection object to mark the start of the transaction.
    - Assign the Transaction object to the Transaction property of the SqlCommand to be executed.
    - Execute the required commands.
    - Call the Commit method of the **SqlTransaction** object to complete the transaction or call the Rollback method to end the transaction.
    - If the connection is closed or disposed before either the Commit or Rollback methods have been executed, the transaction is rolled back.

ADO.NET Transactions

- **Distributed Transactions:**
  - A distributed transaction is a transaction that affects several resources.
  - In the .NET Framework, distributed transactions are managed through the API in the **System.Transactions** namespace.
  - The System.Transactions API will delegate distributed transaction handling to a transaction monitor such as the **Microsoft Distributed Transaction Coordinator (MS DTC)** when multiple persistent resource managers are involved.

- **Using TransactionScope:**
  - The **TransactionScope** class makes a code block transactional by implicitly enlisting connections in a distributed transaction.
  - You must call the **Complete** method at the end of the TransactionScope block before leaving it. Leaving the block invokes the **Dispose** method.

LTI
Let's Solve

ADO.NET

Quiz

A Larsen & Toubro Group Company

Q: To use the .NET Framework Data Provider for SQL Server, an application must reference the _____ namespace.

- System.Data.Client
- System.Data.SqlClient
- System.Data.Sql
- None of the mentioned

Q: _____method of the command object is best suited when you have aggregate functions in a SELECT statement?

- ExecuteScalar
- ExecuteReader
- ExecuteNonQuery
- None of the mentioned

Q: Which of the following is not the method of DataAdapter?

- Fill
- FillSchema
- ReadData
- Update

Let's Solve