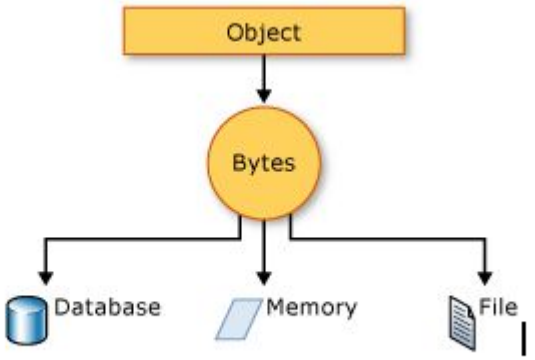
1 Serialization in C#

-Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file.
-Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

How serialization works

This illustration shows the overall process of serialization.



-The object is serialized to a stream, which carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it can be stored in a database, a file, or memory.

Uses for serialization

-Serialization allows the developer to save the state of an object and recreate it as needed, providing storage of objects as well as data exchange.

-Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

Types

- Binary Serialization(uses using system.runtime.formatters.binary)
    - XML Serialization
    - JSON Serialization

Example

Create an employee class

Employee.cs

```
[Serializable]

Class Employee{
    public int id;

    public string name;

public Employee(int id,string name){

        this.id=id;

        this.name=name

}

}
```

Now in program.cs file

```
public static void Main(String[] args)

{

Employee emp=new Employee(211,”khushi”);

String path=@””;

FileStream fs= new FileStream (path,FileMode.OpenorCreate);

BinaryFormatter bf= new BinaryFormatter();
```

```
bf.Serialize(fs,emp);

fs.close();

}
```

2 Generic class,method and interface.

-Generic is a class which allows the user to define classes and methods with the placeholder.

- The basic idea behind using Generic is to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes, and interfaces.

-A primary limitation of collections is the absence of effective type checking.

-This means that you can put any object in a collection because all classes in the C# programming languages extend from the object base class.

-This compromises type safety and contradicts the basic definition of C# as a type-safe language.

-In addition, using collections involves a significant performance overhead in the form of implicit and explicit type casting that is required to add or retrieve objects from a collection.

-To address the type safety issue, the .NET framework provides generics to create classes, structures, interfaces, and methods that have placeholders for the types they use.

-Generics are commonly used to create type-safe collections for both reference and value types.

-The .NET framework provides an extensive set of interfaces and classes in the **System.Collections.Generic** namespace for implementing generic collections.

Advantages of Generics

Reusability: You can use a single generic type definition for multiple purposes in the same code without any alterations. For example, you can create a generic method to add two numbers. This method can be used to add two integers as well as two floats without any modification in the code.

Type Safety: Generic data types provide better type safety, especially in the case of collections. When using generics you need to define the type of objects to be passed to a collection. This helps the compiler to ensure that only those object types that are defined in the definition can be passed to the collection.

Performance: Generic types provide better performance as compared to normal system types because they reduce the need for boxing, unboxing, and typecasting of variables or objects.

Generic class

-Generics in C# is its most powerful feature.

-It allows you to define the type-safe data structures.

-This out-turn in a remarkable performance boost and high-grade code, because it helps to reuse data processing algorithms without replicating type-specific code.

- Generics are similar to templates in C++ but are different in implementation and capabilities.

-Generics introduces the concept of type parameters, because of which it is possible to create methods and classes that defers the framing of data type until the class or method is declared and is instantiated by client code.

-Generic types perform better than normal system types because they reduce the need for boxing, unboxing, and type casting the variables or objects.

Parameter types are specified in generic class creation.

```
using System;

public class GFG<T> {

    private T data;


    public T value

    {

        get

        {

            return this.data;

        }

        set

        {

            this.data = value;

        }

    }

}


class Test {

    static void Main(string[] args)

    {
```

```
        GFG<string> name = new GFG<string>();
        name.value = "GeeksforGeeks";
        GFG<float> version = new GFG<float>();
        version.value = 5.0F;
        Console.WriteLine(name.value);
        Console.WriteLine(version.value);
    }
}
```

Explanation-The preceding example defines a generic class, *GFG*, which uses a generic type parameter ‘T’. In the Main() method, two instances of GFG have been created by replacing ‘T’ with ‘string’ and ‘float’ data types. These objects are used to store ‘string’ and ‘float’ values respectively. The GFG class ensures type safety by accepting the required type in its constructor.

Generic method

A Generic method with various parameters: Just as a method can take one argument, generics can take various parameters. One argument can be passed as a familiar type and other as a generic type, as shown below :

```
using System;
public class GFG {
public void Display<TypeOfValue>(string msg, TypeOfValue value)
    {
        Console.WriteLine("{0}:{1}", msg, value);
    }
}
```

```
public class Example {
   public static int Main()
   {
      GFG p = new GFG();


      p.Display<int>("Integer", 122);
      p.Display<char>("Character", 'H');
      p.Display<double>("Decimal", 255.67);
      return 0;
   }
}
```

Generic interface

A generic collection is a generic class that implements the IEnumerable<T> generic interface.

The IEnumerable<T> interface allows you to support a foreach loop.

```
public interface IMyGenericInterfce<T>
{
    void Method1(T a, T b);
    T Method2(T a, T b);
}
```

You can use the constraints also with the generic interface:

```
//the type parameter must be a value type
```

```
public interface IMyGenericInterfce<T> where T :
struct
 {
void Method1(T a, T b);
 T Method2(T a, T b);
}
```

Implementing our generic interface:

```
public class MyClass : IMyGenericInterfce<int>
 {
public void Method1(int a, int b) {. . .}
 public int Method2(int a, int b) {. . .}
 }
```

3 What is ASP.NET MVC?

ASP.NET MVC is an open source web development framework from Microsoft that provides a Model View Controller architecture.

ASP.net MVC offers an alternative to ASP.net web forms for building web applications.

It is a part of the .Net platform for building, deploying and running web apps.

You can develop web apps and websites with the help of HTML, CSS, jQuery, Javascript, etc.

Why ASP.net MVC?

Although web forms were very successful, Microsoft thought of developing ASP.net MVC. The main issue with ASP.net webForms is performance.

In a web application, there are four main aspects which define performance:-

- Response time issues
- Problem of Unit Testing
- HTML customization
- Reusability of the code-behind class

ASP.net MVC excels on the above parameters.

Features of MVC

- Easy and frictionless testability
- Full control over your HTML, JavaScript , and URLs
- Leverage existing ASP.Net Features
- A new presentation option for ASP.Net
- A simpler way to program Asp.Net
- Clear separation of logic: Model, View, Controller
- Test-Driven Development
- Support for parallel development

Things to remember while creating MVC Application

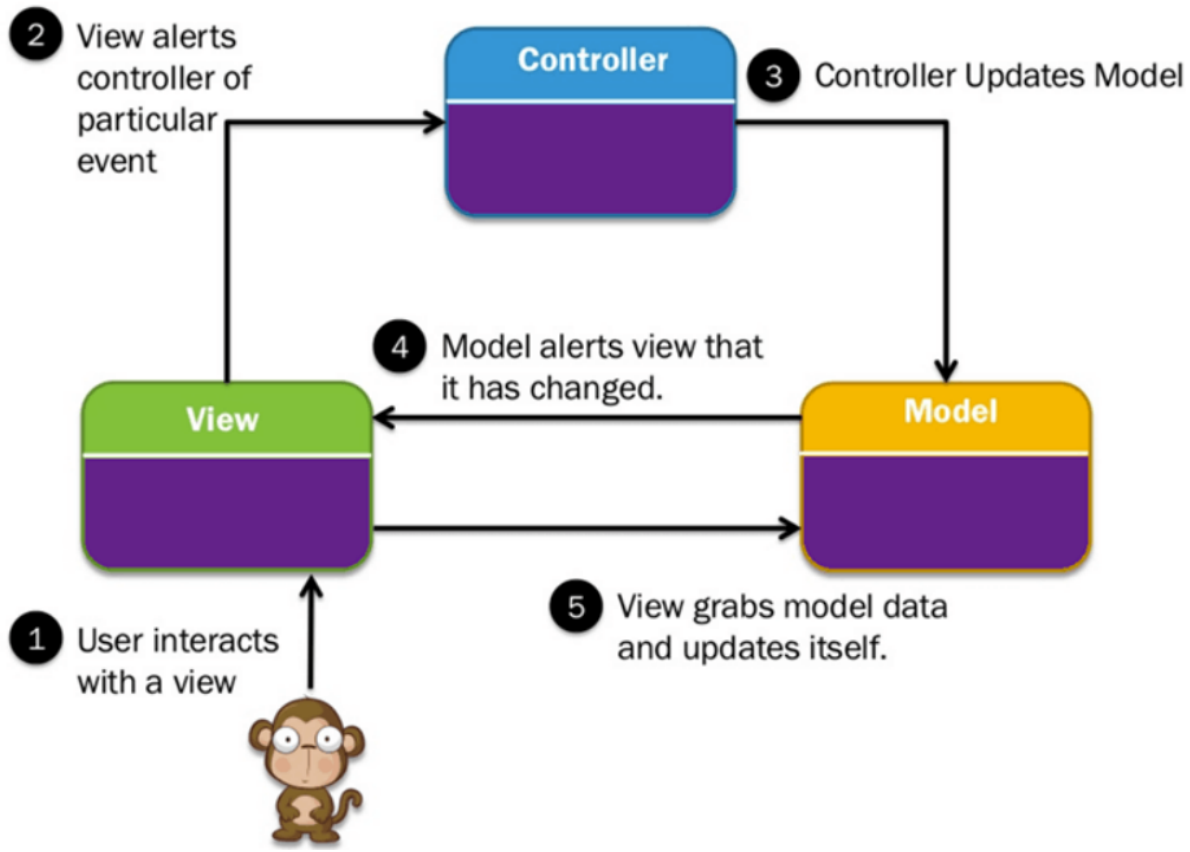
Here, are a few useful things which you need to remember for creating MVC application:

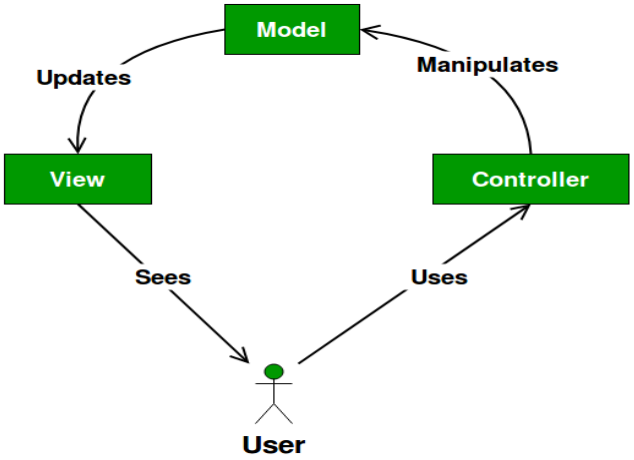
- You need to remember that ASP .net MVC is NOT a replacement of ASP.Net web forms based applications
- The approach of MVC app development must be decided based on the application requirements and features provided by ASP .net MVC to suit the specific development needs.

· Application development process with ASP .NET MVC is more complex compared with web forms based applications.

· Application maintainability is always higher with separation of application tasks.

MVC architectural Pattern





MVC is a software architecture pattern which follows the separation of concerns method. In this model .Net applications are divided into three interconnected parts which are called Model, View, and Controller.

The goal of the MVC pattern is that each of these parts can be developed, tested in relative isolation and also combined to create a very robust application.

Let see all of them in detail:

Models

Model objects are parts of the application which implement the logic for the application's data domain. It retrieves and stores model state in a database. For example, a product object might retrieve information from a database, operate on it. Then write information back to the products table in the SQL server.

Views

Views are the components which are used to display the application's user interface (UI). It displays the .Net MVC application which is created from the model data.

The common example would be an edit view of an Item table. It displays text boxes, pop-ups and checks boxes based on the current state of products & objects.

Controller

Controllers handle user interaction, work with the model, and select a view to render that displays UI. In a .Net MVC app, the view only displays information, the controller manages and responds to user input & interaction.

For example, the controller manages query-string values and passes those values to the model.

Advantages of ASP.NET MVC

- Highly maintainable applications by default
- It allows you to replace any component of the application.
- Better support for Test Driven Development

· Complex applications are easy to manage because of divisions of Model, View, and Controllers.

· Offering robust routing mechanism with front controller pattern

· Offers better control over application behaviour with the elimination of view state and server-based forms

· .Net MVC applications are supported by large teams of developers and Web designers

· It offers more control over the behaviours of the application. It also uses an optimised bandwidth for requests made to the server

Disadvantages of ASP.NET MVC

· You can't see design page previews like the .aspx page.

· You need to run the program every time to see its actual design.

· Understanding the flow of the application can be challenging

· It is quite complicated to implement, so it is not an ideal option for small level applications

· ASP.NET MVC is hard to learn, as it requires a great understanding of MVC pattern

Best practices while using ASP.Net MVC

· Create a separate assembly for MODEL in case of large and complex code to avoid any unwanted situation

· The model should include business logic, session maintenance, validation part, and data logic part.

· VIEW should not contain any business logic and session maintenance, use ViewData to access data in View

· Business logic and data access should never occur in ControllerViewData

· The controller should only be responsible for preparing and returning a view, calling model, redirect to action, etc.

· Delete Demo code from the application when you create it Delete AccountController

· Use only a specific view engine to create HTML markup from your view as it is the combination of HTML and the programming code.

Summary

· ASP.NET MVC is an open source web development framework from Microsoft that provides a Model View Controller architecture.

· ASP.net MVC offers an alternative to ASP.net web forms for building web applications

· The main issue with ASP.net webForms is performance.

· ASP.net MVC offer Easy and frictionless testability with Full control over your HTML & URLs

· You need to remember that ASP .net MVC is NOT a replacement of ASP.Net web forms based applications

· The approach of MVC app development must be decided based on the application requirements and features provided by ASP .net MVC to suit the specific development needs.

· ASP.net MVC offers Highly maintainable applications by default

· With ASP.net you can't see design page preview like the .aspx page.

· As a best practice, the model should include business logic, session maintenance, validation part, and data logic part.

1. MVC stands for Model, View and Controller.
2. Model is responsible for maintaining application data and business logic.
3. View is a user interface of the application, which displays the data.
4. Controller handles user's requests and renders appropriate Views with Model data.

4 Tuple

The `Tuple<T>` class was introduced in .NET Framework 4.0.

A tuple is a data structure that contains a sequence of elements of different data types.

It can be used where you want to have a data structure to hold an object with properties, but you don't want to create a separate type for it.

```
Tuple<T1, T2, T3, T4, T5, T6, T7, TRest>
```

The following example creates a tuple with three elements:

```
Tuple<int, string, string> person = new Tuple <int, string, string>(1, "Steve", "Jobs");
```

In the above example, we created an instance of the `Tuple` that holds a person's record.

We specified a type for each element and passed values to the constructor.

Specifying the type of each element is cumbersome. C# includes a static helper class Tuple, which returns an instance of the `Tuple<T>` without specifying each element's type, as shown below.

```
var person = Tuple.Create(1, "Steve", "Jobs");
```

A tuple can only include a maximum of eight elements.

It gives a compiler error when you try to include more than eight elements.

```
var numbers = Tuple.Create(1, 2, 3, 4, 5, 6, 7, 8);
```

Accessing Tuple Elements

A tuple element can be accessed with `Item<elementNumber>` properties, e.g., Item1, Item2, Item3, and so on up to Item7 property.

The Item1 property returns the first element, Item2 returns the second element, and so on.

The last element (the 8th element) will be returned using the `Rest` property.

Eg of Accessing Tuple Elements

```
var person = Tuple.Create(1, "Steve", "Jobs");
person.Item1; // returns 1
person.Item2; // returns "Steve"
person.Item3; // returns "Jobs"

var numbers = Tuple.Create("One", 2, 3, "Four", 5, "Six", 7, 8);
numbers.Item1; // returns "One"
numbers.Item2; // returns 2
numbers.Item3; // returns 3
numbers.Item4; // returns "Four"
numbers.Item5; // returns 5
numbers.Item6; // returns "Six"
numbers.Item7; // returns 7
numbers.Rest; // returns (8)
numbers.Rest.Item1; // returns 8
```

Generally, the 8th position is for the nested tuple, which you can access using the `Rest` property.

Nested Tuples

If you want to include more than eight elements in a tuple, you can do that by nesting another tuple object as the eighth element.

The last nested tuple can be accessed using the `Rest` property. To access the nested tuple's element, use the `Rest.Item1.Item<elelementNumber>` property.

Eg of Nested Tuple

var numbers = Tuple.Create(1, 2, 3, 4, 5, 6, 7, Tuple.Create(8, 9, 10, 11, 12, 13));

numbers.Item1; // returns 1

numbers.Item7; // returns 7

numbers.Rest.Item1; //returns (8, 9, 10, 11, 12, 13)

numbers.Rest.Item1.Item1; //returns 8

numbers.Rest.Item1.Item2; //returns 9

You can include the nested tuple object anywhere in the sequence. However, it is recommended to place the nested tuple at the end of the sequence so that it can be accessed using the `Rest` property.

Tuple as a Method Parameter

A method can have a tuple as a parameter.

```
static void Main(string[] args)
{
var person = Tuple.Create(1, "Steve", "Jobs");
DisplayTuple(person); }
Static void DisplayTuple(Tuple<int,string,string>
person)
{
Console.WriteLine($"Id = { person.Item1}");
Console.WriteLine($"First Name = { person.Item2}");
Console.WriteLine($"Last Name = { person.Item3}");
}
```

Tuple as a Return Type

A Tuple can be returned from a method.

```
static void Main(string[] args)
{
var person = GetPerson();
}
static Tuple<int, string, string> GetPerson()
 {
return Tuple.Create(1, "Bill", "Gates");
}
```

Usage of Tuple

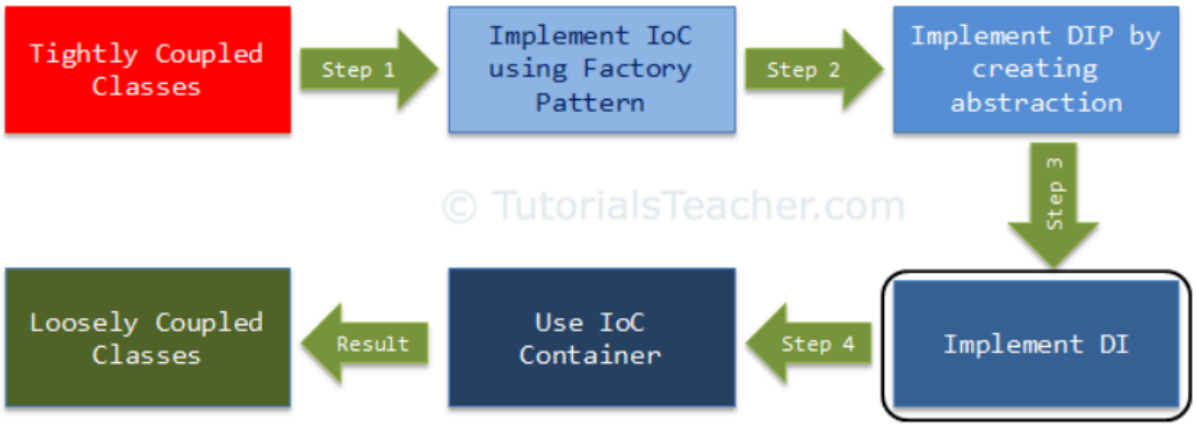
Tuples can be used in the following scenarios:

1. When you want to return multiple values from a method without using `ref` or `out` parameters.
2. When you want to pass multiple values to a method through a single parameter.
3. When you want to hold a database record or some values temporarily without creating a separate class.

Tuple Limitations:

1. The `Tuple` is a reference type and not a value type. It allocates on heap and could result in CPU intensive operations.
2. The `Tuple` is limited to include eight elements. You need to use nested tuples if you need to store more elements. However, this may result in ambiguity.
3. The `Tuple` elements can be accessed using properties with a name pattern `Item<elementNumber>`, which does not make sense.

5 Dependency Injection



Dependency Injection (DI) is a design pattern used to implement IoC.

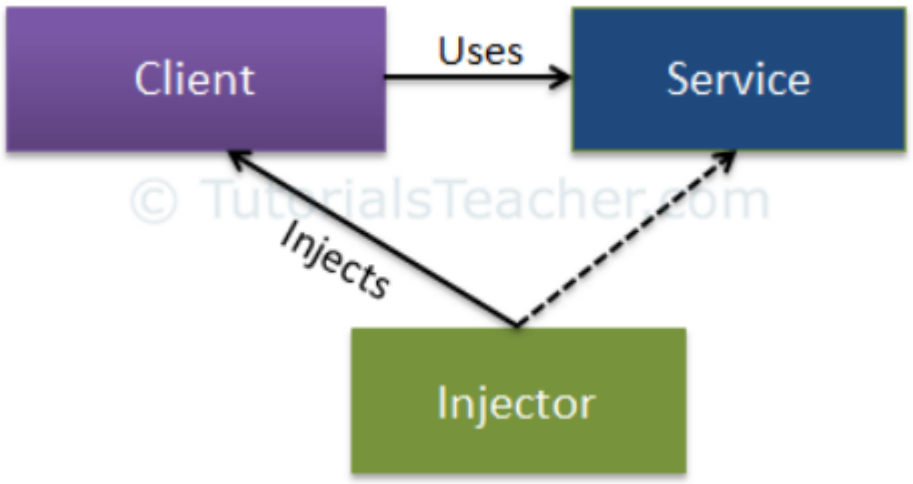
It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 3 types of classes.

1. **Client Class:** The client class (dependent class) is a class which depends on the service class

2. **Service Class:** The service class (dependency) is a class that provides service to the client class.
3. **Injector Class:** The injector class injects the service class object into the client class.

The following figure illustrates the relationship between these classes:



As you can see, the injector class creates an object of the service class, and injects that object to a client object.

In this way, the DI pattern separates the responsibility of creating an object of the service class out of the client class.

Dependency Injection implementation ways

As you have seen above, the injector class injects the service (dependency) to the client (dependent). The injector class injects dependencies broadly in three ways: through a constructor, through a property, or through a method.

Constructor Injection: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.

- This is a widely used way to implement DI.

- Dependency Injection is done by supplying the DEPENDENCY through the class's constructor when creating the instance of that class.
- The injected component can be used anywhere within the class.
- Recommended to use when the injected dependency, you are using across the class methods.
- It addresses the most common scenario where a class requires one or more dependencies.

Property Injection: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.

- Recommended using when a class has optional dependencies, or where the implementations may need to be swapped.
- Different logger implementations could be used in this way.
- Does not require the creation of a new object or modifying the existing one. Without changing the object state, it could work.

Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

- Inject the dependency into a single method and generally for the use of that method.
- It could be useful, where the whole class does not need the dependency, only one method having that dependency.
- This is the way is rarely used.

Service lifetimes

Services can be registered with one of the following lifetimes:

Transient

Scoped

Singleton

The following sections describe each of the preceding lifetimes. Choose an appropriate lifetime for each registered service.

Transient

Transient lifetime services are created each time they're requested from the service container. This lifetime works best for lightweight, stateless services. Register transient services with AddTransient.

In apps that process requests, transient services are disposed at the end of the request.

Scoped

For web applications, a scoped lifetime indicates that services are created once per client request (connection). Register scoped services with AddScoped.

In apps that process requests, scoped services are disposed of at the end of the request.

When using Entity Framework Core, the AddDbContext extension method registers DbContext types with a scoped lifetime by default.

Singleton

Singleton lifetime services are created either:

The first time they're requested.

By the developer, when providing an implementation instance directly to the container. This approach is rarely needed.

6 LINQ

The acronym LINQ stands for Language Integrated Query.

Microsoft's query language is fully integrated and offers easy data access from in-memory objects, databases, XML documents, and many more.

It is through a set of extensions, LINQ ably integrates queries in C# and Visual Basic.

Developers across the world have always encountered problems in querying data because of the lack of a defined path and need to master a multiple of technologies like SQL, Web Services, XQuery, etc.

Introduced in Visual Studio 2008 and designed by Anders Hejlsberg, LINQ (Language Integrated Query) allows writing queries even without the knowledge of query languages like SQL, XML etc.

LINQ queries can be written for diverse data types.

Example of a LINQ query

```
using System;

using System.Linq;

class Program {

 static void Main()   {

string[] words = {"hello", "wonderful", "LINQ", "beautiful", "world"};

var shortWords = from word in words
```

```
                    where word.Length <= 5

                   select word;

foreach (var word in shortWords) {

      Console.WriteLine(word);

   }

  Console.ReadLine();  }  }
```

Syntax of LINQ

There are two syntaxes of LINQ. These are the following ones.

Lamda (Method) Syntax

```
var longWords = words.Where( w ⇒ w.length > 10);

Dim longWords = words.Where(Function(w) w.length > 10)
```

Query (Comprehension) Syntax

```
var longwords = from w in words where w.length > 10;

Dim longwords = from w in words where w.length > 10
```

Types of LINQ

The types of LINQ are mentioned below in brief.

· LINQ to Objects

The LINQ to Objects provider allows us to query an in-memory object such as an array, collection, and generics types. It provides many built-in functions that we can use to perform many useful operations such as filtering, ordering, and grouping with minimum code.

· LINQ to XML(XLINQ)

The LINQ to XML provider is basically designed to work with an XML document. So, it allows us to perform different operations on XML data sources such as querying or reading, manipulating, modifying, and saving the changes to XML documents. The **System.Xml.Linq** namespace contains the required classes for LINQ to XML.

· LINQ to DataSet

The LINQ to Datasets provider provides us the flexibility to query data cached in a Dataset in an easy and faster way. It also allows us to do further data manipulation operations such as searching, filtering, sorting, etc. on the Dataset using the LINQ Syntax.

· LINQ to SQL (DLINQ)

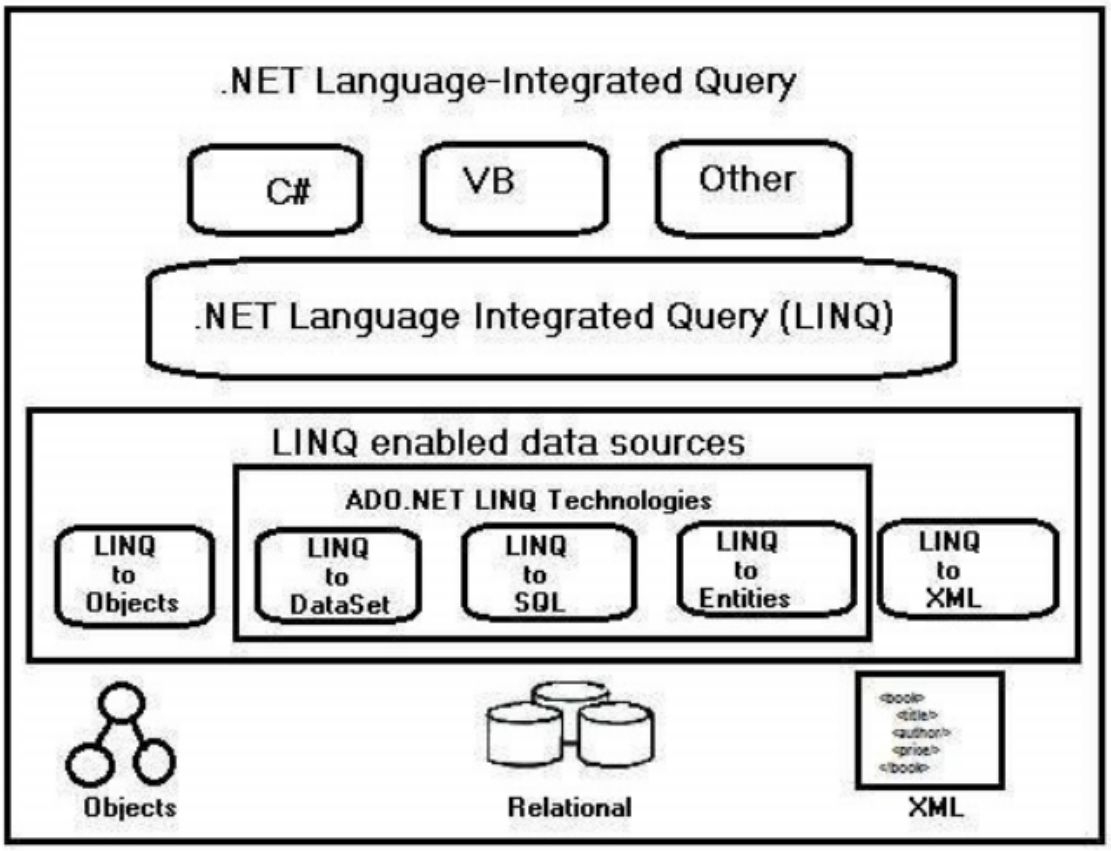
The LINQ to SQL Provider is designed to work with only the SQL Server database. You can consider this as an object-relational mapping (ORM) framework which allows one-to-one mapping between the SQL Server database and related .NET Classes. These .NET classes are going to be created automatically by the wizard based on the database table.

· LINQ to Entities

The LINQ to Entities provider looks like LINQ to SQL. It is also an object-relational mapping (ORM) framework that allows one-to-one, one-to-many, and many-to-many mapping between the database tables and .NET Classes. The point that you need to remember is that it is used to query any database such as SQL Server, Oracle, MySQL, DB2, etc. Now, it is called ADO.NET Entity Framework.

LINQ Architecture in .NET

LINQ has a 3-layered architecture in which the uppermost layer consists of the language extensions and the bottom layer consists of data sources that are typically objects implementing IEnumerable <T> or IQueryable <T> generic interfaces. The architecture is shown below.



Query Expressions

Query expression is nothing but a LINQ query, expressed in a form similar to that of SQL with query operators like Select, Where and OrderBy. Query expressions usually start with the keyword "From".

To access standard LINQ query operators, the namespace **System.Query** should be imported by default. These expressions are written within a declarative query syntax which was C# 3.0.

Below is an example to show a complete query operation which consists of data source creation, query expression definition and query execution.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Operators {
   class LINQQueryExpressions {
      static void Main() {
      int[] scores = new int[] { 97, 92, 81, 60 };
 IEnumerable<int> scoreQuery = from score in scores
                                          where score > 80
                                         select score;
  foreach (int i in scoreQuery) {
            Console.Write(i + " ");
        }
        Console.ReadLine();
     }
   }
}
```

When the above code is compiled and executed, it produces the following result −

```
97 92 81
```

Difference between LINQ and Stored Procedure

There is an array of differences existing between LINQ and Stored procedures. These differences are mentioned below.

- Stored procedures are much faster than a LINQ query as they follow an expected execution plan.
- It is easier to avoid run-time errors while executing a LINQ query than in comparison to a stored procedure as the former has Visual Studio's Intellisense support as well as full-type checking during compile-time.
- LINQ allows debugging by making use of the .NET debugger which is not in case of stored procedures.
- LINQ offers support for multiple databases in contrast to stored procedures, where it is essential to re-write the code for diverse types of databases.
- Deployment of a LINQ based solution is easy and simple in comparison to deployment of a set of stored procedures.

Need For LINQ

Prior to LINQ, it was essential to learn C#, SQL, and various APIs that bind together the both to form a complete application. Since, these data sources and programming languages face an impedance mismatch; a need of short coding is felt.

Below is an example of how many diverse techniques were used by the developers while querying data before the advent of LINQ.

```
SqlConnection = new SqlConnection(connectString);
SqlConnection.Open();

System.Data.SqlClient.SqlCommand sqlCommand = new
SqlCommand();
sqlCommand.Connection = sqlConnection;

sqlCommand.CommandText = "Select * from Customer";
return
```

```
sqlCommand.ExecuteReader(CommandBehavior.CloseConnection)
```

Interestingly, out of the featured code lines, the query gets defined only by the last two. Using LINQ, the same data query can be written in a readable colour-coded form like the following one mentioned below that too in a very less time.

```
Northwind db = new Northwind(@"C:\Data\Northwnd.mdf");

var query = from c in db.Customers select c;
```

Advantages of LINQ

LINQ offers a host of advantages and among them the foremost is its powerful expressiveness which enables developers to express declaratively. Some of the other advantages of LINQ are given below.

- LINQ offers syntax highlighting that proves helpful to find out mistakes during design time.
- LINQ offers IntelliSense which means writing more accurate queries easily.
- Writing codes is quite faster in LINQ and thus development time also gets reduced significantly.
- LINQ makes easy debugging due to its integration in the C# language.
- Viewing relationships between two tables is easy with LINQ due to its hierarchical feature and this enables composing queries joining multiple tables in less time.
- LINQ allows usage of a single LINQ syntax while querying many diverse data sources and this is mainly because of its unique foundation.
- LINQ is extensible, which means it is possible to use knowledge of LINQ to query new data source types.

- LINQ offers the facility of joining several data sources in a single query as well as breaking complex problems into a set of short queries easy to debug.
- LINQ offers easy transformation for conversion of one data type to another like transforming SQL data to XML data.

7 Architecture of .NET Framework

The two major components of .NET Framework are the Common Language Runtime and the .NET Framework Class Library.

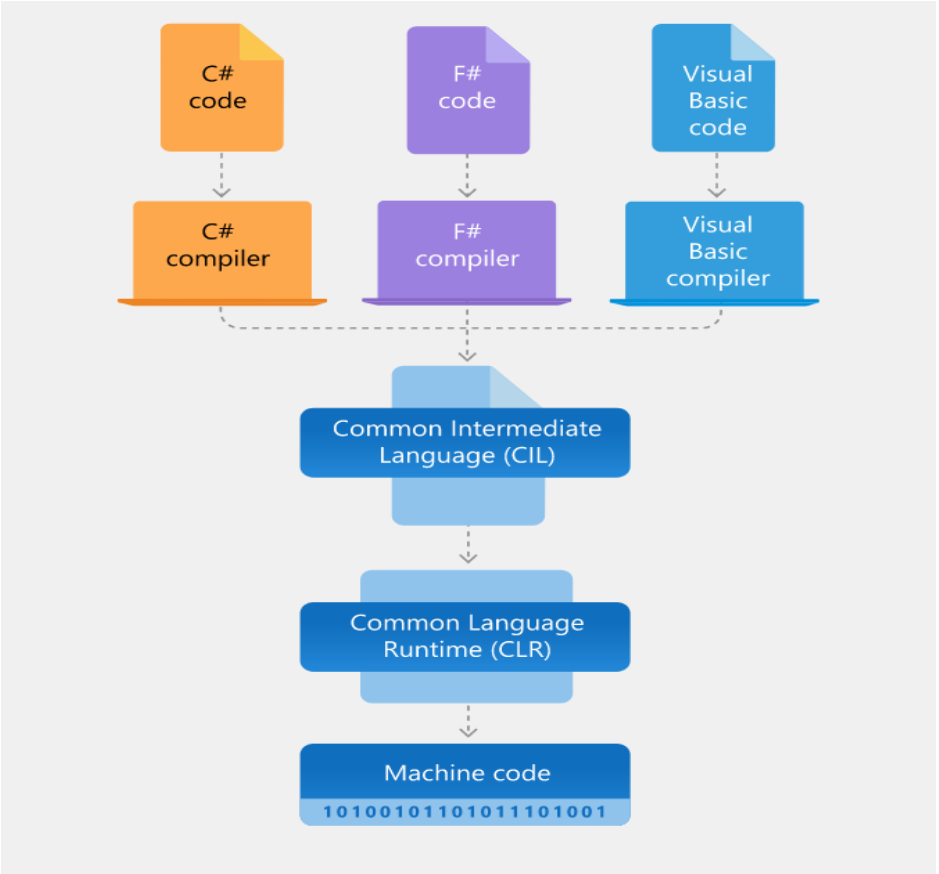
The **Common Language Runtime** (CLR) is the execution engine that handles running applications. It provides services like thread management, garbage collection, type-safety, exception handling, and more.

The **Class Library** provides a set of APIs and types for common functionality. It provides types for strings, dates, numbers, etc. The Class Library includes APIs for reading and writing files, connecting to databases, drawing, and more.

.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.



8 ViewBag,ViewData and TempData

ViewBag, ViewData, and TempData all are objects in ASP.NET MVC and these are used to pass the data in various scenarios.

The following are the scenarios where we can use these objects.

1. Pass the data from Controller to View.
2. Pass the data from one action to another action in the same Controller.
3. Pass the data in between Controllers.
4. Pass the data between consecutive requests.

ViewBag

ViewBag is a dynamic object to pass the data from Controller to View. And, this will pass the data as a property of object ViewBag. And we

have no need to typecast to read the data or for null checking. The scope of ViewBag is permitted to the current request and the value of ViewBag will become null while redirecting.

Ex Controller

```
Public ActionResult Index()
{
   ViewBag.Title = “Welcome”;
   return View();
}
```

View

```
<h2>@ViewBag.Title</h2>
```

ViewData

ViewData is a dictionary object to pass the data from Controller to View where data is passed in the form of key-value pair. And typecasting is required to read the data in View if the data is complex and we need to ensure null check to avoid null exceptions. The scope of ViewData is similar to ViewBag and it is restricted to the current request and the value of ViewData will become null while redirecting.

Ex

Controller:

```
Public ActionResult Index()
{
   ViewData[”Title”] = “Welcome”;
   return View();
}
```

View

```
<h2>@ViewData[“Title”]</h2>
```

TempData

TempData is a dictionary object to pass the data from one action to another action in the same Controller or different Controllers. Usually, TempData objects will be stored in a session object. Tempdata is also required to typecast and for null checking before reading data from it.

TempData scope is limited to the next request and if we want Tempdata to be available even further, we should use Keep and peek.

Ex - Controller

```
Public ActionResult Index()
{
  TempData[”Data”] = “I am from Index action”;
return View();
}
Public string Get()
{
    return TempData[”Data”] ;
}
```

To summarise, ViewBag and ViewData are used to pass the data from Controller action to View and TempData is used to pass the data from action to another action or one Controller to another Controller.

9 Pattern Matching

C# pattern matching is a feature that allows us to perform matching on data or any object. We can perform pattern matching using the is expression and switch statement.

is expression is used to check, whether an object is compatible with a given type or not.

In the following example, we are implementing **is expression.**

C# Pattern Matching using is Example

Syntax:

expression is pattern

using System;

namespace CSharpFeatures

{

```
    class Student
    {
        public string Name { get; set; } = "Rahul kumar";
    }
    class PatternMatchingExample
    {
        public static void Main(string[] args)
        {
            Student student = new Student();
            if(student is Student)
            {
                Console.WriteLine(student.Name);
            }
        }
    }
}
```

Output:

Rahul kumar

We can also perform pattern matching in **switch statements**. See the following example.

C# Pattern in Switch Case Example 2

Syntax:

```
switch (expression)
```

```
{
    case pattern1:
    // code to be executed
    // if expression matches pattern1
    break;
    case pattern2:
    // code to be executed
    // if expression matches pattern2
    break;
    …
    case patternN:
    // code to be executed
    // if expression matches patternN
    break;
    default:
    // code to be executed if expression
    // does not match any of the above patterns
}

using System;
namespace CSharpFeatures
{
    class Student
```

```
{
    public string Name { get; set; } = "Rahul kumar";
}
class Teacher
{
    public string Name { get; set; } = "Peter";
    public string Specialization { get; set; } = "Computer Science";
}
class PatternMatchingExample
{
    public static void Main(string[] args)
    {
        Student student = new Student();
        Teacher teacher = new Teacher();
        PatterInSwitch(student);
        PatterInSwitch(teacher);
    }
    public static void PatterInSwitch(object obj)
    {
        switch (obj)
        {
            case Student student:
                Console.WriteLine(student.Name);
                break;
```

```
            case Teacher teacher:
                Console.WriteLine(teacher.Name);
                Console.WriteLine(teacher.Specialization);
                break;
            default:
                throw new ArgumentException(
                    message: "Object is not recognized",
                    paramName: nameof(obj));
        }
    }
  }
}
```

Pattern matching allows operations like:

1 Type Pattern

Syntax:

TypeName variable

TypeName _

2 Relational Pattern

Syntax:

< constant

<= constant

> constant

>= constant

3 Property Pattern

Syntax:

{ Property1: value1, Property2 : value2, …, PropertyN: valueN }

4 Positional Pattern

Syntax:

(constant1, constant2, …)

5 var Pattern

Syntax:

var varName

var (varName1, varName2, …)

6 Constant Pattern

expression is 2 // int literal

expression is "Geeks" // string literal

expression is System.DayOfWeek.Monday // enum

expression is null // null

type checking(type pattern)

10 asynchronous types in c#

Asynchronous programming is all about improvement of performance. Basically, we can implement Ajax in two ways (in ASP.NET).

The first option is by updating the panel and Ajax toolkit and the second option is by the jQuery Ajax method. (Let's ignore the various third-party JavaScript libraries).

In C# 5.0 Microsoft has given us the ability to write our own asynchronous code with C#.

Before starting with an example I would like to discuss two master keywords of asynchronous programming, called async and await.

Async

This keyword is used to qualify a function as an asynchronous function. In other words, if we specify the async keyword in front of a function then we can call this function asynchronously. Have a look at the syntax of the asynchronous method.

```
public async void CallProcess()

{

}
```

Here the callProcess() method is declared as an asynchronous method because we have declared the async keyword in front of it. Now it's ready to be called asynchronously.

Await

This keyword is used when we want to call any function asynchronously. Have a look at the following example to understand how to use the await keyword.

Let's think; in the following, we have defined a long-running process.

```
public static Task LongProcess()

{

return Task.Run(() =>

{

System.Threading.Thread.Sleep(5000);

});
```

}

Now, we want to call this long process asynchronously. Here we will use the await keyword.

await LongProcess();

The return type of an asynchronous function is Task. In other words, when it finishes its execution it will complete a Task. Each and every asynchronous method can return three types of values

- Void
- Task
- Task<T>

.

1 Void: Means nothing to return

Though it's not recommended to return void from an asynchronous function, we can return void theoretically. Now, the question is, why is it not recommended to return void? The answer is if we return void then the caller function will not be informed of the completion of the asynchronous function. OK, then in which scenario can we return void? When we want to call an asynchronous function from an event (like a button click) then we can specify void in the event.

2 Task: It will perform one operation, a little similar to void but the difference is there.

Let's see how to return a task from an asynchronous method. Basically the returning task is nothing but sending one signal to the caller function that the task has finished. When a method returns a task, we can use the await keyword to call it.

Note: The await keyword should be within a function qualified by an async keyword (in other words asynchronous), otherwise the compiler will treat it as a normal synchronous function

3 Task<T>: Will return a task with a T type parameter. (I hope you are familiar with the concept of T)

Let's clarify a few more concepts here:

The Main method cannot be defined as asynchronous.

It (the Main method) cannot be invoked by the await keyword.

If any asynchronous method is not invoked by the await keyword then by nature it will behave like the synchronous method.

Function properties should not be defined as asynchronous.

The await keyword may not be inside a "lock" section.

A try/catch block should not call any asynchronous methods from the catch or finally block.

A class constructor or destructor should not define an asynchronous method nor should it be called asynchronously.

Return Task<T> from asynchronous method

Now, let's see how to return Task<T> from an asynchronous method. I hope all of you understand the meaning of T