**1) What is LINQ?**

*LINQ (Language Integrated Query)* is uniform query syntax in C# and VB.NET *used to save and retrieve data from different sources*. It is released within the .NET 3.5 Framework.

**2) Advantages of LINQ**

**Familiar language:** Developers don’t have to learn a new query language for each type of data source or data format.

***Less coding:*** It reduces the amount of code to be written as compared with a more traditional approach.

***Readable code:*** LINQ makes the code more readable so other developers can easily understand and maintain it.

**Standardized way of querying multiple data sources:** The same LINQ syntax can be used to query multiple data sources.

***Compile time safety of queries:*** It provides type checking of objects at compile time.

***IntelliSense Support:*** LINQ provides IntelliSense for generic collections.

Shaping data: You can retrieve data in different shapes.

*LINQ expressions are Strongly Typed*

**3) What we mean by Strongly Typed?**

Strongly typed expressions ensure access to values as the correct type at compile time & prevents type mismatch errors being caught when the code is compiled rather at run-time.

**4) The three main components of LINQ are:**

**Standard Query Operators:** These are the extension methods that form the LINQ patterns. It operates on sequences and forms an API that enables querying of any .NET array. The Standard Query Operators in LINQ allows you to perform functions like determining if a value exists in the sequence, summation over a sequence, etc. The query operators supported by the API are:

| Classification | Standard Query Operators |
| --- | --- |
| Filtering | Where, OfType |
| Sorting | OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse |
| Grouping | GroupBy, ToLookup |
| Join | GroupJoin, Join |
| Projection | Select, SelectMany |
| Aggregation | Aggregate, Average, Count, LongCount, Max, Min, Sum |
| Quantifiers | All, Any, Contains |
| Elements | ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault |
| Set | Distinct, Except, Intersect, Union |
| Partitioning | Skip, SkipWhile, Take, TakeWhile |
| Concatenation | Concat |
| Equality | SequenceEqual |
| Generation | DefaultEmpty, Empty, Range, Repeat |
| Conversion | AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList |

**Language extensions:** LINQ defines optional language extensions when LINQ is primarily implemented as a library for .NET framework. This makes queries a first-class language construct and also provides syntax for writing queries.

**LINQ Providers:** These are set of classes that generate a method that executes the same query against a particular data source. The different providers define the different flavours of LINQ:

LINQ to SQL

LINQ to XML

LINQ to Objects

LINQ to Dataset

LINQ to Entities

**5) Deferred Vs Immediate Query Execution in LINQ**

In **Deferred Execution**, the query is not executed when declared. It is executed when the query object is iterated over a loop.



In **Immediate Execution**, the query is executed when it is declared.



**6) Explain how LINQ is useful than Stored Procedures?**

**Debugging:** It is difficult to debug a stored procedure but as LINQ is part of.NET, visual studios debugger can be used to debug the queries.

**Deployment:** For stored procedure, additional script should be provided but with LINQ everything gets compiled into single DLL hence deployment becomes easy.

**Type Safety:** LINQ is type safe, so queries errors are type checked at compile time.

**7) What is Lambda Expression?**

A lambda expression is an anonymous function that you can use to create delegates or expression tree types. By using lambda expressions, you can write local functions that can be passed as arguments or returned as the value of function calls. ... A lambda expression is the most convenient way to create that delegate.

**8) What is Expression Tree?**

An expression tree is a mechanism to translate executable code into data. Using an expression tree, you can produce a data structure that represents your program.

Consider the following very simple lambda expression:

**Func<int, int, int> function = (a,b) => a + b;**

This statement consists of three sections.

1. A declaration: Func<int, int, int> function
2. An equals operator: =
3. A lambda expression: (a,b) => a + b;

**Expression trees are not executable code,** **they are a form of data structure**. So how does one translate the raw code found in an expression into an expression tree? How does one translate code into data?

LINQ provides a simple syntax for translating code into a data structure called an expression tree. The first step is to add a using statement to introduce the **System.Linq.Expressions** namespace:

Now we can create an expression tree:

**Expression<Func<int, int, int>> expression = (a,b) => a + b;**

The identical lambda expression shown in the previous example is converted into an expression tree declared to be of type Expression<T>. The identifier expression is not executable code; it is a data structure called an expression tree.

**Expression<Func<int, int, int>> addTwoNumbersExpression = (x, y) => x + y;**

**BinaryExpression body = (BinaryExpression)addTwoNumbersExpression.Body;**

**ParameterExpression left = (ParameterExpression)body.Left;**

**ParameterExpression right = (ParameterExpression)body.Right;**

**9) Compiled Queries in LINQ**

There may be scenario where we need to execute a particular query many times and repeatedly. LINQ allows us to make this task very easy by enabling us to create a query and make it compiled always. We call this type of query a compiled query.

**Benefits of Compiled Queries**

1. Query does need to compiled each time so execution of the query is fast.
2. Query is compiled once and can be used any number of times.
3. Query does need to be recompiled even if the parameter of the query is being changed.

**Steps to create a Compiled Query**

1. Create a static class
2. Add namespace System.Data.Linq.
3. Use CompiledQuery class to create complied LINQ query.

namespace ConsoleApplication5  
  {  
    class Program  
        {  
            static void Main(string[] args)  
                {  
                    DataClasses1DataContext context = new DataClasses1DataContext();  
                    var result = MyCompliedQueries.CompliedQueryForPesron(context);  
                    foreach (var r in result)  
                    {  
                        Console.WriteLine(r.FirstName + r.LastName);  
                    }  
   
                    Console.ReadKey(true);  
                      
                }  
        }  
   
    static class MyCompliedQueries  
        {  
               public static Func<DataClasses1DataContext ,IQueryable<Person>>  
               CompliedQueryForPesron = CompiledQuery.Compile(  
                                          (DataClasses1DataContext context)=>  
                                              from c in context.Persons select c );  
   
           
        }  
}

**What is Dependency Injection?**

The Dependency Injection pattern is a particular implementation of Inversion of Control.

**Inversion of Control (IoC)** *means that objects do not create other objects on which they rely to do their work.* Instead, they get the objects that they need from an outside source (for example, an xml configuration file).

**Dependency Injection (DI)** *means that this is done without the object intervention, usually by a framework component that passes constructor parameters and set properties.*

* Dependency injection **eliminates tight coupling between objects** to make both the objects and applications more flexible, reusable, and easier to test that use them,
* It **facilitates the creation of loosely coupled objects and their dependencies**.
* The basic idea behind Dependency Injection is that you should isolate the implementation of an object from the construction of other objects on which it depends.

**Advantages of Dependency Injection**

The advantages of using Dependency Injection pattern and Inversion of Control are the following:

* Reduces class coupling
* Increases code reusing
* Centralized configuration
* Improves code maintainability
* Improves application testing (Code becomes more testable because it abstracts and isolates class dependencies.)

**Disadvantages of Dependency Injection**

However, the primary drawback of dependency injection is that *wiring instances together can become a nightmare if there are too many instances and many dependencies that need to be addressed.*

**Use Dependency with**

DI with Autofac

DI with Unity

DI with Dagger 2

DI with Structure Map (StructureMap.dll.)

**Different Types of Injections are**

* Constructor Injection
* Setter Injection
* Interface Injection
* Service Locator

**In MVC 6**

// Scoped is a single instance for the duration of the scoped request, which means per HTTP request in ASP.NET.

*AddScoped<IGreeting, Greeting>();*

// Singleton is a single instance for the lifetime of the application domain.

*AddSingleton <IGreeting, Greeting>();*

// Transient is a single instance per code request.

*AddTransient<IGreeting, Greeting>();*

**What is mocking?**

Let's think that one application is being develop and many developers are working in this project and each one is assigned to develop a function. Let's think that I am developing a function that will insert one employee information into the DB; if it is not present in the DB then fine and one of my fellow developer is developing the function to check the existence.

And I have completed my function but this guy has not, as he has a little bit of a workload, haha.. Now, as I completed my task, I wanted to test my function but for that I need to depend on the checking function that is still not developed.

So, how I will do that? I need to create mock object that will bypass the checking function. The point to make here is that there are many mocking frameworks to implement the mock object. In this article we will use MOQ as a mocking framework.

Here is our code that we will test using the unit test application.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace TestProjectLibrary

{

public class checkEmployee

{

public virtual Boolean checkEmp()

{

throw new NotImplementedException();

}

}

public class processEmployee

{

public Boolean insertEmployee(checkEmployee objtmp)

{

objtmp.checkEmp();

return true;

}

}

}

Now, see the implementation, the checkEmployee class contains a checkEmp() function that is still not implemented. And we are sending an object of the checkEmployee class to the insertEmployee() function to check whether the employee already exists before it is inserted into the DB.

So, the concept is that since the checkEmployee class is not fully implemented, we will send a mock object of the checkEmployee class as an argument of the insertEmployee() function. Here is sample code of the implementation.

using System;

using Microsoft.VisualStudio.TestTools.UnitTesting;

using TestProjectLibrary;

**using Moq;**

namespace UnitTest

{

[TestClass]

public class UnitTest

{

[TestMethod]

public void TestMethod2()

{

Mock<checkEmployee> chk = new Mock<checkEmployee>();

chk.Setup(x => x.checkEmp()).Returns(true);

processEmployee obje = new processEmployee();

Assert.AreEqual(obje.insertEmployee(chk.Object), true);

}

}

}

Have a look at the first two lines of TestMethod2(). We are defining a mock object associated with checkCmployee class and in the next line we are setting the mock object.

chk.Setup(x => x.checkEmp()).Returns(true);

The preceding line is a bit interesting. Moq has a Setup() function by which we can set up the mock object. We need to use a lambda expression to point to a specific function. Here we are referring to the checkEmp() function and the Returns parameter value is true.

This means that whenever the unit test application encounters the checkEmp() function it will always return true without executing it's code. So, ultimately, it will not execute at all and the result will be always true.

**NUnitTesting**

In this IT world a unit refers to simply a smallest piece of code which takes an input, does certain operation, and gives an output.

And testing this small piece of code is called Unit Testing.

A lot of unit test frameworks are available for .Net nowadays, if we check in Visual Studio we have MSTest from Microsoft integrated in Visual Studio.

Some 3rd party frameworks are:

NUnit

MbUnit

Out of all these Nunit is the most-used testing Framework.

Required DLL

1. NUnit.dll
2. NUnit3TestAdapter (Transient is a single instance per code request.)

**[TestFixture]**

The class that is to be tested using Nunit should be decorated with TextFixture.

**[Test]**

This attribute identifies the method to be tested. If we do not write this attribute then we can't identify the test in Test Explorer. We have an Assert class with the following methods for validating different conditions in the TestFixture.

**[SetUp]**

*Method decorated with SetUp will be execute first before each test execution.*

This attribute is used when you want to execute a piece of code in each test case. It identifies a method to be executed each time before a TestMethod/Test is executed.

*If two SetUp classes are there the class that was written first will be executed first and then after that, the class that is written next to it will execute. (Top to bottom approach).*

**[TearDown]**

*The method decorated with TearDown will be execute after each test are executed.*

*Similarly, in TearDown the order is reversed -- it will follow bottom to top approach.*