**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.

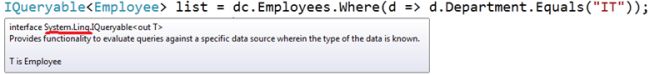
namespace ConsoleApplication5  
  {  
    class Program  
        {  
            static void Main(string[] args)  
                {  
                    DataClasses1DataContext context = new DataClasses1DataContext();  
                    var result = MyCompliedQueries.CompliedQueryForPesron(context);  
                    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 );  
   
           
        }  
}

**10) IQueryable vs. IEnumerable**

**IEnumerable**

1. IEnumerable exists in the System.Collections namespace.  
     
   
2. The IEnumerable version signature is: Where(Func<Customer, bool> predicate)
3. IEnumerable is suitable for querying data from in-memory collections like List, Array and so on.
4. While querying data from the database, IEnumerable executes "select query" on the server-side, loads data in-memory on the client-side and then filters the data.   
     
   
5. IEnumerable is beneficial for LINQ to Object and LINQ to XML queries.
6. If you want repeated filtering on your original result (several end results). Doing that on the IQueryable interface will make several roundtrips to the database, where as doing it on IEnumerable will do the filtering in the memory, making it faster (unless the amount of data is HUGE)

**IQueryable**

1. IQueryable exists in the System.Linq Namespace.  
     
   
2. The IQueryable version signature is: Where(Expression<Func<Customer, bool>> predicate)
3. IQueryable is suitable for querying data from out-memory (like remote database, service) collections.
4. While querying data from a database, IQueryable executes a "select query" on server-side with all filters.  
     
   https://csharpcorner-mindcrackerinc.netdna-ssl.com/UploadFile/a20beb/ienumerable-vs-iqueryable-in-linq/Images/IEnumerable%20vs%20IQueryable%20in%20LINQ10.jpg
5. IQueryable is beneficial for LINQ to SQL queries.
6. Working on [IQueryable<T>](https://msdn.microsoft.com/en-us/library/bb351562.aspx) can in many cases save you from returning too many rows from the database. Another prime example is doing paging: If you use [Take](https://msdn.microsoft.com/en-us/library/bb300906.aspx) and [Skip](https://msdn.microsoft.com/en-us/library/bb357513.aspx) on [IQueryable](https://msdn.microsoft.com/en-us/library/system.linq.iqueryable.aspx), you will only get the number of rows requested; doing that on an [IEnumerable<T>](https://msdn.microsoft.com/en-us/library/9eekhta0.aspx) will cause all of your rows to be loaded in memory
7. The major difference is that IEnumerable will enumerate all elements, while IQueryable will enumerate elements (or even do other things) based on a query. In the case of the IQueryable, the LINQ query gets used by IQueryProvider which must be interpreted or compiled in order to get the result. I.e., the extension methods defined for IQueryable take Expression objects instead of Func objects (which is what IEnumerable uses), meaning the delegate it receives is an expression tree instead of a method to invoke.
8. IEnumerable is great for working with in-memory collections, but IQueryable<t> allows for a remote data source, like a database or web service.

**11) Joins in LINQ**

CREATE TABLE COURSE  
(  
     course\_id int IDENTITY(1,1) PRIMARY KEY,  
     course\_name nvarchar(70) NOT NULL,  
     course\_desc nvarchar(255) NULL,  
     modified\_date date NULL,  
)  
CREATE TABLE STUDENT  
(  
     student\_id int IDENTITY(1,1) PRIMARY KEY,  
     student\_name nvarchar(70),  
     student\_city nvarchar(30),  
     course\_id int NOT NULL  
)

ALTER TABLE STUDENT  
ADD CONSTRAINT [FK\_STUDENT\_COURSE] FOREIGN KEY (course\_id) REFERENCES COURSE(course\_id)

**Cross Join in LINQ to SQL**   
var studentInfo = from student in STUDENTs  
from course in COURSEs  
select new { student.student\_name, student.student\_city, course.course\_name, course.course\_desc };

**Inner Join in LINQ to SQL**  
var studentInfo = from student in STUDENTs  
join course in COURSEs  
on student.course\_id equals course.course\_id  
select new {student.student\_name, student.student\_city, course.course\_name, course.course\_desc};

**Left Join in LINQ to SQL**   
var courseInfo = from course in COURSEs  
join student in STUDENTs  
on course.course\_id equals student.course\_id into studentInfo  
from students in studentInfo.DefaultIfEmpty()  
select new  
{  
     STUDENTNAME = (students.student\_name == null) ? "NULL" : students.student\_name,  
     STUDENTCITY = (students.student\_city == null) ? "NULL" : students.student\_city,  
     COURSENAME = course.course\_name,  
     COUSREDESCRIPTION = course.course\_desc  
};

class Person

{

public string FirstName { get; set; }

public string LastName { get; set; }

}

class Pet

{

public string Name { get; set; }

public Person Owner { get; set; }

}

public static void LeftOuterJoinExample()

{

Person magnus = new Person { FirstName = "Magnus", LastName = "Hedlund" };

Person terry = new Person { FirstName = "Terry", LastName = "Adams" };

Person charlotte = new Person { FirstName = "Charlotte", LastName = "Weiss" };

Person arlene = new Person { FirstName = "Arlene", LastName = "Huff" };

Pet barley = new Pet { Name = "Barley", Owner = terry };

Pet boots = new Pet { Name = "Boots", Owner = terry };

Pet whiskers = new Pet { Name = "Whiskers", Owner = charlotte };

Pet bluemoon = new Pet { Name = "Blue Moon", Owner = terry };

Pet daisy = new Pet { Name = "Daisy", Owner = magnus };

// Create two lists.

List<Person> people = new List<Person> { magnus, terry, charlotte, arlene };

List<Pet> pets = new List<Pet> { barley, boots, whiskers, bluemoon, daisy };

var query = from person in people

join pet in pets on person equals pet.Owner into gj

from subpet in gj.DefaultIfEmpty()

select new { person.FirstName, PetName = subpet?.Name ?? String.Empty };

foreach (var v in query)

{

Console.WriteLine($"{v.FirstName+":",-15}{v.PetName}");

}

}

// This code produces the following output:

//

// Magnus: Daisy

// Terry: Barley

// Terry: Boots

// Terry: Blue Moon

// Charlotte: Whiskers

// Arlene: