LINQ & Entity Framework Core

Lesson 01 : Introduction to LINQ and LINQ to Objects

Capgemini

Lesson Objectives

- In this lesson we will cover the following
  - Overview of LINQ
  - LINQ Architecture and Components
  - Basic Query Operations with LINQ
  - Introduction to DataSet & DataTable

Overview

- LINQ stands for Language Integrated Query
- It was introduced from .NET Framework 3.5 and continued as a part of .NET Framework 4.0 and 4.5
- It bridges the gap between the world of objects and the world of data
- It adds Query capabilities to the Programming Language
- LINQ enables you to query data from within the .NET Programming Language in the same way the SQL enables your to query data from a database.

Overview

- Traditionally , queries against data was a simple string without type checking at compile time and IntelliSense support , which used to result in runtime exceptions
- Unlike the traditional query LINQ query uses language construct and uses features like compile time check and IntelliSense

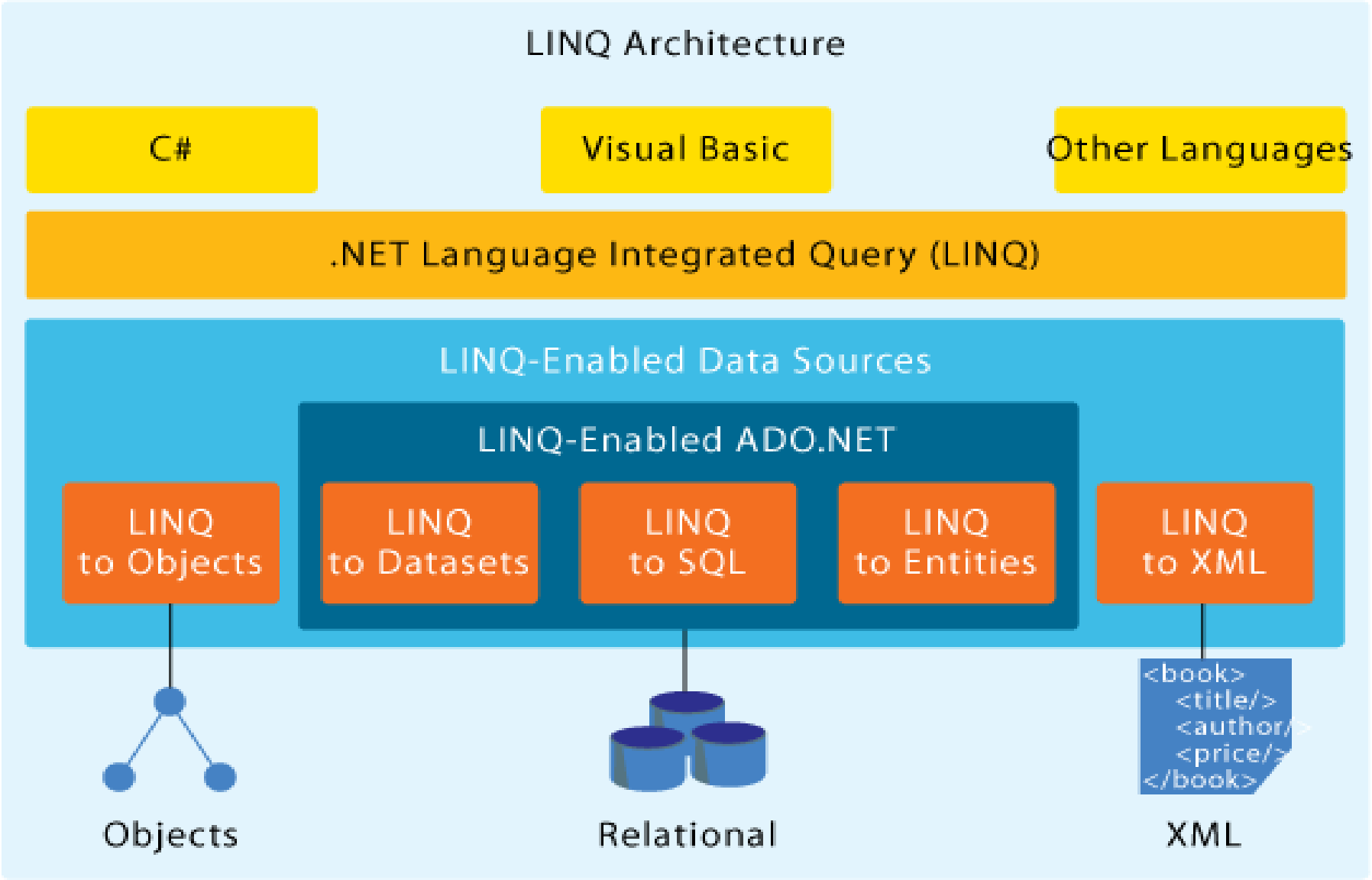
Overview

- Benefit of using LINQ is that one can:
  - work with data in a consistent way, regardless of the type of data
  - interact with data as objects
  - integrate better with programming languages
  - improve productivity through IntelliSense in Visual Studio

Overview

- The design goals for LINQ are:
  - to integrate objects, relational data, and XML
  - to provide SQL and XQuery-like power in C# and VB
  - to provide Extensibility model for languages
  - to provide Extensibility model for multiple data sources
  - to provide Type safety
  - to provide Extensive IntelliSense support (enabled by strong-typing)
  - to provide Debugger support

LINQ Architecture and Component



LINQ Architecture and Components

- Different Flavors of LINQ
  - LINQ to Objects
  - LINQ to ADO.NET
    - LINQ to SQL
    - LINQ to Datasets
    - LINQ to Entities
  - LINQ to XML
  - Parallel LINQ(PLINQ)

Basic Query Operations with LINQ

- LINQ Queries are written using the LINQ declarative query syntax.
- These queries uses a set of query keywords built into the .NET Framework that allows the developer to write SQL like commands in programming language.
- Commonly used Keywords are
  - from / in
  - where
  - orderby
  - select
  - groupby

Basic Query Operations with LINQ

- All LINQ query operations require the following three distinct actions:
  - Obtain the data source
  - Create the query
  - Execute the query

Example

```
class IntrotoLINQ
{
   static void Main()
   {
         //1.Data Source
         int [] numbers = new int[7]{0,1,2,3,4,5,6};

         //2.Query Creation
         var numQuery= from num in numbers
                       where (num%2)==0
                       select num;
         //3.Query Execution
         foreach(int num in numQuery)
         {Console.Write("{0,1}",num);}
   }
}
```

Basic Query Operations with LINQ

- Syntax

```
from <range variable> in <collection>

<filter, joining, grouping, aggregate operators etc.> <lambda expression>

<select or groupBy operator> <formulate the result>
```

Basic Query Operations with LINQ

- Code Snippet

```
static void Main(string[] args)
{
    int[] numbers = { 10, 22, 98, 76, 81, 99, 181, 71, 31 };

    var result = from number in numbers
                 where number % 2 == 0
                 select number;

}
```

Demo

- Demo of Implementing Simple LINQ Queries

Basic Query Operators LINQ

- LINQ provide some standard query operator that can be used to query data
- Following is the categories of operator in LINQ
  - Filtering Operators
  - Projection Operators
  - Sorting Operators
  - Aggregation
  - Grouping Operators
  - Conversions

Filtering Operator

- Where Operator:-
  - It is filteriing opertor
  - It filter a sequence based on a predicate function

Projection Operator

- Select operator:
  - It is a projection operator
  - It enumerates the source sequence, and yields the results of evaluating the selector function for each element

- SelectMany:
  - It is a projection operator
  - It performs a one-to-many element projection over a sequence

Sorting Operator

- OrderBy and OrderByDescending:
  - The OrderBy and OrderByDescending operators order elements of a sequence according to a given key
  - The OrderByDescending operator inverts the ordering.

- ThenBy and ThenByDescending:
  - These operators are useful for specifying additional ordering keys after the first one is specified either by the OrderBy or OrderByDescending operator
  - ThenByDescending is similar to ThenBy. However, it inversely sorts the sequence

- Reverse:
  - This operator returns a new sequence having elements in a reverse ordering of the source sequence

Grouping Operators

- GroupBy:
  - This operator groups the elements of a sequence.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

var query = numbers.ToLookup(i => i % 2);

foreach (IGrouping<int, int> group in query)
{
    Console.WriteLine("Key:  {0}", group.Key);
    foreach (int number in group)
    {
        Console.WriteLine(number);
    }
}
```

Concatenation Operator

- Concat:
  - This operator concatenates two sequences

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int[] moreNumbers = { 10, 11, 12, 13 };
var query = numbers.Concat(moreNumbers);
foreach(var item in query)
Console.WriteLine(item);
```

Set Operator

- Distinct:
  - This operator eliminates duplicate elements from a sequence
- Except:
  - This operator enumerates the first sequence, collecting all distinct elements. Subsequently, it enumerates the second sequence, thus removing elements contained in the first sequence
- Intersect:
  - The operator enumerates the first sequence, collecting all distinct elements
  - It then enumerates the second sequence, yielding elements that occur in both sequences
- Union:
  - The operator produces a set union of two sequences

Conversion Operators

- AsEnumerable:
  - This operator returns its argument typed as IEnumerable<T>
- OfType:
  - This operator filters the elements of a sequence based on a type
- ToArray:
  - This operator creates an array from a sequence

Aggregate Operators

- Aggregate:
  - The operator applies a function over a sequence

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var query = numbers.Aggregate((a, b) => a * b);
Console.WriteLine(query);
```

- Average:
  - The operator computes the average of a sequence of numeric values Count & LongCount
  - It counts the number of elements in a sequence

Aggregate Operator (Contd..)

- Max:
  - The operator finds the maximum value from a sequence of numeric values
- Min:
  - The operator finds the minimum value from a sequence of numeric values
- Sum:
  - The operator computes the sum of a sequence of numeric values

Element Operators

- First:
  - The operator returns the first element of a sequence
- FirstOrDefault:
  - The operator returns the first element of a sequence, or a default value if no element is found
- Last:
  - The operator returns the last element of a sequence
- LastOrDefault:
  - The operator returns the last element of a sequence, or a default value if no element is found

Element Operators (Contd..)

- Single:
  - The operator returns the single element of a sequence. An exception is thrown if the source sequence contains no match or more than one match

- SingleOrDefault:
  - The operator returns the single element of a sequence, or a default value if no element is found. The default value is for reference and nullable types

Element Operators (Contd..)

- DefaultIfEmpty:
  - The operator supplies a default element for an empty sequence. It can be combined with a grouping join to produce a left outer join
- ElementAt:
  - It returns the element at a given index in a sequence
- ElementAtOrDefault:
  - The operator returns the element at a given index in a sequence, or a default value if the index is out of range

Example

- Example 1:

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var query = numbers.First();
Console.WriteLine("The first element in the sequence");
Console.WriteLine(query);
query = numbers.Last();
Console.WriteLine("The last element in the sequence");
Console.WriteLine(query);
Console.WriteLine("The first even element in the sequence");
query = numbers.First(n => n % 2 == 0);
Console.WriteLine(query);
Console.WriteLine("The last even element in the sequence");
query = numbers.Last(n => n % 2 == 0);
Console.WriteLine(query);
```

Example

- Example 2: FirstOrDefault and LastOrDefault

```
int[] numbers = {1, 3, 5, 7, 9};
var query = numbers.FirstOrDefault(n => n % 2 == 0);
Console.WriteLine("The first even element in the sequence");
Console.WriteLine(query);
Console.WriteLine("The last odd element in the sequence");
query = numbers.LastOrDefault(n => n % 2 == 1);
Console.WriteLine(query);
```

Example

- Example 3: Single and SingleOrDefault

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.Single(n => n > 8);
Console.WriteLine(query);

int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.SingleOrDefault(n => n > 9);
Console.WriteLine(query
```

Example

- Example 4: ElementAt and ElementAtOrDefault

```
int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.ElementAt(4);
Console.WriteLine(query);

int[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
var query = numbers.ElementAtOrDefault(9);
Console.WriteLine(query);
```

Deferred Execution vs Immediate Execution

- The LINQ queries are executed in two different ways as follows.
  - Deferred execution
  - Immediate execution
- In Deferred execution, LINQ Query is not executed at the point of its declaration.
- When we write a LINQ query, it doesn't execute by itself. It executes only when we access the query results.
- Execution of the query is deferred until the query variable is iterated over using for each loop.
- Operators used for deferred execution are called **Deferred or Lazy** Operator. Ex: select, SelectMany, where, Take, Skip, etc..
- In Immediate execution, LINQ Query is executed at the point of its declaration. It forces the query to execute and gets the results immediately.
- Operators used for immediate execution are called **immediate or Greedy** Operator. Ex: count, average, min, max, First, Last, ToArray, ToList, etc..

Example

Deferred Execution:

```
List<int> intlist = new List<int> { 3, 6, 8, 1, 9, 4, 2 };
var result = from data in intlist
             where data > 5
             select data;

intlist.Add(7);
//Notice new value got added after query variable created

foreach (int info in result)
{
    Console.WriteLine(info);
}
```

Query Variable

Output: 6 8 9 7

Immediate Execution:

```
List<int> intlist = new List<int> { 3, 6, 8, 1, 9, 4, 2 };
var count = (from data in intlist
             where data > 5
             select data).Count();

intlist.Add(7);
//Notice new value got added after query variable created

Console.Write("Total Numbers greater than 5 is: " + count );
```

Immediate Execution

Output: Total Numbers greater than 5 is: 3

let Keyword

- In order to define a variable inside a linq expression, you can use the **let** keyword.
- This is usually done in order to store the results of intermediate sub-queries or it gives you the liberty to use its value for the next level.
- These variables can be used anywhere in the query ,after it is declared which helps to write complex queries in simple manner.
- Syntax:

  sub expression1

  let letVariable = query expression which returns a value

  sub expression2 which uses the letVariable

Understanding LINQ to DataSet

- .New LINQ Provider – LINQ to DataSet
  - .NET 3.5 comes with many LINQ providers out-of-the-box and one of them is LINQ to DataSets
- Performing LINQ Queries on DataSet & DataTable
  - LINQ to Dataset means performing a LINQ query operations on Dataset or DataTable
- Used to Perform LINQ Operations on ADO.NET Disconnected Architecture
- LINQ to DataSet is also know as LINQ to DataTable
  - Writing LINQ queries on records available in DataSet's DataTable object.

Implementing LINQ to DataSet

- Call AsEnumerable() on DataTable to Access LINQ API
  - To support LINQ on DataTable, we have to call AsEnumerable() method on the DataTable, which returns an object implementing IEnumerable<T> interface.
- Write LINQ Query on IEnumerable<DataRow> using Field<T>()
  - Calling Field<T>() on DataRow element provides strongly-typed access to each of the column values in specified row.
- Can use CopyToDataTable() to convert LINQ result into DataTable
  - To convert the IEnumerable<DataRow> back to the DataTable we can use CopyToDataTable() method.

LINQ to DataSet – Sample Code

```
//Obtaining the data source on which to write LINQ
var empDataSource = GetDataSet().Tables["Emp"].AsEnumerable();
//LINQ query to get employees having salary greater than 38000
var query = from r in empDataSource
            where r.Field<decimal>("Salary") > 38000
            orderby r.Field<decimal>("Salary")
            select r;
//executing the query to retrive the LINQ result
foreach (var row in query)
{
    string str = $"EmpId: {row["EmpId"]}, Name: {row["EmpName"]},
                  Salary: {row["Salary"]}";
    Console.WriteLine(str);
}
//Loading the query results into DataTable object.
DataTable table = query.CopyToDataTable();
```

Demo

- Demo of Implementing standard operator in LINQ
- Demo of LINQ to DataTable

Lab

- Lab Topic

Summary

- In this lesson we have learnt the following topic
  - Overview of LINQ
  - LINQ Architecture and Components
  - Basic Query Operation using LINQ
  - Deferred Execution vs Immediate Execution
  - let Keyword
  - LINQ to DataSet & DataTable

Review Question

- Which of the following version of .NET framework introduced LINQ?
  - .NET Framework 2.0
  - .NET Framework 3.5
  - .NET Framework 3.5 SP1
  - .NET Framework 3.0

- LINQ uses programming language syntax to define queries
  - True
  - False

Review Question

- LINQ stands for _____
  - Language in Query
  - Language Integrated Query
  - Language Independent Query
  - Language Include Query