

LINQ - Overview

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 LanguageIntegratedQueryLanguageIntegratedQuery 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

C#

```
using System;
using System.Linq;

class Program
{
    static void Main()
    {
        string[] words = {"hello", "wonderful", "LINQ", "beautiful", "world"};
        //Get only short words
        var shortWords = from word in words
                        where word.Length <= 5
                        select word;

        //Print each word out
        foreach (var word in shortWords)
        {
            Console.WriteLine(word);
        }
        Console.ReadLine();
    }
}
```

When the above code of C# is compiled and executed, it produces the following result:

```
hello
LINQ
world
```

Syntax of LINQ

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

- **Lamda Method Syntax**

Example

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

- **Query Comprehension Syntax**

Example

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

Types of LINQ

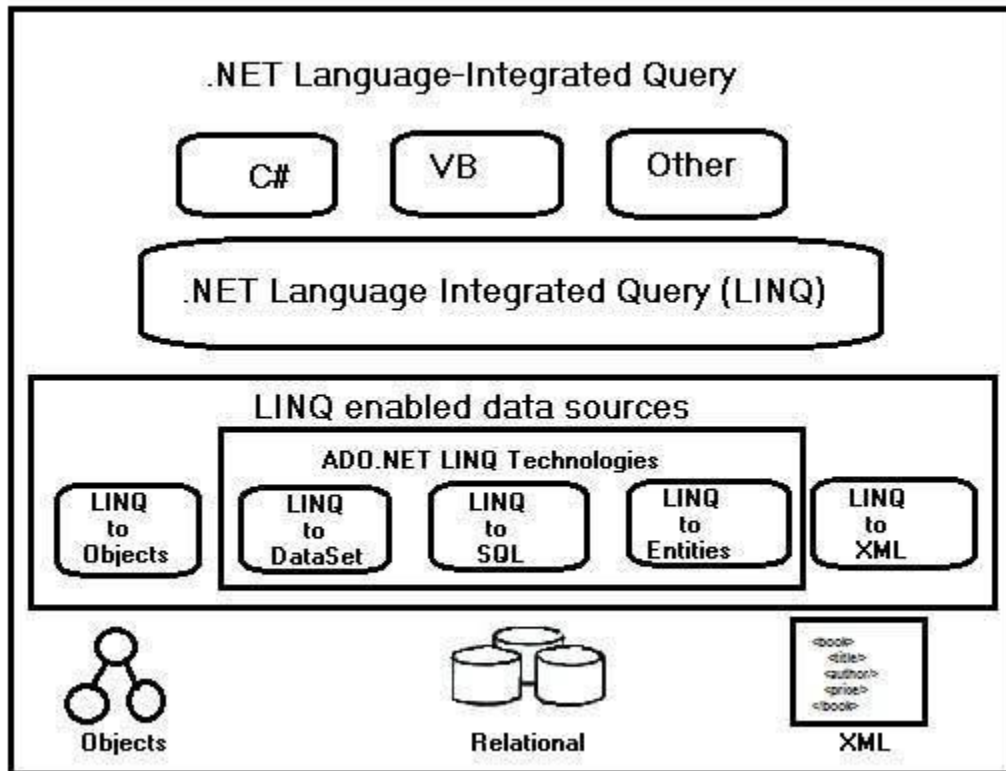
The types of LINQ are mentioned below in brief.

- LINQ to Objects
- LINQ to XML
- LINQ to DataSet
- LINQ to SQL
- LINQ to Entities

Apart from the above, there is also a LINQ type named PLINQ which is Microsoft's parallel LINQ.

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 in Figure.



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.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Operators
{
```

```

class LINQQueryExpressions
{
    static void Main()
    {
        // Specify the data source.
        int[] scores = new int[] { 97, 92, 81, 60 };

        // Define the query expression.
        IEnumerable<int> scoreQuery = from score in scores
                                     where score > 80
                                     select score;

        // Execute the query.
        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
```

Extension Methods

Introduced with .NET 3.5, Extension methods are declared in static classes only and allow inclusion of custom methods to objects to perform some precise query operations to extend a class without being an actual member of that class. These can be overloaded also.

In a nutshell, extension methods are used to translate query expressions into traditional method calls object oriented.

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 easy 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 .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 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 a data before the advent of LINQ.

```
SqlConnection sqlConnection = new SqlConnection(connectionString);
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, query gets defined only by the last two. Using LINQ, the same data query can be written in a readable color-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 relationship 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 unitive foundation.**
- LINQ is extensible that means it is possible to use knowledge of LINQ to querying 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.**

LINQ - Query Operators

A set of extension methods forming a query pattern is known as LINQ Standard Query Operators. As building blocks of LINQ query expressions, these operators offer a range of query capabilities like filtering, sorting, projection, aggregation, etc.

LINQ standard query operators can be categorized into the following ones on the basis of their functionality.

- Filtering Operators
- Join Operators
- Projection Operations
- Sorting Operators
- Grouping Operators
- Conversions
- Concatenation
- Aggregation
- Quantifier Operations
- Partition Operations
- Generation Operations
- Set Operations
- Equality
- Element Operators

Filtering Operators

Filtering is an operation to restrict the result set such that it has only selected elements satisfying a particular condition.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Where	Filter values based on a predicate function	where	Where
OfType	Filter values based on their ability to be as a specified type	Not Applicable	Not Applicable

Join Operators

Joining refers to an operation in which data sources with difficult to follow relationships with each other in a direct way are targeted.

```
using System;
using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] words = { "humpty", "dumpty", "set", "on", "a", "wall" };

            IEnumerable<string> query = from word in words
                                      where word.Length == 3
                                      select word;
```

```

        foreach (string str in query)
        {
            Console.WriteLine(str);
            Console.ReadLine();
        }
    }
}

```

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Join	The operator join two sequences on basis of matching keys	join ... in ... on ... equals ...	From x In ... y In ... Where x.a = y.a
GroupJoin	Join two sequences and group the matching elements	join ... in ... on ... equals ... into ...	Group Join ... In ... On ...

Projection Operations

Projection is an operation in which an object is transformed into an altogether new form with only specific properties.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Select	The operator projects values on basis of a transform function	select	Select
SelectMany	The operator project the sequences of values which are based on a transform function as well as flattens them into a single sequence	Use multiple from clauses	Use multiple From clauses

Sorting Operators

A sorting operation allows ordering the elements of a sequence on basis of a single or more attributes.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
OrderBy	The operator sort values in an ascending order	orderby	Order By
OrderByDescending	The operator sort values in a descending order	orderby ... descending	Order By ... Descending
ThenBy	Executes a secondary sorting in an ascending order	orderby ..., ...	Order By ..., ...
ThenByDescending	Executes a secondary sorting in a descending order	orderby ..., ... descending	Order By ..., ... Descending
Reverse	Performs a reversal of the order of the elements in a collection	Not Applicable	Not Applicable

Grouping Operators

The operators put data into some groups based on a common shared attribute.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
GroupBy	Organize a sequence of items in groups and return them as an IEnumerable collection of type IGrouping<key, element>	group ... by -or- group ... by ... into ...	Group ... By ... Into ...
ToLookup	Execute a grouping operation in which a sequence of key pairs are returned	Not Applicable	Not Applicable

Conversions

The operators change the type of input objects and are used in a diverse range of applications.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
AsEnumerable	Returns the input typed as IEnumerable<T>	Not Applicable	Not Applicable
AsQueryable	A generic IEnumerable is converted to a generic IQueryable	Not Applicable	Not Applicable
Cast	Performs casting of elements of a collection to a specified type	Use an explicitly typed range variable. Eg: from string str in words	From ... As ...
OfType	Filters values on basis of their , depending on their capability to be cast to a particular type	Not Applicable	Not Applicable
ToArray	Forces query execution and does conversion of a collection to an array	Not Applicable	Not Applicable
ToDictionary	On basis of a key selector function set elements into a Dictionary<TKey, TValue> and forces execution of a LINQ query	Not Applicable	Not Applicable
ToList	Forces execution of a query by converting a collection to a List<T>	Not Applicable	Not Applicable
ToLookup	Forces execution of a query and put elements into a Lookup<TKey, TElement> on basis of a key selector function	Not Applicable	Not Applicable

Concatenation

Performs concatenation of two sequences and is quite similar to the Union operator in terms of its operation except of the fact that this does not remove duplicates.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Concat	Two sequences are concatenated for the formation of a single one sequence.	Not Applicable	Not Applicable

Aggregation

Performs any type of desired aggregation and allows creating custom aggregations in LINQ.

[Show Examples](#)

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Aggregate	Operates on the values of a collection to perform custom aggregation operation	Not Applicable	Not Applicable
Average	Average value of a collection of values is calculated	Not Applicable	Aggregate ... In ... Into Average
Count	Counts the elements satisfying a predicate function within collection	Not Applicable	Aggregate ... In ... Into Count
LongCount	Counts the elements satisfying a predicate function within a huge collection	Not Applicable	Aggregate ... In ... Into LongCount
Max	Find out the maximum value within a collection	Not Applicable	Aggregate ... In ... Into Max
Min	Find out the minimum value existing within a collection	Not Applicable	Aggregate ... In ... Into Min
Sum	Find out the sum of a values within a collection	Not Applicable	Aggregate ... In ... Into Sum

Quantifier Operations

These operators return a Boolean value i.e. True or False when some or all elements within a sequence satisfy a specific condition.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
All	Returns a value 'True' if all elements of a sequence satisfy a predicate condition	Not Applicable	Aggregate ... In ... Into All.....
Any	Determines by searching a sequence that whether any element of the same satisfy a specified condition	Not Applicable	Aggregate ... In ... Into Any
Contains	Returns a 'True' value if finds that a specific element is there in a sequence if the sequence does not contain that specific element, 'false' value is returned	Not Applicable	Not Applicable

Partition Operators

Divide an input sequence into two separate sections without rearranging the elements of the sequence and then returning one of them.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Skip	Skips some specified number of elements within a sequence and returns the remaining ones	Not Applicable	Skip
SkipWhile	Same as that of Skip with the only exception that number of elements to skip are specified by a Boolean condition	Not Applicable	Skip While
Take	Take a specified number of elements from a sequence and skip the remaining ones	Not Applicable	Take
TakeWhile	Same as that of Take except the fact that number of elements to take are specified by a Boolean condition	Not Applicable	Take While

Generation Operations

A new sequence of values is created by generational operators.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
DefaultIfEmpty	When applied to an empty sequence, generate a default element within a sequence	Not Applicable	Not Applicable
Empty	Returns an empty sequence of values and is the most simplest generational operator	Not Applicable	Not Applicable
Range	Generates a collection having a sequence of integers or numbers	Not Applicable	Not Applicable
Repeat	Generates a sequence containing repeated values of a specific length	Not Applicable	Not Applicable

Set Operations

There are four operators for the set operations, each yielding a result based on different criteria.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
Distinct	Results a list of unique values from a collection by filtering duplicate data if any	Not Applicable	Distinct
Except	Compares the values of two collections and return the ones from one collection who are not in the other collection	Not Applicable	Not Applicable
Intersect	Returns the set of values found to be identical in two separate collections	Not Applicable	Not Applicable
Union	Combines content of two different collections into a single list that too without any duplicate content	Not Applicable	Not Applicable

Equality

Compares two sequences enumerable and determine are they an exact match or not.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
SequenceEqual	Results a Boolean value if two sequences are found to be identical to each other	Not Applicable	Not Applicable

Element Operators

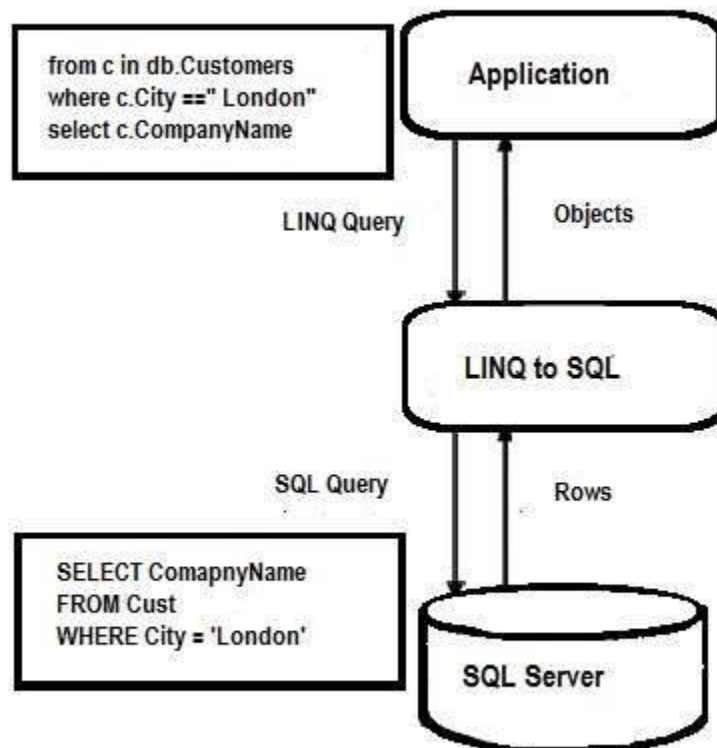
Except the DefaultIfEmpty, all the rest eight standard query element operators return a single element from a collection.

Operator	Description	C# Query Expression Syntax	VB Query Expression Syntax
ElementAt	Returns an element present within a specific index in a collection	Not Applicable	Not Applicable
ElementAtOrDefault	Same as ElementAt except of the fact that it also returns a default value in case the specific index is out of range	Not Applicable	Not Applicable
First	Retrieves the first element within a collection or the first element satisfying a specific condition	Not Applicable	Not Applicable
FirstOrDefault	Same as First except the fact that it also returns a default value in case there is no existence of such elements	Not Applicable	Not Applicable
Last	Retrieves the last element present in a collection or the last element satisfying a specific condition	Not Applicable	Not Applicable
LastOrDefault	Same as Last except the fact that it also returns a default value in case there is no existence of any such element	Not Applicable	Not Applicable
Single	Returns the lone element of a collection or the lone element that satisfy a certain condition	Not Applicable	Not Applicable

SingleOrDefault	Same as Single except that it also returns a default value if there is no existence of any such lone element	Not Applicable	Not Applicable
DefaultIfEmpty	Returns a default value if the collection or list is empty or null	Not Applicable	Not Applicable

Introduction of LINQ To SQL

- For most ASP.NET developers, LINQ to SQL (also known as DLINQ) is an electrifying part of Language Integrated Query as this allows querying data in SQL server database by using usual LINQ expressions. It also allows to update, delete and insert data, but the only drawback from which it suffers is its limitation to the SQL server database. However, there are many benefits of LINQ to SQL over ADO.NET like reduced complexity, few lines of coding and many more.
- Below is a diagram showing the execution architecture of LINQ to SQL.



A Dataset offers an extremely useful data representation in memory and is used for a diverse range of data based applications. LINQ to Dataset as one of the technology of LINQ to ADO.NET facilitates performing queries on the data of a Dataset in a hassle-free manner and enhance productivity.

LINQ - Objects

LINQ to Objects offers usage of any LINQ query supporting `IEnumerable<T>` for accessing in-memory data collections without any need of LINQ provider API as in case of LINQ to SQL or LINQ to XML.

Introduction of LINQ To Objects

Queries in LINQ to Objects return variables of type usually `IEnumerable<T>` only. In short, LINQ to Objects offers a fresh approach to collections as earlier, it was vital to write long coding `foreach` loops of much complexity for retrieval of data from a collection which is now replaced by writing declarative code which clearly describes the desired data that is required to retrieve.

There are also many advantages of LINQ to Objects over traditional `foreach` loops like more readability, powerful filtering, capability of grouping, enhanced ordering with minimal application coding. Such LINQ queries are also more compact in nature and are portable to any other data sources without any modification or with just a little modification.

Below is a simple LINQ to Objects example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace LINQtoObjects
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] tools = { "Tablesaw", "Bandsaw", "Planer", "Jointer",
"Drill",
                           "Sander" };
            var list = from t in tools
                      select t;

            StringBuilder sb = new StringBuilder();
```



```

        foreach (string s in list)
        {
            sb.Append(s + Environment.NewLine);
        }
        Console.WriteLine(sb.ToString(), "Tools");
        Console.ReadLine();
    }
}

```

In the example, an array of strings tooltools is used as the collection of objects to be queried using LINQ to Objects.

Objects query is:

```

var list = from t in tools
            select t;

```

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

```

Tablesaw
Bandsaw
Planer
Jointer
Drill
Sander

```

Querying in Memory Collections using LINQ to Objects

C#

```

using System;
using System.Collections.Generic;
using System.Linq;

namespace LINQtoObjects
{
    class Department
    {
        public int DepartmentId { get; set; }
        public string Name { get; set; }
    }

    class LinqToObjects
    {
        static void Main(string[] args)
        {
            List<Department> departments = new List<Department>();
            departments.Add(new Department { DepartmentId = 1, Name = "Account"
});
            departments.Add(new Department { DepartmentId = 2, Name = "Sales" });
            departments.Add(new Department { DepartmentId = 3, Name = "Marketing"
});
        }
    }
}

```

```

var departmentList = from d in departments
                      select d;

foreach (var dept in departmentList)
{
    Console.WriteLine("Department Id = {0} , Department Name = {1}",
                      dept.DepartmentId, dept.Name);
}
Console.WriteLine("\nPress any key to continue.");
Console.ReadKey();
}
}
}

```

When the above code of C# is compiled and executed, it produces the following result:

```

Department Id = 1, Department Name = Account
Department Id = 2, Department Name = Sales
Department Id = 3, Department Name = Marketing

Press any key to continue.

```

Introduction of LINQ To Dataset

LINQ to Dataset has made the task of querying simple for the developers. They don't need to write queries in a specific query language instead the same can be written in programming language. LINQ to Dataset is also usable for querying where data is consolidated from multiple data sources. This also does not need any LINQ provider just like LINQ to SQL and LINQ to XML for accessing data from in memory collections.

Below is a simple example of a LINQ to Dataset query in which a data source is first obtained and then the dataset is filled with two data tables. A relationship is established between both the tables and a LINQ query is created against both tables by the means of join clause. Finally, foreach loop is used to display the desired results.

C#

```

using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LINQtoDataset
{

```

```

class Program
{
    static void Main(string[] args)
    {
        string connectionString =
System.Configuration.ConfigurationManager.ConnectionStrings["LinqToSQLDBConne
ctionString"].ToString();

        string sqlSelect = "SELECT * FROM Department;" +
                            "SELECT * FROM Employee;";

        // Create the data adapter to retrieve data from the database
        SqlDataAdapter da = new SqlDataAdapter(sqlSelect, connectionString);

        // Create table mappings
        da.TableMappings.Add("Table", "Department");
        da.TableMappings.Add("Table1", "Employee");

        // Create and fill the DataSet
        DataSet ds = new DataSet();
        da.Fill(ds);

        DataRelation dr = ds.Relations.Add("FK_Employee_Department",
            ds.Tables["Department"].Columns["DepartmentId"],
            ds.Tables["Employee"].Columns["DepartmentId"]);

        DataTable department = ds.Tables["Department"];
        DataTable employee = ds.Tables["Employee"];

        var query = from d in department.AsEnumerable()
                    join e in employee.AsEnumerable()
                    on d.Field<int>("DepartmentId") equals
                    e.Field<int>("DepartmentId")
                    select new
                    {
                        EmployeeId = e.Field<int>("EmployeeId"),
                        Name = e.Field<string>("Name"),
                        DepartmentId = d.Field<int>("DepartmentId"),
                        DepartmentName = d.Field<string>("Name")
                    };

        foreach (var q in query)
        {
            Console.WriteLine("Employee Id = {0} , Name = {1} , Department
Name = {2}",
                                q.EmployeeId, q.Name, q.DepartmentName);
        }

        Console.WriteLine("\nPress any key to continue.");
        Console.ReadKey();
    }
}

```

LINQ - XML

LINQ to XML offers easy accessibility to all LINQ functionalities like standard query operators, programming interface, etc. Integrated in the .NET framework, LINQ to XML also makes the best use of .NET framework functionalities like debugging, compile-time checking, strong typing and many more to say.

Introduction of LINQ to XML

While using LINQ to XML, loading XML documents into memory is easy and more easier is quering and document modification. It is also possible to save XML documents existing in memory to disk and to serialize them. It eliminates the need for a developer to learn the XML query language which is somewhat complex.

LINQ to XML has its power in the System.Xml.Linq namespace. This has all the 19 necessary classes to work with XML. These classes are the following ones.

- XAttribute
- XCDATA
- XComment
- XContainer
- XDeclaration
- XDocument
- XDocumentType
- XElement
- XName
- XNamespace
- XNode
- XNodeDocumentOrderComparer
- XNodeEqualityComparer
- XObject
- XObjectChange
- XObjectChangeEventArgs
- XObjectEventHandler
- XProcessingInstruction
- XText

Read the XML File using LINQ

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace LINQtoXML
{
    class ExampleOfXML
    {
        static void Main(string[] args)
        {
            string myXML = @"<Departments>
                            <Department>Account</Department>
                            <Department>Sales</Department>
                            <Department>Pre-Sales</Department>
                            <Department>Marketing</Department>
                            </Departments>";

            XDocument xdoc = new XDocument();
            xdoc = XDocument.Parse(myXML);

            var result = xdoc.Element("Departments").Descendants();

            foreach (XElement item in result)
            {
                Console.WriteLine("Department Name - " + item.Value);
            }

            Console.WriteLine("\nPress any key to continue.");
            Console.ReadKey();
        }
    }
}
```

Add New Node

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace LINQtoXML
{
    class ExampleOfXML
    {
        static void Main(string[] args)
        {
```

```

        string myXML = @"<Departments>
                        <Department>Account</Department>
                        <Department>Sales</Department>
                        <Department>Pre-Sales</Department>
                        <Department>Marketing</Department>
                        </Departments>";

        XDocument xdoc = new XDocument();
        xdoc = XDocument.Parse(myXML);

        //Add new Element
        xdoc.Element("Departments").Add(new XElement("Department",
"Finance"));

        //Add new Element at First
        xdoc.Element("Departments").AddFirst(new XElement("Department",
"Support"));

        var result = xdoc.Element("Departments").Descendants();

        foreach (XElement item in result)
        {
            Console.WriteLine("Department Name - " + item.Value);
        }

        Console.WriteLine("\nPress any key to continue.");
        Console.ReadKey();
    }
}

```

When the above code of C# or VB is compiled and executed, it produces the following result:

```

Department Name - Support
Department Name - Account
Department Name - Sales
Department Name - Pre-Sales
Department Name - Marketing
Department Name - Finance

```

Press any key to continue.

Deleting Particular Node

C#

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Xml.Linq;

namespace LINQtoXML
{

```

```

class ExampleOfXML
{
    static void Main(string[] args)
    {
        string myXML = @"<Departments>
                        <Department>Support</Department>
                        <Department>Account</Department>
                        <Department>Sales</Department>
                        <Department>Pre-Sales</Department>
                        <Department>Marketing</Department>
                        <Department>Finance</Department>
                        </Departments>";

        XDocument xdoc = new XDocument();
        xdoc = XDocument.Parse(myXML);

        //Remove Sales Department
        xdoc.Descendants().Where(s =>s.Value == "Sales").Remove();

        var result = xdoc.Element("Departments").Descendants();

        foreach (XElement item in result)
        {
            Console.WriteLine("Department Name - " + item.Value);
        }

        Console.WriteLine("\nPress any key to continue.");
        Console.ReadKey();
    }
}

```

When the above code of C# is compiled and executed, it produces the following result:

```

Department Name - Support
Department Name - Account
Department Name - Pre-Sales
Department Name - Marketing
Department Name - Finance

Press any key to continue.

```

LINQ - Entities

A part of the ADO.NET Entity Framework, LINQ to Entities is more flexible than LINQ to SQL, but is not much popular because of its complexity and lack of key features. However, it does not have the limitations of LINQ to SQL that allows data query only in SQL server database as LINQ to Entities facilitates data query in a large number of data providers like Oracle, MySQL, etc.

Moreover, it has got a major support from ASP.Net in the sense that users can make use of a data source control for executing a query via LINQ to Entities and facilitates binding of the results without any need of extra coding.

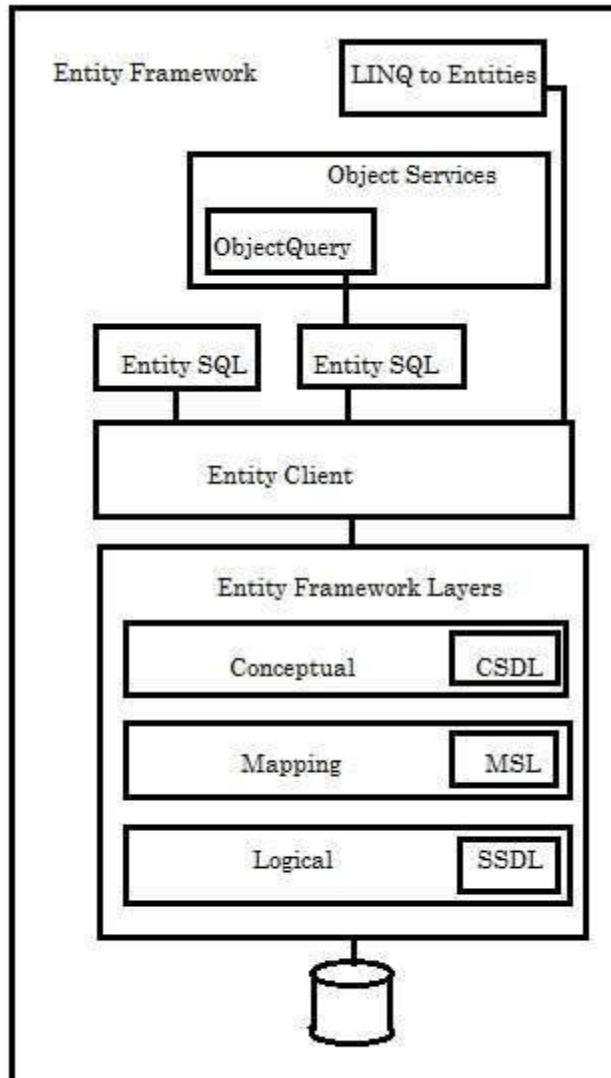
LINQ to Entities has for these advantages become the standard mechanism for the usage of LINQ on databases nowadays. It is also possible with LINQ to Entities to change queried data details and committing a batch update easily. What is the most intriguing fact about LINQ to Entities is that it has same syntax like that of SQL and even has the same group of standard query operators like Join, Select, OrderBy, etc.

LINQ to Entities Query Creation and Execution Process

- Construction of an **ObjectQuery** instance out of an **ObjectContext** EntityConnectionEntityConnection
- Composing a query either in C# or Visual Basic VBVB by using the newly constructed instance
- Conversion of standard query operators of LINQ as well as LINQ expressions into command trees
- Executing the query passing any exceptions encountered to the client directly
- Returning to the client all the query results

ObjectContext is here the primary class that enables interaction with **Entity Data Model** or in other words acts as a bridge that connects LINQ to the database. Command trees are here query representation with compatibility with the Entity framework. The Entity Framework on the other hand is actually **Object Relational Mapper** abbreviated generally as ORM by the developers that does the generation of business objects as well as entities as per the database tables and facilitates various basic operations like create, update, delete and read.

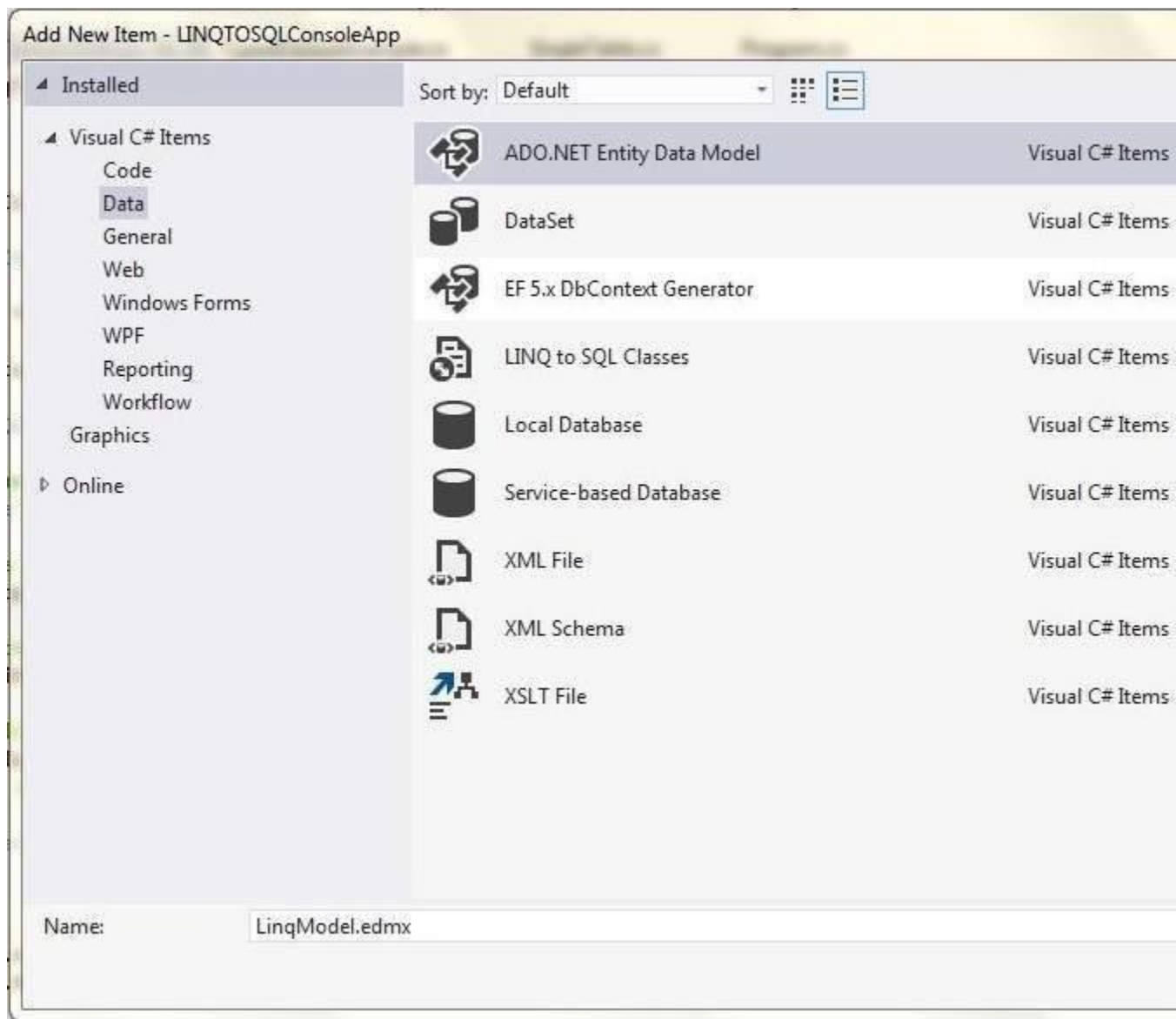
Below is a diagram of the entity framework to have a better understanding of the concept.



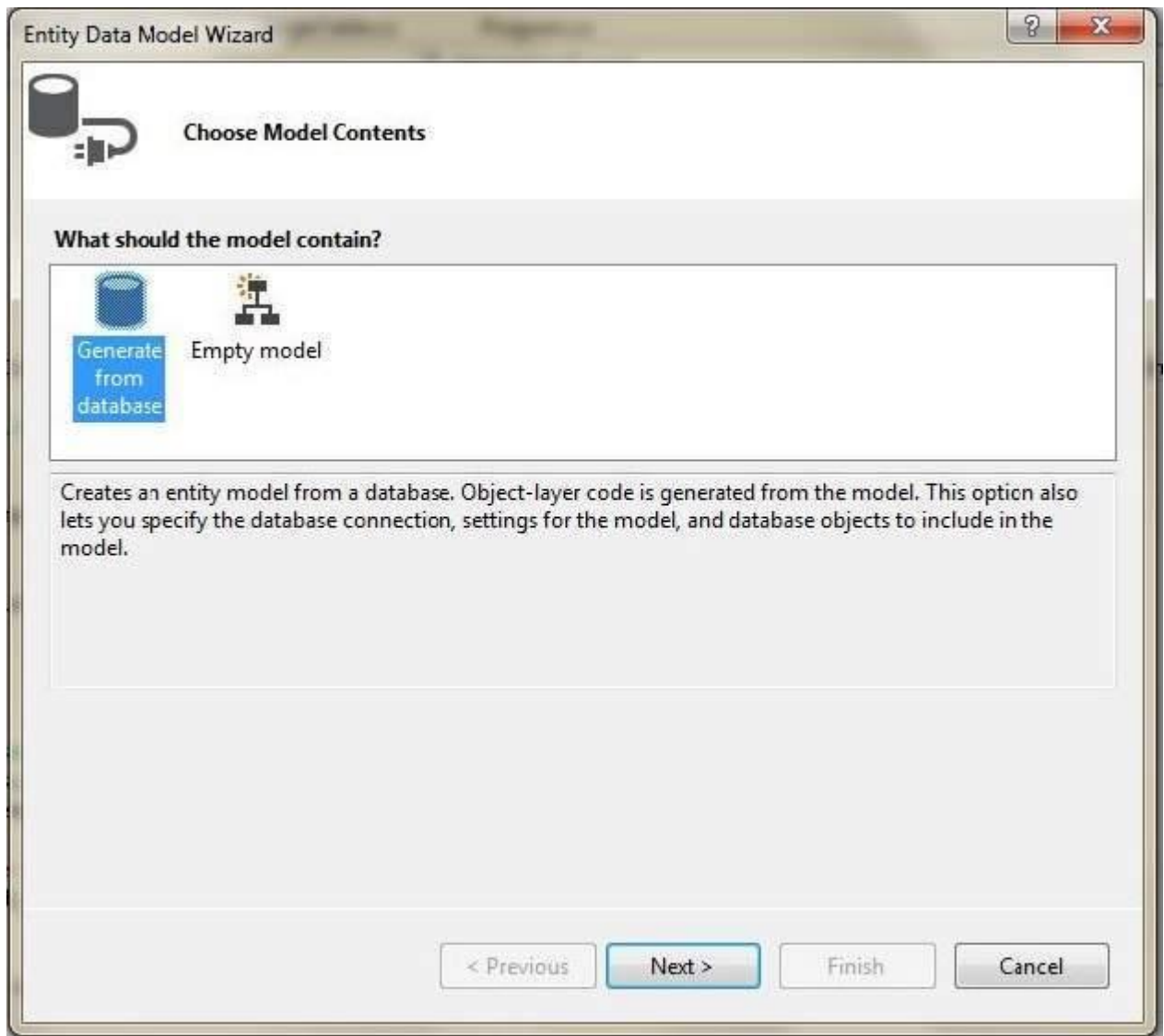
Example of ADD, UPDATE and DELETE using LINQ with Entity Model

First add Entity Model by following below steps.

- **Step 1:** Right click on project and click add new item will open window as per below. Select ADO.NET Entity Data Model and specify name and click on Add.




- **Step 2:** Select Generate from database.



- **Step 3: Choose Database Connection.**

Entity Data Model Wizard

 Choose Your Data Connection

Which data connection should your application use to connect to the database?

LinqToSQLDBConnectionString (Settings) New Connection...

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☐ No, exclude sensitive data from the connection string. I will set it in my application code.

☐ Yes, include the sensitive data in the connection string.

Entity connection string:

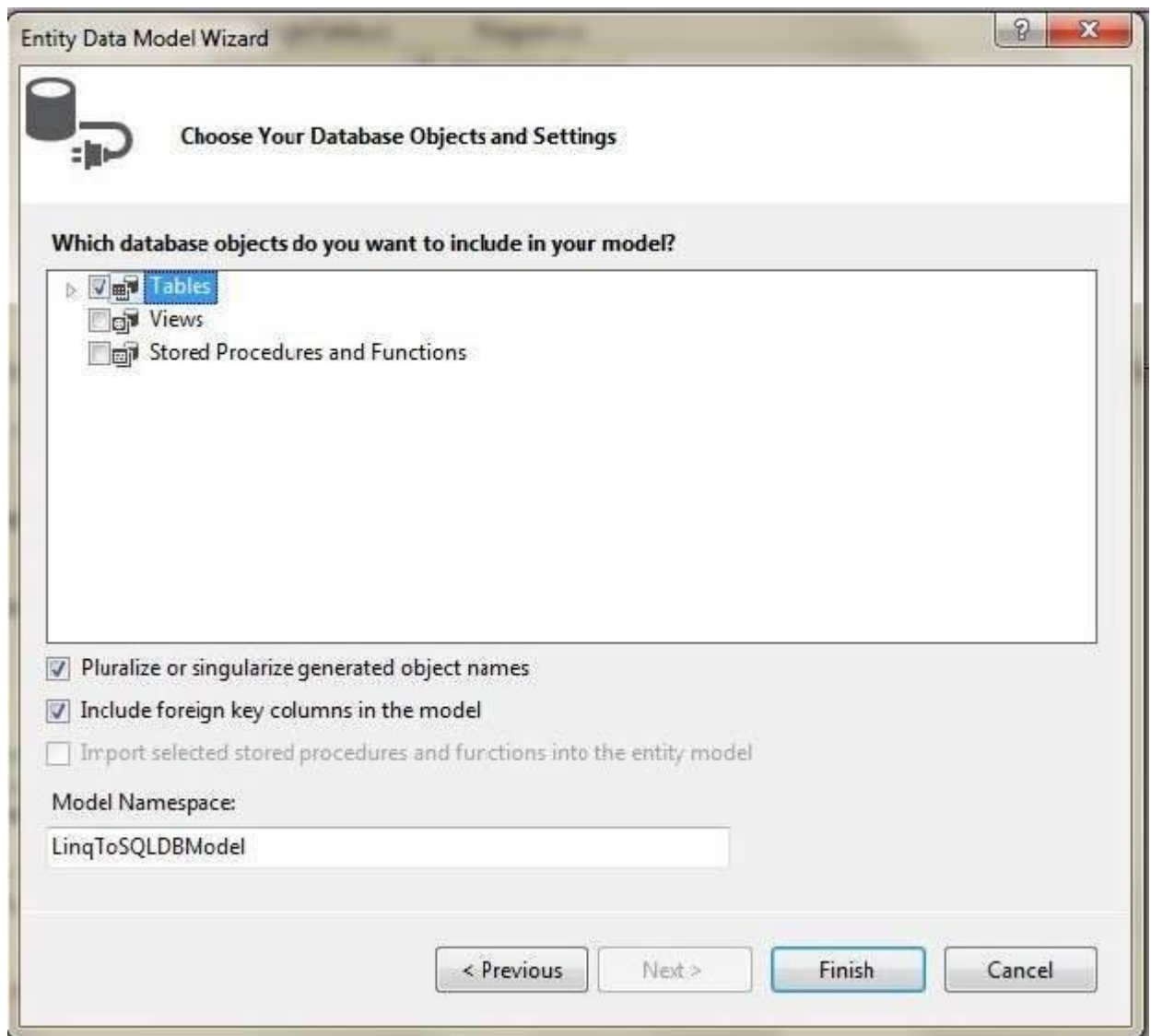
```
metadata=res://*/LinqModel.csdl|res://*/LinqModel.ssdl|
res://*/LinqModel.msl;provider=System.Data.SqlClient;provider connection string="data
source=HITESH-PC\MSSQLSERVER2008;initial catalog=LinqToSQLDB;integrated
security= True;MultipleActiveResultSets= True;App=EntityFramework"
```

☒ Save entity connection settings in App.Config as:

LinqToSQLDBEntities

< Previous **Next >** Finish Cancel

- **Step 4:** Select all the tables.



Now write the following code.

```
using DataAccess;
using System;
using System.Linq;

namespace LinqToSQLConsoleApp
{
    public class LinqToEntityModel
    {
        static void Main(string[] args)
        {
            using (LinqToSQLDBEntities context = new LinqToSQLDBEntities())
            {
                //Get the List of Departments from Database
                var departmentList = from d in context.Departments
                                    select d;
            }
        }
    }
}
```

```

        foreach (var dept in departmentList)
        {
            Console.WriteLine("Department Id = {0} , Department Name =
{1}",
                                dept.DepartmentId, dept.Name);
        }

        //Add new Department
        DataAccess.Department department = new DataAccess.Department();
        department.Name = "Support";

        context.Departments.Add(department);
        context.SaveChanges();

        Console.WriteLine("Department Name = Support is inserted in
Database");

        //Update existing Department
        DataAccess.Department updateDepartment =
context.Departments.FirstOrDefault(d =>d.DepartmentId == 1);
        updateDepartment.Name = "Account updated";
        context.SaveChanges();

        Console.WriteLine("Department Name = Account is updated in
Database");

        //Delete existing Department
        DataAccess.Department deleteDepartment =
context.Departments.FirstOrDefault(d =>d.DepartmentId == 3);
        context.Departments.Remove(deleteDepartment);
        context.SaveChanges();

        Console.WriteLine("Department Name = Pre-Sales is deleted in
Database");

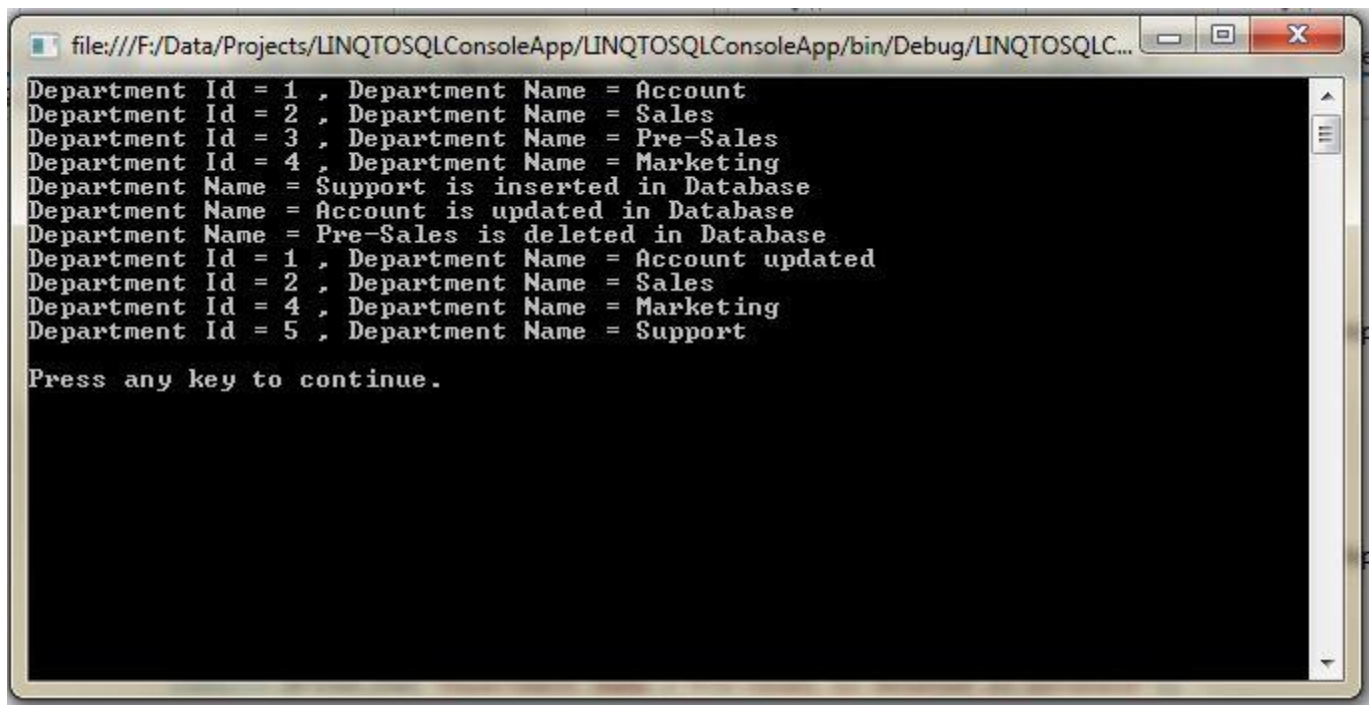
        //Get the Updated List of Departments from Database
        departmentList = from d in context.Departments
        select d;

        foreach (var dept in departmentList)
        {
            Console.WriteLine("Department Id = {0} , Department Name =
{1}",
                                dept.DepartmentId, dept.Name);
        }
    }

    Console.WriteLine("\nPress any key to continue.");
    Console.ReadKey();
}
}

```

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



```
file:///F:/Data/Projects/LINQTOSQLConsoleApp/LINQTOSQLConsoleApp/bin/Debug/LINQTOSQLC...
Department Id = 1 , Department Name = Account
Department Id = 2 , Department Name = Sales
Department Id = 3 , Department Name = Pre-Sales
Department Id = 4 , Department Name = Marketing
Department Name = Support is inserted in Database
Department Name = Account is updated in Database
Department Name = Pre-Sales is deleted in Database
Department Id = 1 , Department Name = Account updated
Department Id = 2 , Department Name = Sales
Department Id = 4 , Department Name = Marketing
Department Id = 5 , Department Name = Support

Press any key to continue.
```

LINQ - Lambda Expressions

The term 'Lambda expression' has derived its name from 'lambda' calculus which in turn is a mathematical notation applied for defining functions. Lambda expressions as a LINQ equation's executable part translate logic in a way at run time so it can pass on to the data source conveniently. However, lambda expressions are not just limited to find application in LINQ only.

These expressions are expressed by the following syntax:

inputparameters => expression or statement block

Below is an example of lambda expression

$y \Rightarrow y * y$

The above expression specifies a parameter named y and that value of y is squared. However, it is not possible to execute a lambda expression in this form. Example of a lambda expression in C# is shown below.

C#

```
using System;
```

```

using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace lambdaexample
{
    class Program
    {
        delegate int del(int i);
        static void Main(string[] args)
        {
            del myDelegate = y => y * y;
            int j = myDelegate(5);
            Console.WriteLine(j);
            Console.ReadLine();
        }
    }
}

```

When the above code of C# is compiled and executed, it produces the following result:

25

Expression Lambda

As the expression in the syntax of lambda expression shown above is on the right hand side, these are also known as expression lambda.

Async Lambdas

The lambda expression created by incorporating asynchronous processing by the use of async keyword is known as async lambdas. Below is an example of async lambda.

```
Func<Task<string>> getWordAsync = async() => "hello";
```

Lambda in Standard Query Operators

A lambda expression within a query operator is evaluated by the same upon demand and continually works on each of the elements in the input sequence and not the whole sequence. Developers are allowed by Lambda expression to feed their own logic into the standard query operators. In the below example, the developer has used the ‘Where’ operator to reclaim the odd values from given list by making use of a lambda expression.

C#


```
//Get the average of the odd Fibonacci numbers in the series...
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace lambdaexample
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] fibNum = { 1, 1, 2, 3, 5, 8, 13, 21, 34 };
            double averageValue = fibNum.Where(num => num % 2 == 1).Average();
            Console.WriteLine(averageValue);
            Console.ReadLine();
        }
    }
}
```

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

```
7.333333333333333
```

Type Inference in Lambda

In C#, type inference is used conveniently in a variety of situations and that too without specifying the types explicitly. However in case of a lambda expression, type inference will work only when each type has been specified as the compiler must be satisfied. Let's consider the following example.

```
delegate int Transformer (int i);
```

Here the compiler employ the type inference to draw upon the fact that x is an integer and this is done by examining the parameter type of the Transformer.

Variable Scope in Lambda Expression

There are some rules while using variable scope in a lambda expression like variables that are initiated within a lambda expression are not meant to be visible in an outer method. There is also a rule that a captured variable is not to be garbage collected unless the delegate referencing the same becomes eligible for the act of garbage collection. Moreover, there is a rule that prohibits a return statement within a lambda expression to cause return of an enclosing method.

Here is an example to demonstrate variable scope in lambda expression.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace lambdaexample
{
    class Program
    {
        delegate bool D();
        delegate bool D2(int i);

        class Test
        {
            D del;
            D2 del2;
            public void TestMethod(int input)
            {
                int j = 0;
                // Initialize the delegates with lambda expressions.
                // Note access to 2 outer variables.
                // del will be invoked within this method.
                del = () => { j = 10; return j > input; };

                // del2 will be invoked after TestMethod goes out of scope.
                del2 = (x) => { return x == j; };

                // Demonstrate value of j:
                // The delegate has not been invoked yet.
                Console.WriteLine("j = {0}", j);           // Invoke the delegate.
                bool boolResult = del();

                Console.WriteLine("j = {0}. b = {1}", j, boolResult);
            }

            static void Main()
            {
                Test test = new Test();
                test.TestMethod(5);

                // Prove that del2 still has a copy of
                // local variable j from TestMethod.
                bool result = test.del2(10);

                Console.WriteLine(result);

                Console.ReadKey();
            }
        }
    }
}
```

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

```
j = 0
j = 10. b = True
True
```

Expression Tree

Lambda expressions are used in **Expression Tree** construction extensively. An expression tree give away code in a data structure resembling a tree in which every node is itself an expression like a method call or can be a binary operation like $x < y$. Below is an example of usage of lambda expression for constructing an expression tree.

Statement Lambda

There is also **statement lambdas** consisting of two or three statements, but are not used in construction of expression trees. A return statement must be written in a statement lambda.

Syntax of statement lambda

```
(params) => {statements}
```

Example of a statement lambda

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Linq.Expressions;

namespace lambdaexample
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] source = new[] { 3, 8, 4, 6, 1, 7, 9, 2, 4, 8 };

            foreach (int i in source.Where(
                x =>
                {
                    if (x <= 3)
                        return true;
                    else if (x >= 7)
                        return true;
                    return false;
                }
            ))
                Console.WriteLine(i);
            Console.ReadLine();
        }
    }
}
```

```
}  
}  
}
```

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

```
3  
8  
1  
7  
9  
2  
8
```

Lambdas are employed as arguments in LINQ queries based on methods and never allowed to have a place on the left side of operators like `is` or `as` just like anonymous methods. Although, Lambda expressions are much alike anonymous methods, these are not at all restricted to be used as delegates only.

Important points to remember while using lambda expressions

- A lambda expression can return a value and may have parameters.
- Parameters can be defined in a myriad of ways with a lambda expression.
- If there is single statement in a lambda expression, there is no need of curly brackets whereas if there are multiple statements, curly brackets as well as return value are essential to write.
- With lambda expressions, it is possible to access variables present outside of the lambda expression block by a feature known as closure. Use of closure should be done cautiously to avoid any problem.
- It is impossible to execute any unsafe code inside any lambda expression.
- Lambda expressions are not meant to be used on the operator's left side.

https://www.tutorialspoint.com/linq/linq_overview.htm

Copyright © tutorialspoint.com