



LinQ

language integrated query

Microsoft



.NET

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

About the Tutorial

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.

This tutorial offers a complete insight into LINQ with ample examples and coding. The entire tutorial is divided into various topics with subtopics that a beginner can be able to move gradually to more complex topics of LINQ.

Audience

The aim of this tutorial is to offer an easy understanding of LINQ to the beginners who are keen to learn the same.

Prerequisites

It is essential before proceeding to have some basic knowledge of C# and Visual Basic in .NET.

Copyright & Disclaimer

© Copyright 2015 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute, or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness, or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

Table of Contents

About the Tutorial.....	i
Audience	i
Prerequisites	i
Copyright & Disclaimer.....	i
Table of Contents	ii
1. OVERVIEW.....	1
Example of a LINQ query	1
Syntax of LINQ.....	2
Types of LINQ.....	3
LINQ Architecture in .NET.....	3
Query Expressions.....	4
Extension Methods	5
Difference between LINQ and Stored Procedure.....	5
Need For LINQ.....	5
Advantages of LINQ.....	6
2. ENVIRONMENT.....	7
Getting Visual Studio 2010 Installed on Windows 7	7
Writing C# Program using LINQ in Visual Studio 2010	11
Writing VB Program using LINQ in Visual Studio 2010	13
3. QUERY OPERATORS.....	15
Filtering Operators	15
Filtering Operators in LINQ.....	16
Join Operators.....	17
Join Operators in LINQ	18
Projection Operations	22

Projection Operations in LINQ.....	22
Sorting Operators.....	26
Grouping Operators	29
Conversions.....	31
Concatenation.....	35
Aggregation.....	38
Quantifier Operations	40
Partition Operators	43
Generation Operations.....	48
Set Operations	53
Equality.....	59
Element Operators	62
4. SQL.....	68
Introduction of LINQ To SQL.....	68
How to Use LINQ to SQL?	69
Querying with LINQ to SQL.....	71
Insert, Update, and Delete using LINQ to SQL	72
5. OBJECTS	80
Introduction of LINQ to Objects	80
Querying in Memory Collections Using LINQ to Objects	81
6. DATASET.....	84
Introduction of LINQ to Dataset	84
Querying Dataset using LinQ to Dataset	88
7. XML.....	92
Introduction of LINQ to XML	92
Read an XML File using LINQ	93

Add New Node	95
Deleting Particular Node	97
8. ENTITIES	101
LINQ to Entities Query Creation and Execution Process	101
Example of ADD, UPDATE, and DELETE using LINQ with Entity Model.....	102
9. LAMBDA EXPRESSIONS	108
Expression Lambda	109
Async Lambdas.....	109
Lambda in Standard Query Operators	109
Type Inference in Lambda	111
Variable Scope in Lambda Expression	111
Expression Tree	113
Statement Lambda	113
10. ASP.NET.....	116
LINQDataSource Control	117
INSERT, UPDATE, and DELETE data in ASP.NET Page using LINQ.....	121

1. 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 (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

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();
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim words As String() = {"hello", "wonderful", "LINQ", "beautiful", "world"}

        ' Get only short words
        Dim shortWords = From word In words _ Where word.Length <= 5 _ Select word

        ' Print each word out.

        For Each word In shortWords
            Console.WriteLine(word)
        Next

        Console.ReadLine()
    End Sub
End Module
```

When the above code of C# or VB 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

```
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

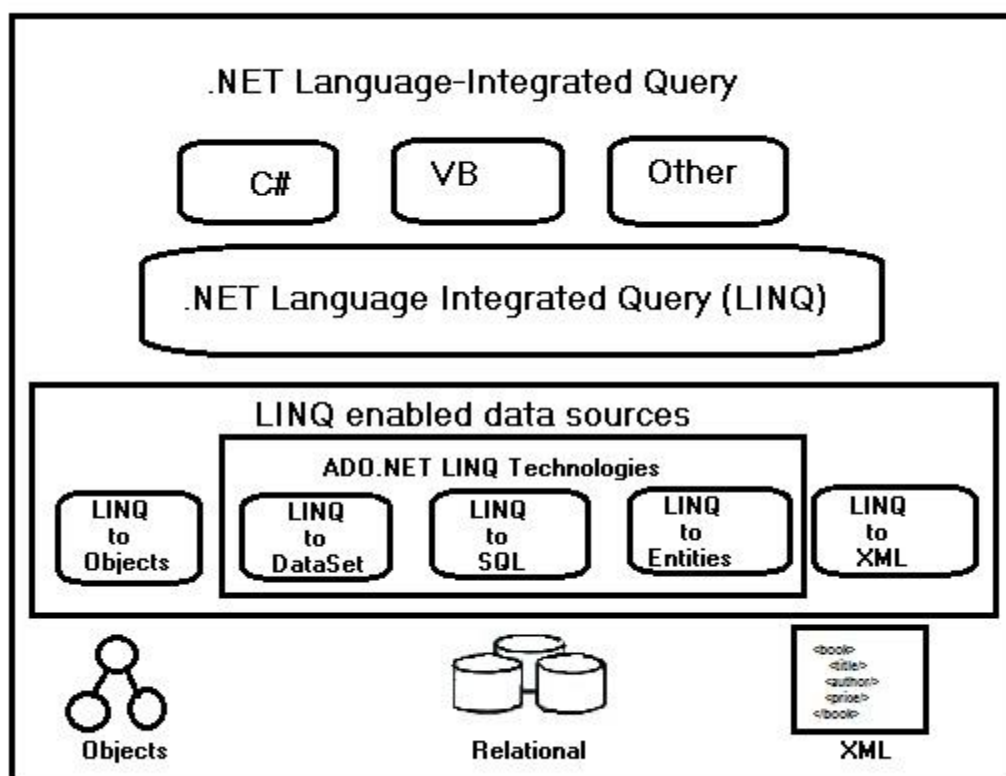
The types of LINQ are mentioned below in brief.

- LINQ to Objects
- LINQ to XML(XLINQ)
- LINQ to DataSet
- LINQ to SQL (DLINQ)
- 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.



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(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, 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.

2. ENVIRONMENT

Before starting with LINQ programs, it is best to first understand the nuances of setting up a LINQ environment. LINQ needs a .NET framework, a revolutionary platform to have a diverse kind of applications. A LINQ query can be written either in C# or Visual Basic conveniently.

Microsoft offers tools for both of these languages i.e. C# and Visual Basic by means of Visual Studio. Our examples are all compiled and written in Visual Studio 2010. However, Visual Basic 2013 edition is also available for use. It is the latest version and has many similarities with Visual Studio 2012.

Getting Visual Studio 2010 Installed on Windows 7

Visual Studio can be installed either from an installation media like a DVD. Administrator credentials are required to install Visual Basic 2010 on your system successfully. It is vital to disconnect all removable USB from the system prior to installation otherwise the installation may get failed. Some of the hardware requirements essential to have for installation are the following ones.

Hardware Requirements

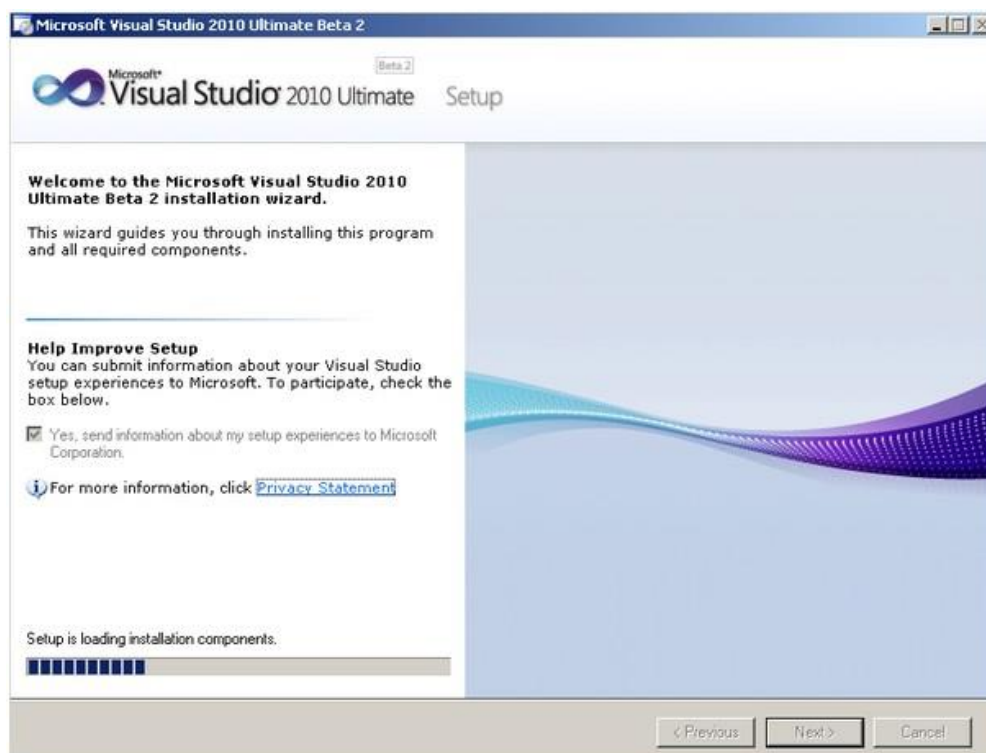
- 1.6 GHz or more
- 1 GB RAM
- 3 GB(Available hard-disk space)
- 5400 RPM hard-disk drive
- DirectX 9 compatible video card
- DVD-ROM drive

Installation Steps

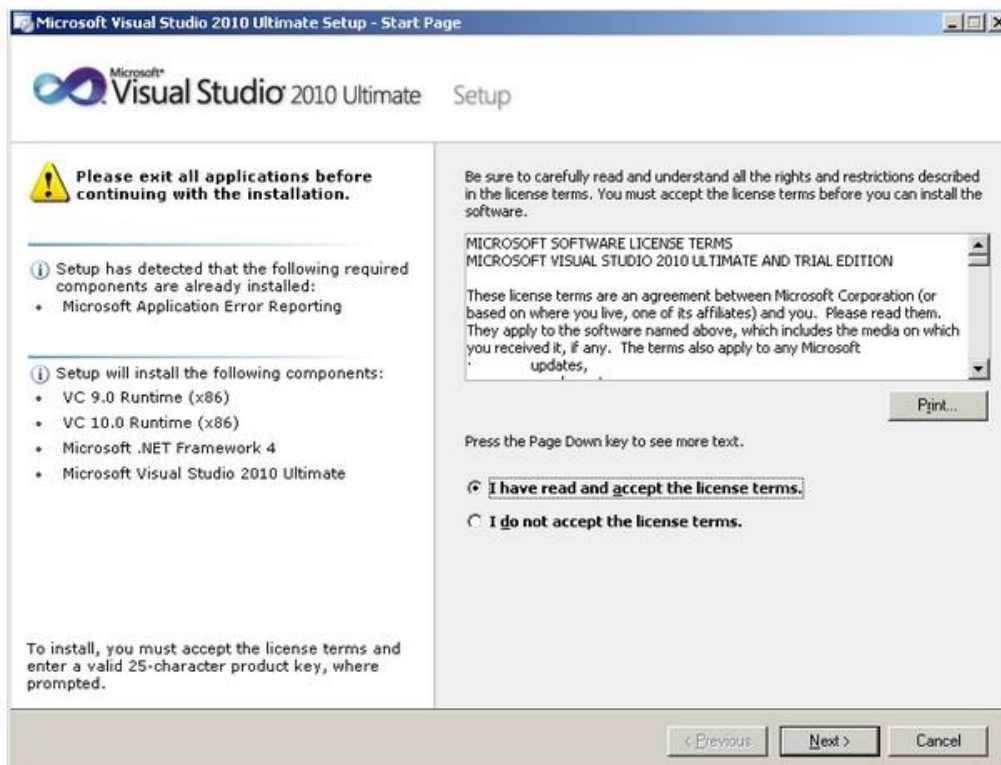
- **Step 1:** First after inserting the DVD with Visual Studio 2010 Package, click on **Install or run program from your media** appearing in a pop-up box on the screen.
- **Step 2:** Now set up for Visual Studio will appear on the screen. Choose **Install Microsoft Visual Studio 2010**.



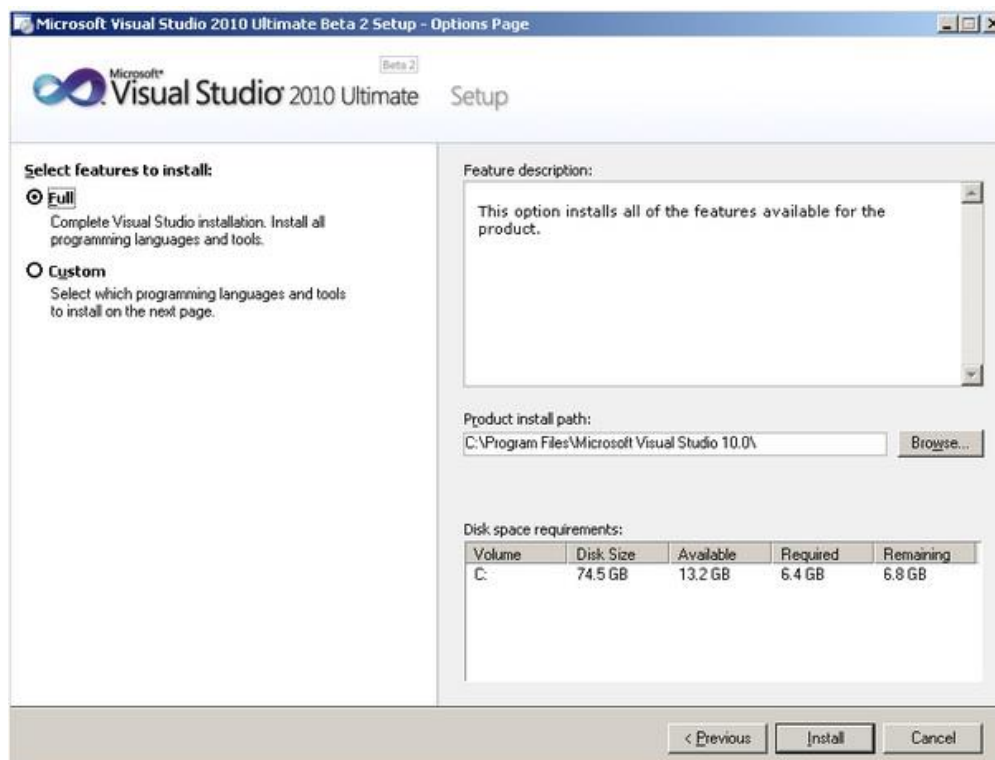
- **Step 3:** As soon as you will click, it the process will get initiated and a set up window will appear on your screen. After completion of loading of the installation components which will take some time, click on **Next** button to move to the next step.



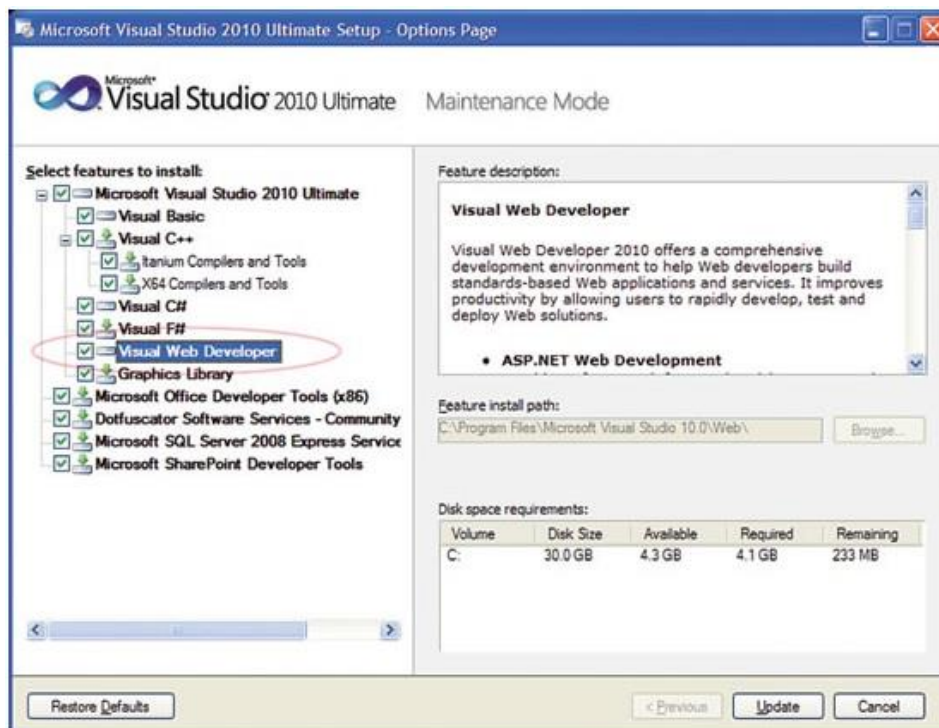
- **Step 4:** This is the last step of installation and a start page will appear in which simply choose "I have read and accept the license terms" and click on **Next** button.



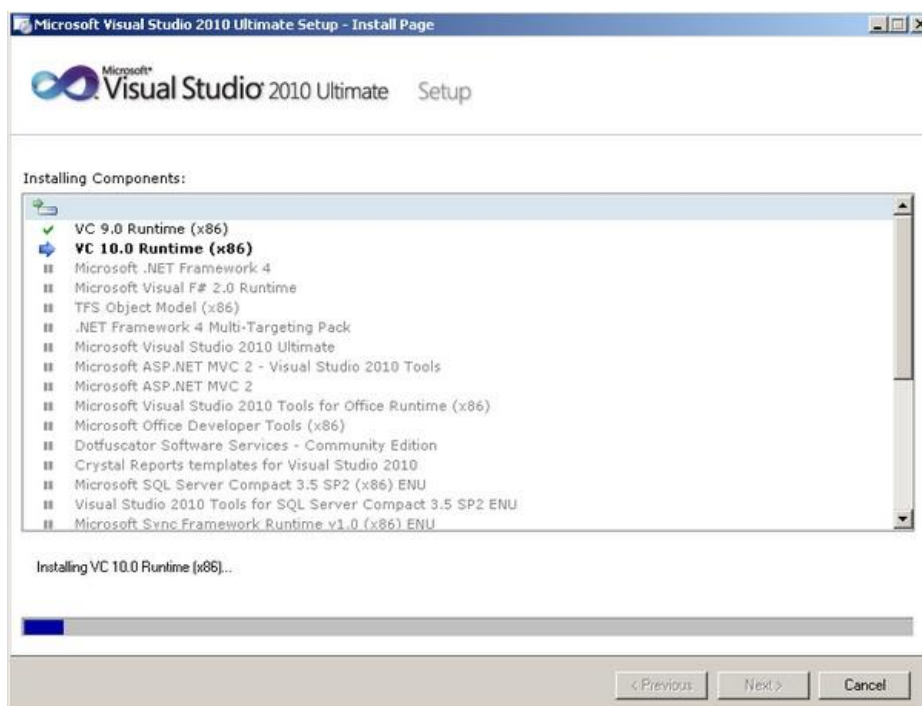
- **Step 5:** Now select features to install from the options page appearing on your screen. You can either choose Full or Custom option. If you have less disk space than required shown in the disk space requirements, then go for Custom.



- **Step 6:** When you choose Custom option, the following window will appear. Select the features that you want to install and click **Update** or else go to step 7. However, it is recommended not to go with the custom option as in future, you may need the features you have chosen to not have.



- **Step 7:** Soon a pop up window will be shown and the installation will start which may take a long time. Remember, this is for installing all the components.

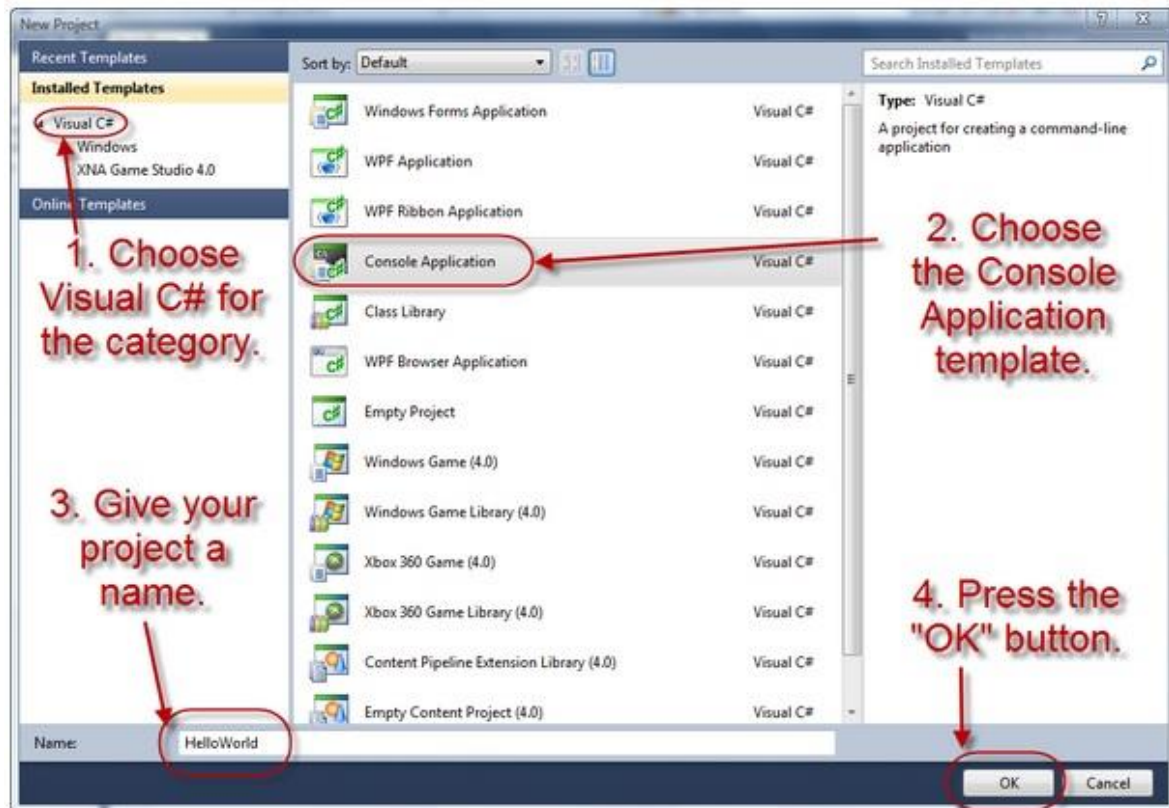


- **Step 8:** Finally, you will be able to view a message in a window that the installation has been completed successfully. Click **Finish**.

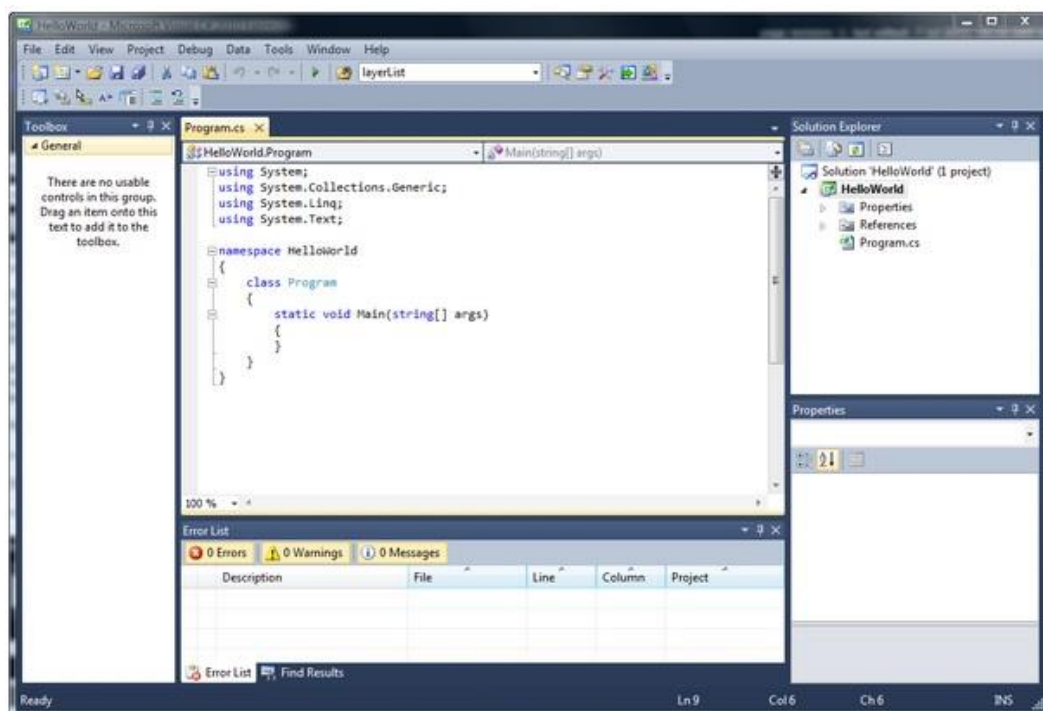


Writing C# Program using LINQ in Visual Studio 2010

1. Start Visual Studio 2010 Ultimate edition and choose File followed by New Project from the menu.
2. A new project dialog box will appear on your screen.
3. Now choose Visual C# as a category under installed templates and next choose Console Application template as shown in figure below.



4. Give a name to your project in the bottom name box and press OK.
5. The new project will appear in the Solution Explorer in the right-hand side of a new dialog box on your screen.



6. Now choose Program.cs from the Solution Explorer and you can view the code in the editor window which starts with 'using System'.
7. Here you can start to code your following C# program.

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

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World")
            Console.ReadKey();
        }
    }
}
```

8. Press F5 key and run your project. It is highly recommended to save the project by choosing **File** → **Save All** before running the project.

Writing VB Program using LINQ in Visual Studio 2010

1. Start Visual Studio 2010 Ultimate edition and choose File followed by New Project from the menu.
2. A new project dialog box will appear on your screen.
3. Now chose Visual Basic as a category under installed templates and next choose Console Application template.
4. Give a name to your project in the bottom name box and press OK.
5. You will get a screen with Module1.vb. Start writing your VB code here using LINQ.

```
Module Module1

    Sub Main()
        Console.WriteLine("Hello World")
        Console.ReadLine()
    End Sub

End Module
```

6. Press F5 key and run your project. It is highly recommended to save the project by choosing File → Save All before running the project.

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

```
Hello World
```

3. 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
--------	--	----------------	----------------

Filtering Operators in LINQ

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

Example of Where – Query Expression

C#

```
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();
    }
}

```

VB

```

Module Module1

    Sub Main()
        Dim words As String() = {"humpty", "dumpty", "set", "on", "a", "wall"}

        Dim query = From word In words Where word.Length = 3 Select word

        For Each n In query
            Console.WriteLine(n)
        Next
        Console.ReadLine()
    End Sub

End Module

```

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

```
set
```

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.

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 ...
-----------	--	--	------------------------------

Join Operators in LINQ

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

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 ...

Example of Join – Query Expression

C#

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

namespace Operators
{
    class JoinTables
    {
        class DepartmentClass
        {
            public int DepartmentId { get; set; }
            public string Name { get; set; }
        }

        class EmployeeClass
        {
```

```

        public int EmployeeId { get; set; }

        public string EmployeeName { get; set; }

        public int DepartmentId { get; set; }
    }

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

        List <EmployeeClass> employees = new List <EmployeeClass>();
        employees.Add(new EmployeeClass { DepartmentId = 1, EmployeeId = 1,
EmployeeName = "William" });
        employees.Add(new EmployeeClass { DepartmentId = 2, EmployeeId = 2,
EmployeeName = "Miley" });
        employees.Add(new EmployeeClass { DepartmentId = 1, EmployeeId = 3,
EmployeeName = "Benjamin" });

        var list = (from e in employees join d in departments on
e.DepartmentId equals d.DepartmentId select new
        {
            EmployeeName = e.EmployeeName,
            DepartmentName = d.Name
        });

        foreach (var e in list)
        {
            Console.WriteLine("Employee Name = {0} , Department Name = {1}",
e.EmployeeName, e.DepartmentName);
        }

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

```



```

    }
}
}

```

VB

```

Module Module1

    Sub Main()

        Dim account As New Department With {.Name = "Account", .DepartmentId = 1}
        Dim sales As New Department With {.Name = "Sales", .DepartmentId = 2}
        Dim marketing As New Department With {.Name = "Marketing", .DepartmentId
= 3}

        Dim departments As New System.Collections.Generic.List(Of Department)(New
Department() {account, sales, marketing})

        Dim william As New Employee With {.EmployeeName = "William", .EmployeeId
= 1, .DepartmentId = 1}
        Dim miley As New Employee With {.EmployeeName = "Miley", .EmployeeId = 2,
.DepartmentId = 2}
        Dim benjamin As New Employee With {.EmployeeName = "Benjamin",
.EmployeeId = 3, .DepartmentId = 1}

        Dim employees As New System.Collections.Generic.List(Of Employee)(New
Employee() {william, miley, benjamin})

        Dim list = (From e In employees
                    Join d In departments On e.DepartmentId Equals d.DepartmentId
                    Select New Person With {.EmployeeName = e.EmployeeName,
.DepartmentName = d.Name})

        For Each e In list
            Console.WriteLine("Employee Name = {0} , Department Name = {1}",
e.EmployeeName, e.DepartmentName)
        Next

        Console.WriteLine(vbLf & "Press any key to continue.")
    End Sub
End Module

```

```
        Console.ReadKey()

    End Sub

    Class Employee
        Public Property EmployeeId As Integer
        Public Property EmployeeName As String
        Public Property DepartmentId As Integer
    End Class

    Class Department
        Public Property Name As String
        Public Property DepartmentId As Integer
    End Class

    Class Person
        Public Property EmployeeName As String
        Public Property DepartmentName As String
    End Class

End Module
```

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

```
Employee Name = William, Department Name = Account
Employee Name = Miley, Department Name = Sales
Employee Name = Benjamin, Department Name = Account

Press any key to continue.
```

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

Projection Operations in LINQ

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

Example of Select – Query Expression

C#

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

```

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> words = new List<string>() { "an", "apple", "a", "day" };

            var query = from word in words select word.Substring(0, 1);

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

```

VB

```

Module Module1

    Sub Main()

        Dim words = New List(Of String) From {"an", "apple", "a", "day"}

        Dim query = From word In words Select word.Substring(0, 1)

        Dim sb As New System.Text.StringBuilder()

        For Each letter As String In query
            sb.AppendLine(letter)
            Console.WriteLine(letter)
        Next
        Console.ReadLine()

    End Sub


```

End Module

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

a
a
a
d

Example of SelectMany – Query Expression

C#

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

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            List<string> phrases = new List<string>() { "an apple a day", "the quick brown fox" };

            var query = from phrase in phrases
                        from word in phrase.Split(' ')
                        select word;

            foreach (string s in query)
                Console.WriteLine(s);
            Console.ReadLine();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()

        Dim phrases = New List(Of String) From {"an apple a day", "the quick
brown fox"}

        Dim query = From phrase In phrases
                    From word In phrase.Split(" ")
                    Select word

        Dim sb As New System.Text.StringBuilder()

        For Each str As String In query
            sb.AppendLine(str)
            Console.WriteLine(str)
        Next
        Console.ReadLine()

    End Sub

End Module
```

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

```
an
apple
a
day
the
quick
brown
fox
```

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

Example of OrderBy, OrderByDescending – Query Expression

C#

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

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] num = { -20, 12, 6, 10, 0, -3, 1 };
        }
    }
}
```

```

        //create a query that obtain the values in sorted order
        var posNums = from n in num
                       orderby n
                       select n;
        Console.WriteLine("Values in ascending order: ");

        // Execute the query and display the results.

        foreach (int i in posNums)
            Console.Write(i + " \n");

        var posNumsDesc = from n in num
                           orderby n descending
                           select n;
        Console.WriteLine("\nValues in descending order: ");

        // Execute the query and display the results.

        foreach (int i in posNumsDesc)
            Console.Write(i + " \n");

        Console.ReadLine();
    }
}
}

```

VB

```

Module Module1

    Sub Main()

        Dim num As Integer() = {-20, 12, 6, 10, 0, -3, 1}

        Dim posNums = From n In num
                       Order By n

```



```
        Select n
        Console.Write("Values in ascending order: ")

        For Each n In posNums
            Console.WriteLine(n)
        Next

        Dim posNumsDesc = From n In num
                           Order By n Descending
                           Select n
        Console.Write("Values in descending order: ")

        For Each n In posNumsDesc
            Console.WriteLine(n)
        Next
        Console.ReadLine()

    End Sub

End Module
```

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

```
Values in ascending order: -20
-3
0
1
6
10
12
Values in descending order: 12
10
6
1
0
-3
```

-20

In Thenby and ThenbyDescending operators, same syntax can be applied and sorting order will depend on more than one columns. Priority will be the column which is maintained first.

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

Example of GroupBy – Query Expression

C#

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

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> numbers = new List<int>() { 35, 44, 200, 84, 3987, 4, 199,
            329, 446, 208 };

            IEnumerable<IGrouping<int, int>> query = from number in numbers
                                                    group number by number % 2;
```

```

        foreach (var group in query)
        {
            Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd numbers:");

            foreach (int i in group)
                Console.WriteLine(i);
        }
        Console.ReadLine();
    }
}

```

VB

```

Module Module1

    Sub Main()

        Dim numbers As New System.Collections.Generic.List(Of Integer)(
            New Integer() {35, 44, 200, 84, 3987, 4, 199, 329, 446, 208})

        Dim query = From number In numbers
                     Group By Remainder = (number Mod 2) Into Group

        For Each group In query
            Console.WriteLine(If(group.Remainder = 0, vbCrLf &"Even numbers:",
                                vbCrLf &"Odd numbers:"))

            For Each num In group.Group
                Console.WriteLine(num)
            Next

        Next

        Console.ReadLine()
    End Sub
End Module

```

```
End Sub
```

```
End Module
```

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

```
Odd numbers:
```

```
35
```

```
3987
```

```
199
```

```
329
```

```
Even numbers:
```

```
44
```

```
200
```

```
84
```

```
4
```

```
446
```

```
208
```

Conversions

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

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

Example of Cast – Query Expression

C#

```
using System;
using System.Linq;

namespace Operators
{
    class Cast
    {
        static void Main(string[] args)
        {
            Plant[] plants = new Plant[] {new CarnivorousPlant { Name = "Venus Fly
Trap", TrapType = "Snap Trap" },
                                         new CarnivorousPlant { Name = "Pitcher
Plant", TrapType = "Pitfall Trap" },
                                         new CarnivorousPlant { Name = "Sundew",
TrapType = "Flypaper Trap" },
                                         new CarnivorousPlant { Name = "Waterwheel
Plant", TrapType = "Snap Trap" }};
```

```

        var query = from CarnivorousPlant cPlant in plants
                     where cPlant.TrapType == "Snap Trap"
                     select cPlant;

        foreach (var e in query)
        {
            Console.WriteLine("Name = {0} , Trap Type = {1}", e.Name,
e.TrapType);
        }

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

class Plant
{
    public string Name { get; set; }
}

class CarnivorousPlant : Plant
{
    public string TrapType { get; set; }
}
}

```

VB

```

Module Module1

    Sub Main()

        Dim plants() As Plant = {New CarnivorousPlant With {.Name = "Venus Fly
Trap", .TrapType = "Snap Trap"},
                                New CarnivorousPlant With {.Name = "Pitcher
Plant", .TrapType = "Pitfall Trap"},

```

```

        New CarnivorousPlant With {.Name =
"Sundew", .TrapType = "Flypaper Trap"},
        New CarnivorousPlant With {.Name = "Waterwheel
Plant", .TrapType = "Snap Trap"}}

    Dim list = From cPlant As CarnivorousPlant In plants
        Where cPlant.TrapType = "Snap Trap"
        Select cPlant

    For Each e In list
        Console.WriteLine("Name = {0} , Trap Type = {1}", e.Name, e.TrapType)
    Next

    Console.WriteLine(vbLf & "Press any key to continue.")
    Console.ReadKey()

End Sub

Class Plant
    Public Property Name As String
End Class

Class CarnivorousPlant
    Inherits Plant
    Public Property TrapType As String
End Class

End Module

```

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

```

Name = Venus Fly Trap, TrapType = Snap Trap
Name = Waterwheel Plant, TrapType = Snap Trap

Press any key to continue.

```

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

Example of Concat – Enumerable.Concat(Of TSource) Method

C#

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

namespace Operators
{
    class Concat
    {
        static void Main(string[] args)
        {
            Pet[] cats = GetCats();
            Pet[] dogs = GetDogs();

            IEnumerable<string> query = cats.Select(cat =>
cat.Name).Concat(dogs.Select(dog => dog.Name));

            foreach (var e in query)
            {
                Console.WriteLine("Name = {0} ", e);
            }

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



```

static Pet[] GetCats()
{
    Pet[] cats = { new Pet { Name="Barley", Age=8 },
                  new Pet { Name="Boots", Age=4 },
                  new Pet { Name="Whiskers", Age=1 } };

    return cats;
}

static Pet[] GetDogs()
{
    Pet[] dogs = { new Pet { Name="Bounder", Age=3 },
                  new Pet { Name="Snoopy", Age=14 },
                  new Pet { Name="Fido", Age=9 } };

    return dogs;
}

}

class Pet
{
    public string Name { get; set; }
    public int Age { get; set; }
}
}

```

VB

```

Module Module1

    Sub Main()

        Dim cats As List(Of Pet) = GetCats()
        Dim dogs As List(Of Pet) = GetDogs()

        Dim list = cats.Cast(Of Pet)().Concat(dogs.Cast(Of Pet)()).ToList()

        For Each e In list

```

```
        Console.WriteLine("Name = {0}", e.Name)
    Next

    Console.WriteLine(vbLf & "Press any key to continue.")
    Console.ReadKey()

End Sub

Function GetCats() As List(Of Pet)

    Dim cats As New List(Of Pet)

    cats.Add(New Pet With {.Name = "Barley", .Age = 8})
    cats.Add(New Pet With {.Name = "Boots", .Age = 4})
    cats.Add(New Pet With {.Name = "Whiskers", .Age = 1})

    Return cats

End Function

Function GetDogs() As List(Of Pet)

    Dim dogs As New List(Of Pet)

    dogs.Add(New Pet With {.Name = "Bounder", .Age = 3})
    dogs.Add(New Pet With {.Name = "Snoopy", .Age = 14})
    dogs.Add(New Pet With {.Name = "Fido", .Age = 9})

    Return dogs

End Function

Class Pet
    Public Property Name As String
    Public Property Age As Integer
End Class
```

End Module

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

Barley Boots Whiskers Bounder Snoopy Fido Press any key to continue.
--

Aggregation

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

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()
-----	--	----------------	------------------------------------

Example

VB

```
Module Module1

    Sub Main()

        Dim num As Integer() = {1, 2, 3, 4, 5, 6, 7, 8, 9}

        Dim intDivByTwo = Aggregate n In num
                                Where n > 6
                                Into Count()
        Console.WriteLine("Count of Numbers: " & intDivByTwo)

        Dim intResult = Aggregate n In num
                                Where n > 6
                                Into Average()
        Console.WriteLine("Average of Numbers: " & intResult)

        intResult = Aggregate n In num
                                Where n > 6
                                Into LongCount()
        Console.WriteLine("Long Count of Numbers: " & intResult)

        intResult = Aggregate n In num
                                Into Max()
        Console.WriteLine("Max of Numbers: " & intResult)

        intResult = Aggregate n In num
                                Into Min()
        Console.WriteLine("Min of Numbers: " & intResult)

        intResult = Aggregate n In num
```

```

        Into Sum()

        Console.WriteLine("Sum of Numbers: " & intResult)

        Console.ReadLine()

    End Sub

End Module

```

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

```

Count of Numbers: 3
Average of Numbers: 8
Long Count of Numbers: 3
Max of Numbers: 9
Min of Numbers: 1
Sum of Numbers: 45

```

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 contains that specific element , 'false' value is returned	Not Applicable	Not Applicable

Example of All – All(Of TSource) Extension Method

VB

```
Module Module1

    Sub Main()

        Dim barley As New Pet With {.Name = "Barley", .Age = 4}
        Dim boots As New Pet With {.Name = "Boots", .Age = 1}
        Dim whiskers As New Pet With {.Name = "Whiskers", .Age = 6}
        Dim bluemoon As New Pet With {.Name = "Blue Moon", .Age = 9}
        Dim daisy As New Pet With {.Name = "Daisy", .Age = 3}

        Dim charlotte As New Person With {.Name = "Charlotte", .Pets = New Pet()
{barley, boots}}
        Dim arlene As New Person With {.Name = "Arlene", .Pets = New Pet()
{whiskers}}
        Dim rui As New Person With {.Name = "Rui", .Pets = New Pet() {bluemoon,
daisy}}

        Dim people As New System.Collections.Generic.List(Of Person)(New Person()
{charlotte, arlene, rui})

        Dim query = From pers In people
                    Where (Aggregate pt In pers.Pets Into All(pt.Age > 2))
                    Select pers.Name

        For Each e In query
            Console.WriteLine("Name = {0}", e)
        Next

        Console.WriteLine(vbLf & "Press any key to continue.")
        Console.ReadKey()

    End Sub

    Class Person
        Public Property Name As String
```

```

        Public Property Pets As Pet()
    End Class

    Class Pet
        Public Property Name As String
        Public Property Age As Integer
    End Class

End Module

```

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

```

Arlene
Rui

Press any key to continue.

```

Example of Any – Extension Method

VB

```

Module Module1

    Sub Main()

        Dim barley As New Pet With {.Name = "Barley", .Age = 4}
        Dim boots As New Pet With {.Name = "Boots", .Age = 1}
        Dim whiskers As New Pet With {.Name = "Whiskers", .Age = 6}
        Dim bluemoon As New Pet With {.Name = "Blue Moon", .Age = 9}
        Dim daisy As New Pet With {.Name = "Daisy", .Age = 3}

        Dim charlotte As New Person With {.Name = "Charlotte", .Pets = New Pet()
        {barley, boots}}
        Dim arlene As New Person With {.Name = "Arlene", .Pets = New Pet()
        {whiskers}}
        Dim rui As New Person With {.Name = "Rui", .Pets = New Pet() {bluemoon,
        daisy}}

        Dim people As New System.Collections.Generic.List(Of Person)(New Person()
        {charlotte, arlene, rui})
    End Sub
End Module

```

```

Dim query = From pers In people
              Where (Aggregate pt In pers.Pets Into Any(pt.Age > 7))
              Select pers.Name

For Each e In query
    Console.WriteLine("Name = {0}", e)
Next

Console.WriteLine(vbLf & "Press any key to continue.")
Console.ReadKey()

End Sub

Class Person
    Public Property Name As String
    Public Property Pets As Pet()
End Class

Class Pet
    Public Property Name As String
    Public Property Age As Integer
End Class

End Module

```

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

Rui

Press any key to continue.

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

Example of Skip – Query Expression

VB

```

Module Module1

    Sub Main()

        Dim words = {"once", "upon", "a", "time", "there", "was", "a", "jungle"}

        Dim query = From word In words
                     Skip 4

        Dim sb As New System.Text.StringBuilder()

        For Each str As String In query
            sb.AppendLine(str)
            Console.WriteLine(str)
        Next

        Console.ReadLine()
    End Sub
End Module

```

```

End Sub

End Module

```

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

```

there
was
a
jungle

```

Example of Skip While – Query Expression

VB

```

Module Module1

    Sub Main()

        Dim words = {"once", "upon", "a", "time", "there", "was", "a", "jungle"}

        Dim query = From word In words
                     Skip While word.Substring(0, 1) = "t"

        Dim sb As New System.Text.StringBuilder()

        For Each str As String In query
            sb.AppendLine(str)
            Console.WriteLine(str)
        Next

        Console.ReadLine()

    End Sub

End Module

```

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

```
once  
upon  
a  
was  
a  
jungle
```

Example of Take – Query Expression

VB

```
Module Module1  
  
    Sub Main()  
  
        Dim words = {"once", "upon", "a", "time", "there", "was", "a", "jungle"}  
  
        Dim query = From word In words  
                     Take 3  
  
        Dim sb As New System.Text.StringBuilder()  
  
        For Each str As String In query  
            sb.AppendLine(str)  
            Console.WriteLine(str)  
        Next  
  
        Console.ReadLine()  
  
    End Sub  
  
End Module
```

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

```
once  
upon
```

a

Example of Take While – Query Expression

VB

```
Module Module1

    Sub Main()

        Dim words = {"once", "upon", "a", "time", "there", "was", "a", "jungle"}

        Dim query = From word In words
                     Take While word.Length < 6

        Dim sb As New System.Text.StringBuilder()

        For Each str As String In query
            sb.AppendLine(str)
            Console.WriteLine(str)
        Next

        Console.ReadLine()

    End Sub

End Module
```

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

```
once
upon
a
time
there
was
```

a

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

Example of DefaultIfEmpty – Enumerable.DefaultIfEmpty.Method

C#

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

namespace Operators
{
    class DefaultEmpty
    {
        static void Main(string[] args)
        {
            Pet barley = new Pet() { Name = "Barley", Age = 4 };
        }
    }
}
```

```

    Pet boots = new Pet() { Name = "Boots", Age = 1 };
    Pet whiskers = new Pet() { Name = "Whiskers", Age = 6 };
    Pet bluemoon = new Pet() { Name = "Blue Moon", Age = 9 };
    Pet daisy = new Pet() { Name = "Daisy", Age = 3 };

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

    foreach (var e in pets.DefaultIfEmpty())
    {
        Console.WriteLine("Name = {0} ", e.Name);
    }

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

class Pet
{
    public string Name { get; set; }
    public int Age { get; set; }
}
}
}

```

VB

```
Module Module1
```

```
    Sub Main()
```

```

        Dim barley As New Pet With {.Name = "Barley", .Age = 4}
        Dim boots As New Pet With {.Name = "Boots", .Age = 1}
        Dim whiskers As New Pet With {.Name = "Whiskers", .Age = 6}
        Dim bluemoon As New Pet With {.Name = "Blue Moon", .Age = 9}
        Dim daisy As New Pet With {.Name = "Daisy", .Age = 3}
    
```

```

    Dim pets As New System.Collections.Generic.List(Of Pet)(New Pet()
{barley, boots, whiskers, bluemoon, daisy})

    For Each e In pets.DefaultIfEmpty()
        Console.WriteLine("Name = {0}", e.Name)
    Next

    Console.WriteLine(vbLf & "Press any key to continue.")
    Console.ReadKey()

End Sub

Class Pet
    Public Property Name As String
    Public Property Age As Integer
End Class

End Module

```

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

```

Name = Barley
Name = Boots
Name = Whiskers
Name = Blue Moon
Name = Daisy

Press any key to continue.

```

Example of Range – Enumerable.Range Method

C#

```

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

```

```

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            // Generate a sequence of integers from 1 to 5
            // and then select their squares.

            IEnumerable<int> squares = Enumerable.Range(1, 5).Select(x => x * x);

            foreach (int num in squares)
            {
                Console.WriteLine(num);
            }
            Console.ReadLine();
        }
    }
}

```

VB

```

Module Module1

    Sub Main()

        Dim squares As IEnumerable(Of Integer) = _Enumerable.Range(1,
5).Select(Function(x) x * x)

        Dim output As New System.Text.StringBuilder

        For Each num As Integer In squares
            output.AppendLine(num)
            Console.WriteLine(num)
        Next
        Console.ReadLine()
    End Sub

```



```
End Sub
```

```
End Module
```

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

```
1
4
9
16
25
```

Example of Repeat – Enumerable.Repeat(Of TResult) Method

C#

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

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            IEnumerable<string> strings = Enumerable.Repeat("I like programming.", 3);

            foreach (String str in strings)
            {
                Console.WriteLine(str);
            }
            Console.ReadLine();
        }
    }
}
```

```
}
```

VB

```
Module Module1

    Sub Main()

        Dim sentences As IEnumerable(Of String) = _Enumerable.Repeat("I like
programming.", 3)

        Dim output As New System.Text.StringBuilder

        For Each sentence As String In sentences
            output.AppendLine(sentence)
            Console.WriteLine(sentence)
        Next

        Console.ReadLine()

    End Sub

End Module
```

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

```
I like programming.
I like programming.
I like programming.
```

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

Example of Distinct – Query Expression

VB

```

Module Module1

    Sub Main()

        Dim classGrades = New System.Collections.Generic.List(Of Integer) From
        {63, 68, 71, 75, 68, 92, 75}

        Dim distinctQuery = From grade In classGrades
                             Select grade Distinct

        Dim sb As New System.Text.StringBuilder("The distinct grades are: ")

        For Each number As Integer In distinctQuery
            sb.Append(number & " ")
        Next

        MsgBox(sb.ToString())

    End Sub

```

End Module

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

The distinct grades are: 63 68 71 75 92

Example of Except – Enumerable.Except Method

C#

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

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            double[] numbers1 = { 2.0, 2.1, 2.2, 2.3, 2.4, 2.5 };
            double[] numbers2 = { 2.2 };

            IEnumerable<double> onlyInFirstSet = numbers1.Except(numbers2);

            foreach (double number in onlyInFirstSet)
                Console.WriteLine(number);
            Console.ReadLine();
        }
    }
}
```

VB

Module Module1

Sub Main()

```

Dim numbers1() As Double = {2.0, 2.1, 2.2, 2.3, 2.4, 2.5}
Dim numbers2() As Double = {2.2}

Dim onlyInFirstSet As IEnumerable(Of Double) = numbers1.Except(numbers2)

Dim output As New System.Text.StringBuilder

For Each number As Double In onlyInFirstSet
    output.AppendLine(number)
    Console.WriteLine(number)
Next

Console.ReadLine()

End Sub

End Module

```

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

```

2
2.1
2.3
2.4
2.5

```

Example of Intersect – Enumerable.Intersect Method

C#

```

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

namespace Operators
{
    class Program

```

```

{
    static void Main(string[] args)
    {
        int[] id1 = { 44, 26, 92, 30, 71, 38 };
        int[] id2 = { 39, 59, 83, 47, 26, 4, 30 };

        IEnumerable<int> both = id1.Intersect(id2);

        foreach (int id in both)
            Console.WriteLine(id);
            Console.ReadLine();
        }
    }
}

```

VB

```

Module Module1

    Sub Main()

        Dim id1() As Integer = {44, 26, 92, 30, 71, 38}
        Dim id2() As Integer = {39, 59, 83, 47, 26, 4, 30}

        Dim intersection As IEnumerable(Of Integer) = id1.Intersect(id2)

        Dim output As New System.Text.StringBuilder

        For Each id As Integer In intersection
            output.AppendLine(id)
            Console.WriteLine(id)
        Next

        Console.ReadLine()

    End Sub


```

End Module

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

26
30

Example of Union – Enumerable.Union Method

C#

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

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] ints1 = { 5, 3, 9, 7, 5, 9, 3, 7 };
            int[] ints2 = { 8, 3, 6, 4, 4, 9, 1, 0 };

            IEnumerable<int> union = ints1.Union(ints2);

            foreach (int num in union)
            {
                Console.Write("{0} ", num);
                Console.WriteLine();
            }
            Console.ReadLine();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()

        Dim ints1() As Integer = {5, 3, 9, 7, 5, 9, 3, 7}
        Dim ints2() As Integer = {8, 3, 6, 4, 4, 9, 1, 0}

        Dim union As IEnumerable(Of Integer) = ints1.Union(ints2)

        Dim output As New System.Text.StringBuilder

        For Each num As Integer In union
            output.AppendLine(num & " ")
            Console.WriteLine(num & " ")
        Next

        Console.ReadLine()

    End Sub

End Module
```

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

```
5
3
9
7
8
6
4
1
0
```

Equality

Compares two sentences (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

Example of SequenceEqual – Enumerable.SequenceEqual Method

C#

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

namespace Operators
{
    class SequenceEqual
    {
        static void Main(string[] args)
        {
            Pet barley = new Pet() { Name = "Barley", Age = 4 };
            Pet boots = new Pet() { Name = "Boots", Age = 1 };
            Pet whiskers = new Pet() { Name = "Whiskers", Age = 6 };

            List<Pet> pets1 = new List<Pet>() { barley, boots };
            List<Pet> pets2 = new List<Pet>() { barley, boots };
            List<Pet> pets3 = new List<Pet>() { barley, boots, whiskers };

            bool equal = pets1.SequenceEqual(pets2);
            bool equal3 = pets1.SequenceEqual(pets3);

            Console.WriteLine("The lists pets1 and pets2 {0} equal.", equal ?
"are" : "are not");

            Console.WriteLine("The lists pets1 and pets3 {0} equal.", equal3 ?
"are" : "are not");
        }
    }
}
```

```

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

    class Pet
    {
        public string Name { get; set; }
        public int Age { get; set; }
    }
}

```

VB

```

Module Module1

    Sub Main()

        Dim barley As New Pet With {.Name = "Barley", .Age = 4}
        Dim boots As New Pet With {.Name = "Boots", .Age = 1}
        Dim whiskers As New Pet With {.Name = "Whiskers", .Age = 6}

        Dim pets1 As New System.Collections.Generic.List(Of Pet)(New Pet()
{barley, boots})
        Dim pets2 As New System.Collections.Generic.List(Of Pet)(New Pet()
{barley, boots})
        Dim pets3 As New System.Collections.Generic.List(Of Pet)(New Pet()
{barley, boots, whiskers})

        Dim equal As Boolean = pets1.SequenceEqual(pets2)
        Dim equal3 As Boolean = pets1.SequenceEqual(pets3)

        Console.WriteLine("The lists pets1 and pets2 {0} equal.", IIf(equal,
"are", "are not"))

        Console.WriteLine("The lists pets1 and pets3 {0} equal.", IIf(equal3,
"are", "are not"))

        Console.WriteLine(vbLf & "Press any key to continue.")
    End Sub
End Module

```

```

        Console.ReadKey()

    End Sub

    Class Pet
        Public Property Name As String
        Public Property Age As Integer
    End Class

End Module

```

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

```

The lists pets1 and pets2 are equal.
The lists pets1 and pets3 are not equal.

Press any key to continue.

```

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	Not Applicable	Not Applicable

	case there is no existence of such elements		
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

Example of ElementAt – Enumerable.ElementAt Method

C#

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

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] names = { "Hartono, Tommy", "Adams, Terry", "Andersen, Henriette Thaulow", "Hedlund, Magnus", "Ito, Shu" };
        }
    }
}
```

```

        Random random = new Random(DateTime.Now.Millisecond);

        string name = names.ElementAt(random.Next(0, names.Length));

        Console.WriteLine("The name chosen at random is '{0}'.", name);
        Console.ReadLine();
    }
}
}

```

VB

```

Module Module1

    Sub Main()

        Dim names() As String = _{"Hartono, Tommy", "Adams, Terry", "Andersen,
Henriette Thaulow", "Hedlund, Magnus", "Ito, Shu"}

        Dim random As Random = New Random(DateTime.Now.Millisecond)

        Dim name As String = names.ElementAt(random.Next(0, names.Length))

        MsgBox("The name chosen at random is " & name)

    End Sub

End Module

```

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

```
The name chosen at random is Ito, Shu
```

Note – Here, the above output will change dynamically and names will be chosen randomly.

Example of First – Enumerable.First Method

C#

```

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

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = { 9, 34, 65, 92, 87, 435, 3, 54, 83, 23, 87, 435, 67,
12, 19 };

            int first = numbers.First();

            Console.WriteLine(first);
            Console.ReadLine();
        }
    }
}

```

VB

```

Module Module1

    Sub Main()

        Dim numbers() As Integer = {9, 34, 65, 92, 87, 435, 3, 54, 83, 23, 87,
435, 67, 12, 19}

        Dim first As Integer = numbers.First()

        MsgBox(first)

    End Sub

```

End Module

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

9

Example of Last – Enumerable.Last Method

C#

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

namespace Operators
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = { 9, 34, 65, 92, 87, 435, 3, 54, 83, 23, 87, 435, 67,
12, 19 };

            int last = numbers.Last();

            Console.WriteLine(last);
            Console.ReadLine();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
```

```
Dim numbers() As Integer = {9, 34, 65, 92, 87, 435, 3, 54, 83, 23, 87, 435, 67, 12, 19}

Dim last As Integer = numbers.Last()

MsgBox(last)

End Sub

End Module
```

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

19

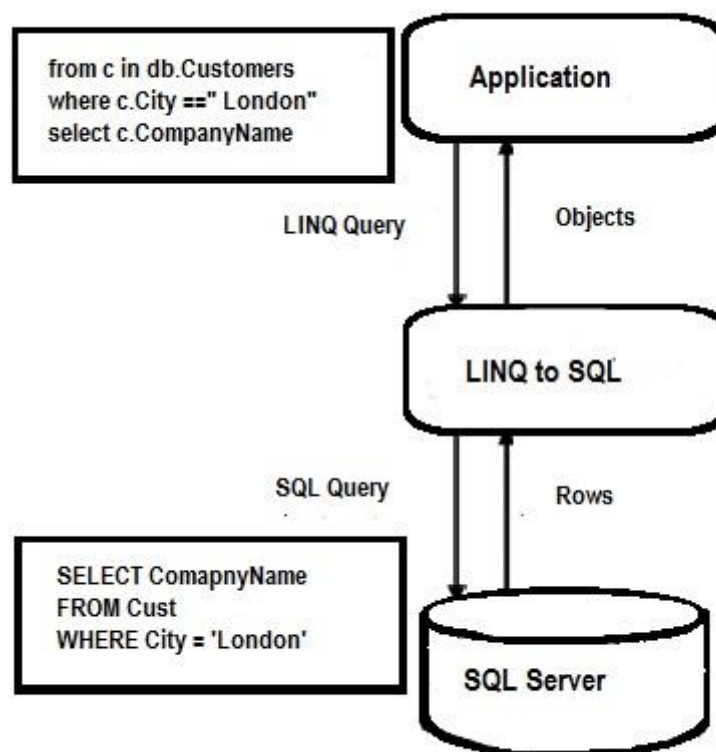
4. SQL

LINQ to SQL offers an infrastructure (run-time) for the management of relational data as objects. It is a component of version 3.5 of the .NET Framework and ably does the translation of language-integrated queries of the object model into SQL. These queries are then sent to the database for the purpose of execution. After obtaining the results from the database, LINQ to SQL again translates them to objects.

Introduction of LINQ To SQL

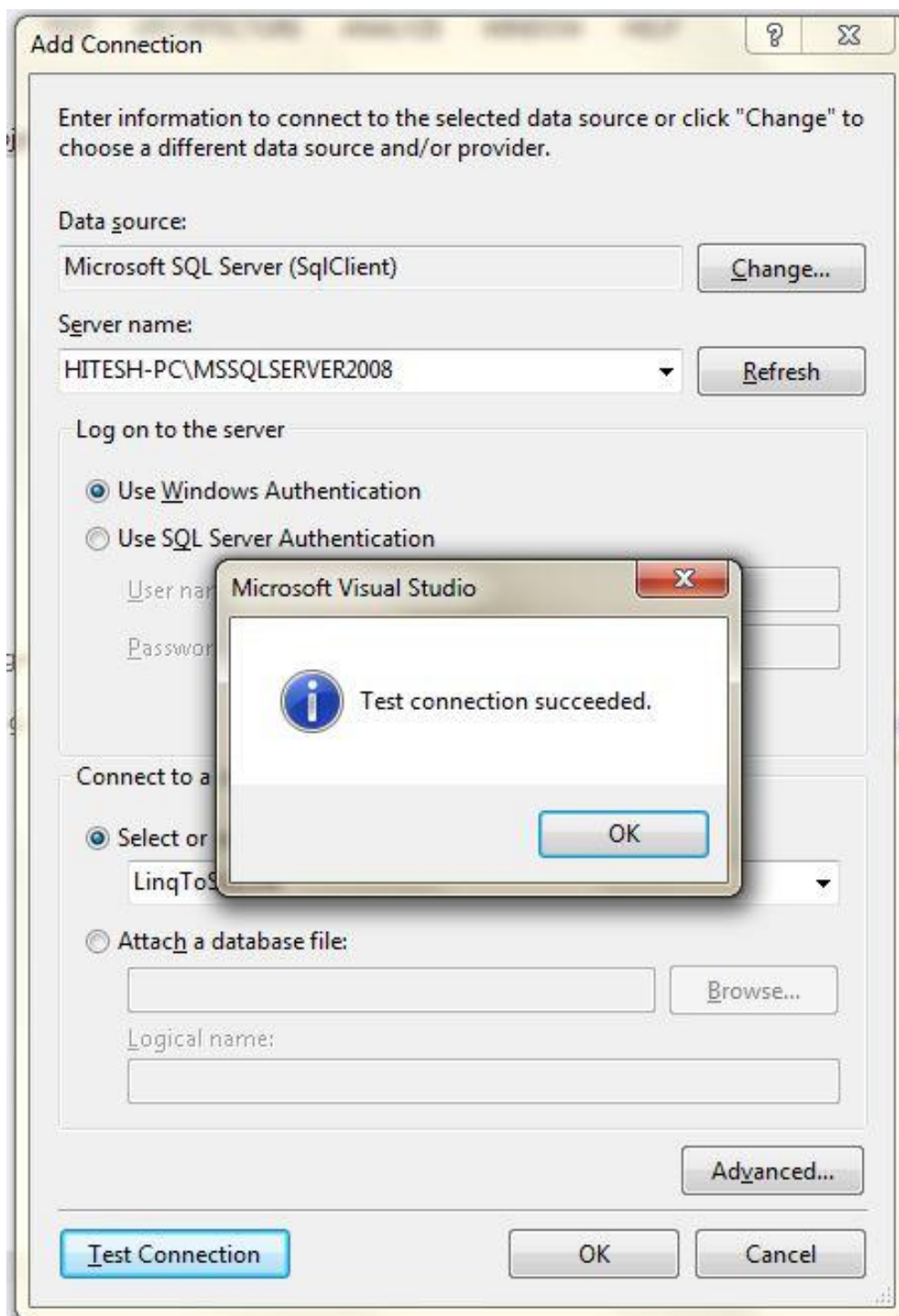
For most ASP.NET developers, LINQ to SQL (also known as DLINQ) is an electrifying part of Language Integrated Query as this allows querying data in SQL server database by using usual LINQ expressions. It also allows to update, delete, and insert data, but the only drawback from which it suffers is its 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.

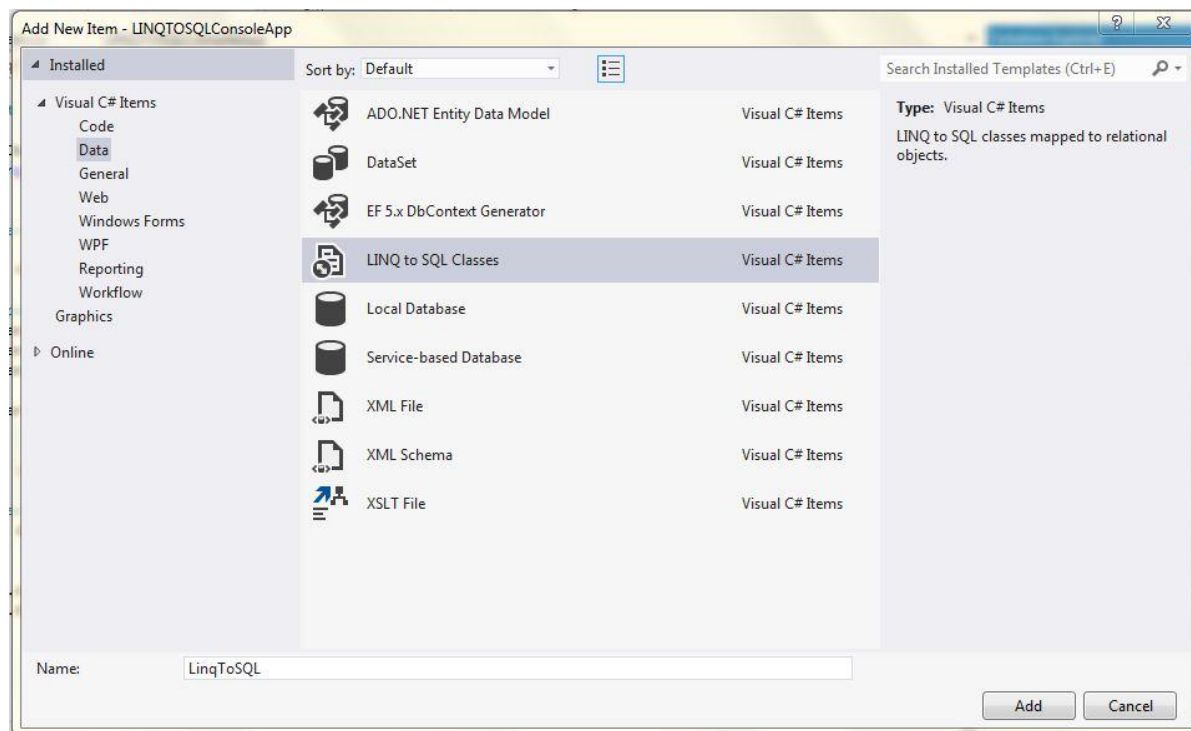


How to Use LINQ to SQL?

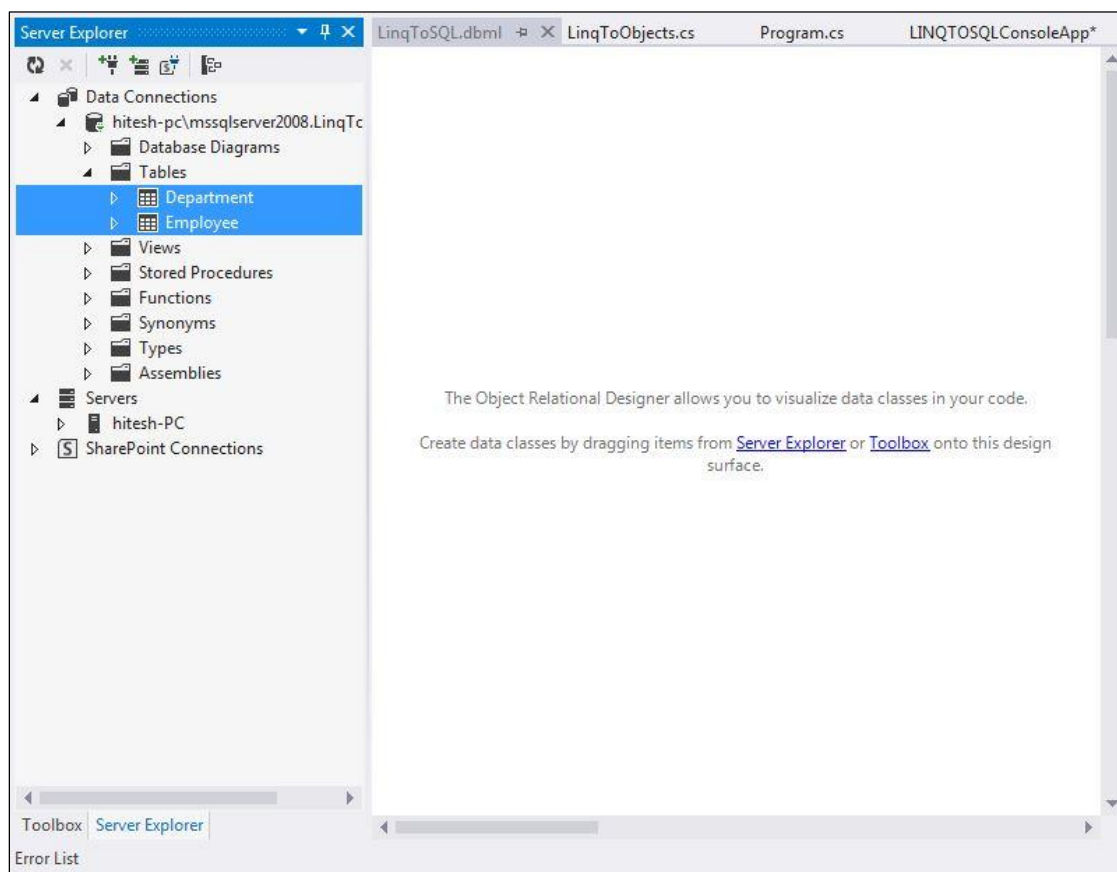
Step 1 – Make a new “Data Connection” with database server. View → Server Explorer → Data Connections → Add Connection

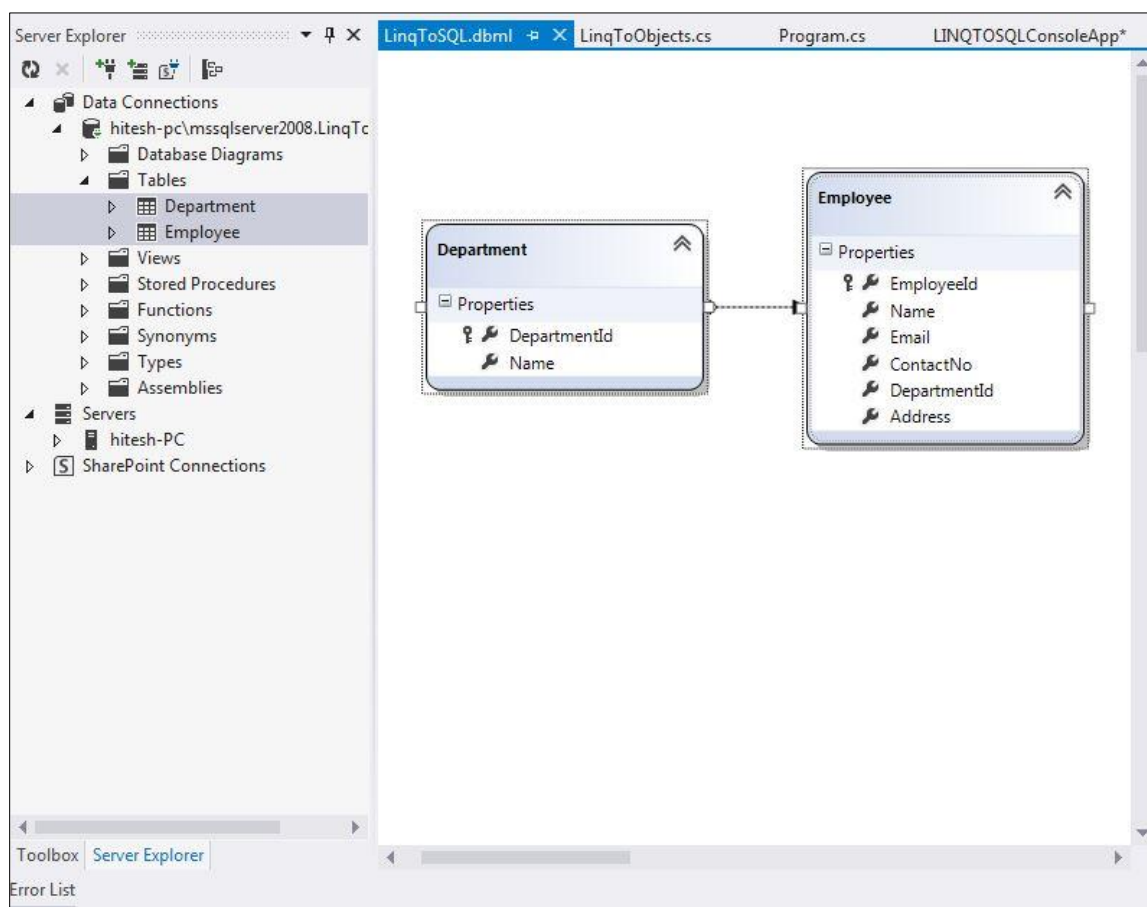


Step 2 – Add LINQ To SQL class file



Step 3 – Select tables from database and drag and drop into the new LINQ to SQL class file.



Step 4 – Added tables to class file.

Querying with LINQ to SQL

The rules for executing a query with LINQ to SQL is similar to that of a standard LINQ query i.e. query is executed either deferred or immediate. There are various components that play a role in execution of a query with LINQ to SQL and these are the following ones.

- **LINQ to SQL API** – requests query execution on behalf of an application and sent it to LINQ to SQL Provider
- **LINQ to SQL Provider** – converts query to Transact SQL(T-SQL) and sends the new query to the ADO Provider for execution
- **ADO Provider** – After execution of the query, send the results in the form of a DataReader to LINQ to SQL Provider which in turn converts it into a form of user object

It should be noted that before executing a LINQ to SQL query, it is vital to connect to the data source via DataContext class.

Insert, Update, and Delete using LINQ to SQL

Add OR Insert

C#

```
using System;
using System.Linq;

namespace LINQtoSQL
{
    class LinqToSQLCRUD
    {
        static void Main(string[] args)
        {
            string connectionString =
System.Configuration.ConfigurationManager.ConnectionStrings["LinqToSQLDBConnect
ionString"].ToString();

            LinqToSQLDataContext db = new LinqToSQLDataContext(connectionString);

            //Create new Employee

            Employee newEmployee = new Employee();
            newEmployee.Name = "Michael";
            newEmployee.Email = "yourname@companyname.com";
            newEmployee.ContactNo = "343434343";
            newEmployee.DepartmentId = 3;
            newEmployee.Address = "Michael - USA";

            //Add new Employee to database
            db.Employees.InsertOnSubmit(newEmployee);

            //Save changes to Database.
            db.SubmitChanges();

            //Get new Inserted Employee
```

```

        Employee insertedEmployee = db.Employees.FirstOrDefault(e
⇒e.Name.Equals("Michael"));

        Console.WriteLine("Employee Id = {0} , Name = {1}, Email = {2},
ContactNo = {3}, Address = {4}",
                        insertedEmployee.EmployeeId, insertedEmployee.Name,
insertedEmployee.Email,
                        insertedEmployee.ContactNo,
insertedEmployee.Address);

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

```

VB

```

Module Module1

    Sub Main()

        Dim connectString As String =
System.Configuration.ConfigurationManager.ConnectionStrings("LinQToSQLDBConnect
ionString").ToString()

        Dim db As New LinQToSQLDataContext(connectString)

        Dim newEmployee As New Employee()

        newEmployee.Name = "Michael"
        newEmployee.Email = "yourname@companyname.com"
        newEmployee.ContactNo = "343434343"
        newEmployee.DepartmentId = 3
        newEmployee.Address = "Michael - USA"

        db.Employees.InsertOnSubmit(newEmployee)

        db.SubmitChanges()
    End Sub
End Module

```

```

        Dim insertedEmployee As Employee =
db.Employees.FirstOrDefault(Function(e) e.Name.Equals("Michael"))

        Console.WriteLine("Employee Id = {0} , Name = {1}, Email = {2}, ContactNo
= {3}, Address = {4}", insertedEmployee.EmployeeId, insertedEmployee.Name,
insertedEmployee.Email, insertedEmployee.ContactNo, insertedEmployee.Address)

        Console.WriteLine(vbLf & "Press any key to continue.")
        Console.ReadKey()

    End Sub

End Module

```

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

```

Employee ID = 4, Name = Michael, Email = yourname@companyname.com, ContactNo =
343434343, Address = Michael - USA

Press any key to continue.

```

Update

C#

```

using System;
using System.Linq;

namespace LINQtoSQL
{
    class LinqToSQLCRUD
    {
        static void Main(string[] args)
        {
            string connectionString =
System.Configuration.ConfigurationManager.ConnectionStrings["LinqToSQLDBConnect
ionString"].ToString();

            LinqToSQLDataContext db = new LinqToSQLDataContext(connectionString);

```

```

        //Get Employee for update
        Employee employee = db.Employees.FirstOrDefault(e
=>e.Name.Equals("Michael"));

        employee.Name = "George Michael";
        employee.Email = "yourname@companyname.com";
        employee.ContactNo = "999999999";
        employee.DepartmentId = 2;
        employee.Address = "Michael George - UK";

        //Save changes to Database.
        db.SubmitChanges();

        //Get Updated Employee
        Employee updatedEmployee = db.Employees.FirstOrDefault(e
=>e.Name.Equals("George Michael"));

        Console.WriteLine("Employee Id = {0} , Name = {1}, Email = {2},
ContactNo = {3}, Address = {4}",
                        updatedEmployee.EmployeeId, updatedEmployee.Name,
updatedEmployee.Email,
                        updatedEmployee.ContactNo, updatedEmployee.Address);

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

```

VB

```
Module Module1
```

```
Sub Main()
```

```

    Dim connectionString As String =
System.Configuration.ConfigurationManager.ConnectionStrings("LinqToSQLDBConnect
ionString").ToString()

```



```

Dim db As New LinqToSQLDataContext(connectString)

Dim employee As Employee = db.Employees.FirstOrDefault(Function(e)
e.Name.Equals("Michael"))

employee.Name = "George Michael"
employee.Email = "yourname@companyname.com"
employee.ContactNo = "999999999"
employee.DepartmentId = 2
employee.Address = "Michael George - UK"

db.SubmitChanges()

Dim updatedEmployee As Employee = db.Employees.FirstOrDefault(Function(e)
e.Name.Equals("George Michael"))

Console.WriteLine("Employee Id = {0} , Name = {1}, Email = {2}, ContactNo
= {3}, Address = {4}", updatedEmployee.EmployeeId, updatedEmployee.Name,
updatedEmployee.Email, updatedEmployee.ContactNo, updatedEmployee.Address)

Console.WriteLine(vbLf & "Press any key to continue.")
Console.ReadKey()

End Sub

End Module

```

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

```

Employee ID = 4, Name = George Michael, Email = yourname@companyname.com,
ContactNo =
999999999, Address = Michael George - UK

```

```

Press any key to continue.

```

Delete

C#

```
using System;
using System.Linq;

namespace LINQtoSQL
{
    class LinqToSQLCRUD
    {
        static void Main(string[] args)
        {
            string connectionString =
System.Configuration.ConfigurationManager.ConnectionStrings["LinqToSQLDBConnect
ionString"].ToString();

            LinqToSQLDataContext db = newLinqToSQLDataContext(connectionString);

            //Get Employee to Delete
            Employee deleteEmployee = db.Employees.FirstOrDefault(e
=>e.Name.Equals("George Michael"));

            //Delete Employee
            db.Employees.DeleteOnSubmit(deleteEmployee);

            //Save changes to Database.
            db.SubmitChanges();

            //Get All Employee from Database
            var employeeList = db.Employees;

            foreach (Employee employee in employeeList)
            {
                Console.WriteLine("Employee Id = {0} , Name = {1}, Email = {2},
ContactNo = {3}", employee.EmployeeId, employee.Name, employee.Email,
employee.ContactNo);
            }
        }
    }
}
```

```

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

```

VB

```

Module Module1

    Sub Main()

        Dim connectionString As String =
System.Configuration.ConfigurationManager.ConnectionStrings("LinqToSQLDBConnect
ionString").ToString()

        Dim db As New LinqToSQLDataContext(connectionString)

        Dim deleteEmployee As Employee = db.Employees.FirstOrDefault(Function(e)
e.Name.Equals("George Michael"))

        db.Employees.DeleteOnSubmit(deleteEmployee)

        db.SubmitChanges()

        Dim employeeList = db.Employees

        For Each employee As Employee In employeeList
            Console.WriteLine("Employee Id = {0} , Name = {1}, Email = {2},
ContactNo = {3}", employee.EmployeeId, employee.Name, employee.Email,
employee.ContactNo)
        Next

        Console.WriteLine(vbLf & "Press any key to continue.")

        Console.ReadKey()

    End Sub

End Module

```

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

```
Employee ID = 1, Name = William, Email = abc@gy.co, ContactNo = 999999999  
Employee ID = 2, Name = Miley, Email = amp@esds.sds, ContactNo = 999999999  
Employee ID = 3, Name = Benjamin, Email = asdsad@asdsa.dsd, ContactNo =
```

Press any key to continue.

5. 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 (tools) 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();
        }
    }
}

```

VB

```

Imports System.Collections.Generic
Imports System.Linq

Module Module1

    Sub Main(ByVal args As String())

```

```

    Dim account As New Department With {.Name = "Account", .DepartmentId = 1}
    Dim sales As New Department With {.Name = "Sales", .DepartmentId = 2}
    Dim marketing As New Department With {.Name = "Marketing", .DepartmentId
= 3}

    Dim departments As New System.Collections.Generic.List(Of Department)(New
Department() {account, sales, marketing})

    Dim departmentList = From d In departments

    For Each dept In departmentList
        Console.WriteLine("Department Id = {0} , Department Name = {1}",
dept.DepartmentId, dept.Name)
    Next

    Console.WriteLine(vbLf & "Press any key to continue.")
    Console.ReadKey()

End Sub

Class Department
    Public Property Name As String
    Public Property DepartmentId As Integer
End Class

End Module

```

When the above code of C# or VB 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.

```


6. DATASET

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.

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["LinqToSQLDBConnect
ionString"].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();
}
}
}

```

VB

```

Imports System.Data.SqlClient
Imports System.Linq

Module LinqToDataSet

    Sub Main()

        Dim connectionString As String =
System.Configuration.ConfigurationManager.ConnectionStrings("LinqToSQLDBConnect
ionString").ToString()

        Dim sqlSelect As String = "SELECT * FROM Department;" + "SELECT * FROM
Employee;"
        Dim sqlCnn As SqlConnection = New SqlConnection(connectionString)
        sqlCnn.Open()

        Dim da As New SqlDataAdapter
        da.SelectCommand = New SqlCommand(sqlSelect, sqlCnn)

        da.TableMappings.Add("Table", "Department")
        da.TableMappings.Add("Table1", "Employee")

        Dim ds As New DataSet()
        da.Fill(ds)

        Dim dr As DataRelation = ds.Relations.Add("FK_Employee_Department",
ds.Tables("Department").Columns("DepartmentId"),
ds.Tables("Employee").Columns("DepartmentId"))
    End Sub
End Module

```

```

Dim department As DataTable = ds.Tables("Department")
Dim employee As DataTable = ds.Tables("Employee")

Dim query = From d In department.AsEnumerable()
             Join e In employee.AsEnumerable() On d.Field(Of Integer)("DepartmentId") Equals
Integer)("DepartmentId") Equals
             e.Field(Of Integer)("DepartmentId")
             Select New Person With{ _
                 .EmployeeId = e.Field(Of Integer)("EmployeeId"),
                 .EmployeeName = e.Field(Of String)("Name"),
                 .DepartmentId = d.Field(Of Integer)("DepartmentId"),
                 .DepartmentName = d.Field(Of String)("Name")
             }

For Each e In query
    Console.WriteLine("Employee Id = {0} , Name = {1} , Department Name = {2}", e.EmployeeId, e.EmployeeName, e.DepartmentName)
Next

Console.WriteLine(vbLf & "Press any key to continue.")
Console.ReadKey()

End Sub

Class Person
    Public Property EmployeeId As Integer
    Public Property EmployeeName As String
    Public Property DepartmentId As Integer
    Public Property DepartmentName As String
End Class

End Module

```

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

```
Employee Id = 1, Name = William, Department Name = Account
Employee Id = 2, Name = Benjamin, Department Name = Account
Employee Id = 3, Name = Miley, Department Name = Sales

Press any key to continue.
```

Querying Dataset using Linq to Dataset

Before beginning querying a Dataset using LINQ to Dataset, it is vital to load data to a Dataset and this is done by either using DataAdapter class or by LINQ to SQL. Formulation of queries using LINQ to Dataset is quite similar to formulating queries by using LINQ alongside other LINQ enabled data sources.

Single-Table Query

In the following single-table query, all online orders are collected from the SalesOrderHeaderTtable and then order ID, Order date as well as order number are displayed as output.

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 SingleTable
    {
        static void Main(string[] args)
        {
            string connectionString =
System.Configuration.ConfigurationManager.ConnectionStrings["LinqToSQLDBConnect
ionString"].ToString();

            string sqlSelect = "SELECT * FROM Department;";
```

```

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

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

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

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

var query = from d in department.AsEnumerable()

select new
{
    DepartmentId = d.Field<int>("DepartmentId"),
    DepartmentName = d.Field<string>("Name")
};

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

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

```

VB

```

Imports System.Data.SqlClient
Imports System.Linq

```

```
Module LinqToDataSet
```

```

Sub Main()

    Dim connectionString As String =
System.Configuration.ConfigurationManager.ConnectionStrings("LinqToSQLDBConnect
ionString").ToString()

    Dim sqlSelect As String = "SELECT * FROM Department;"
    Dim sqlCnn As SqlConnection = New SqlConnection(connectionString)
    sqlCnn.Open()

    Dim da As New SqlDataAdapter
    da.SelectCommand = New SqlCommand(sqlSelect, sqlCnn)

    da.TableMappings.Add("Table", "Department")
    Dim ds As New DataSet()
    da.Fill(ds)

    Dim department As DataTable = ds.Tables("Department")

    Dim query = From d In department.AsEnumerable()
    Select New DepartmentDetail With {.DepartmentId = d.Field(Of
Integer)("DepartmentId"), .DepartmentName = d.Field(Of String)("Name")}

    For Each e In query
        Console.WriteLine("Department Id = {0} , Name = {1}", e.DepartmentId,
e.DepartmentName)
    Next

    Console.WriteLine(vbLf & "Press any key to continue.")
    Console.ReadKey()

End Sub

Public Class DepartmentDetail
    Public Property DepartmentId As Integer

```

```
Public Property DepartmentName As String  
End Class  
End Module
```

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

```
Department Id = 1, Name = Account  
Department Id = 2, Name = Sales  
Department Id = 3, Name = Pre-Sales  
Department Id = 4, Name = Marketing  
Press any key to continue.
```


7. 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 querying 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 an 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();
    }
}
}

```

VB

```

Imports System.Collections.Generic
Imports System.Linq
Imports System.Xml.Linq

Module Module1

    Sub Main(ByVal args As String())

        Dim myXML As String = "<Departments>" & vbCrLf &
                                "<Department>Account</Department>" & vbCrLf &
                                "<Department>Sales</Department>" & vbCrLf &
                                "<Department>Pre-Sales</Department>" & vbCrLf &
                                "<Department>Marketing</Department>" & vbCrLf &
                                "</Departments>"

        Dim xdoc As New XDocument()
        xdoc = XDocument.Parse(myXML)

        Dim result = xdoc.Element("Departments").Descendants()

        For Each item As XElement In result
            Console.WriteLine("Department Name - " + item.Value)
        Next

        Console.WriteLine(vbLf & "Press any key to continue.")
        Console.ReadKey()

    End Sub

End Module

```

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

```
Department Name - Account
Department Name - Sales
Department Name - Pre-Sales
Department Name - Marketing

Press any key to continue.
```

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();
    }
}
}

```

VB

```

Imports System.Collections.Generic
Imports System.Linq
Imports System.Xml.Linq

Module Module1

    Sub Main(ByVal args As String())

        Dim myXML As String = "<Departments>" & vbCrLf &
            "<Department>Account</Department>" & vbCrLf &
            "<Department>Sales</Department>" & vbCrLf &
            "<Department>Pre-Sales</Department>" & vbCrLf &
            "<Department>Marketing</Department>" & vbCrLf &
            "</Departments>"

        Dim xdoc As New XDocument()

        xdoc = XDocument.Parse(myXML)
    
```

```

        xdoc.Element("Departments").Add(New XElement("Department", "Finance"))

        xdoc.Element("Departments").AddFirst(New XElement("Department",
"Support"))

        Dim result = xdoc.Element("Departments").Descendants()

        For Each item As XElement In result
            Console.WriteLine("Department Name - " + item.Value)
        Next

        Console.WriteLine(vbLf & "Press any key to continue.")
        Console.ReadKey()

    End Sub

End Module

```

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();
        }
    }
}

```

VB

```
Imports System.Collections.Generic
Imports System.Linq
Imports System.Xml.Linq

Module Module1

    Sub Main(args As String())

        Dim myXML As String = "<Departments>" & vbCrLf &
            "<Department>Support</Department>" & vbCrLf &
            "<Department>Account</Department>" & vbCrLf &
            "<Department>Sales</Department>" & vbCrLf &
            "<Department>Pre-Sales</Department>" & vbCrLf &
            "<Department>Marketing</Department>" & vbCrLf &
            "<Department>Finance</Department>" & vbCrLf &
            "</Departments>"

        Dim xdoc As New XDocument()
        xdoc = XDocument.Parse(myXML)

        xdoc.Descendants().Where(Function(s) s.Value = "Sales").Remove()

        Dim result = xdoc.Element("Departments").Descendants()

        For Each item As XElement In result
            Console.WriteLine("Department Name - " + item.Value)
        Next

        Console.WriteLine(vbLf & "Press any key to continue.")
        Console.ReadKey()

    End Sub

End Module
```


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 - Pre-Sales
Department Name - Marketing
Department Name - Finance

Press any key to continue.
```

8. 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.

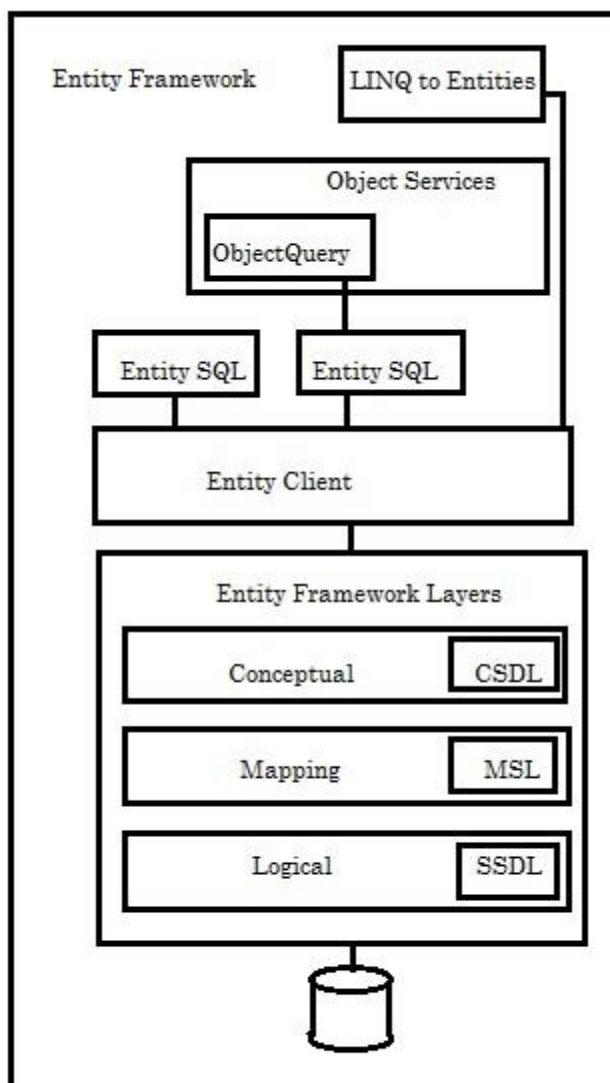
LINQ to Entities has for these advantages become the standard mechanism for the usage of LINQ on databases nowadays. It is also possible with LINQ to Entities to change queried data details and committing a batch update easily. What is the most intriguing fact about LINQ to Entities is that it has same 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** (Entity Connection)
- Composing a query either in C# or Visual Basic (VB) 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.

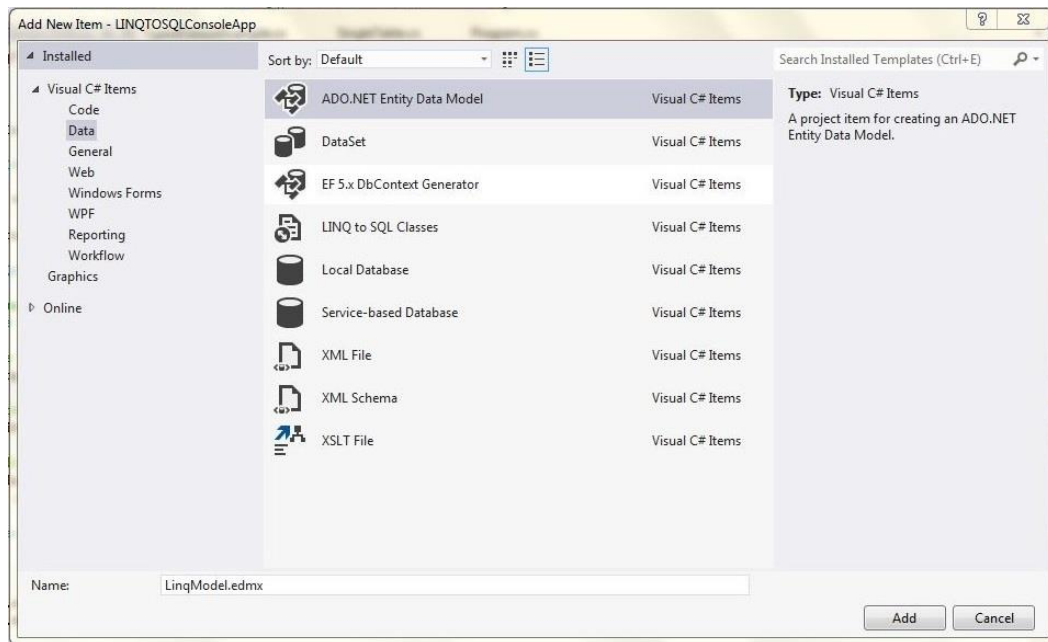
The Entity Framework, on the other hand, is actually **Object Relational Mapper** abbreviated generally as ORM by the developers that does the generation of business objects as well as entities as per the database tables and facilitates various basic operations like create, update, delete and read. The following illustration shows the entity framework and its components.



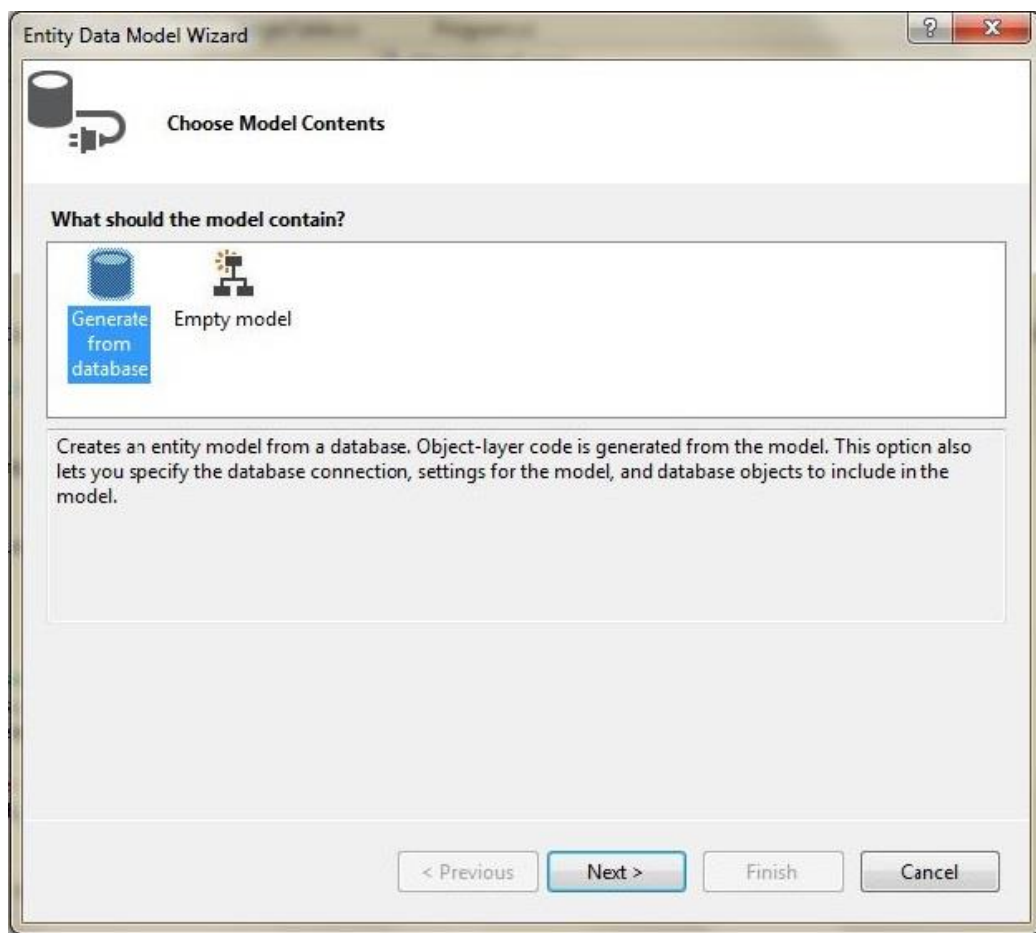
Example of ADD, UPDATE, and DELETE using LINQ with Entity Model

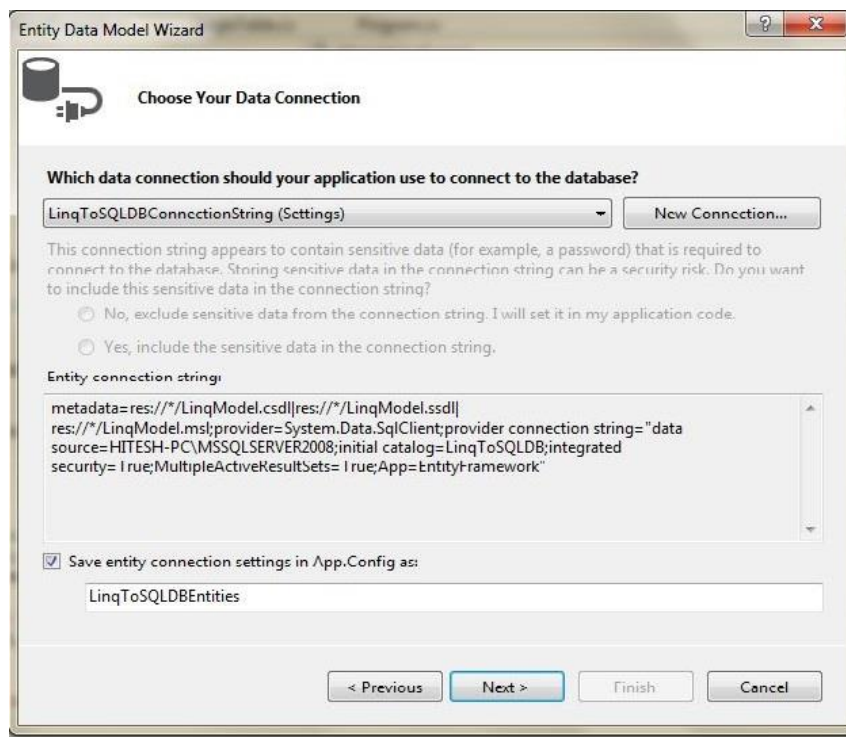
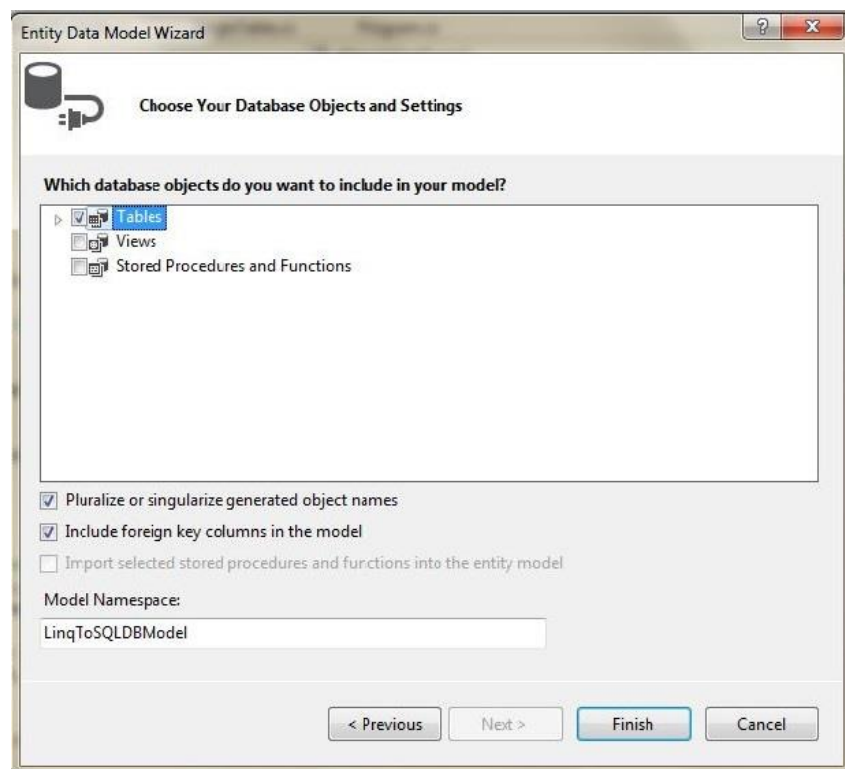
First add Entity Model by following below steps.

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 from the drop-down menu.**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");

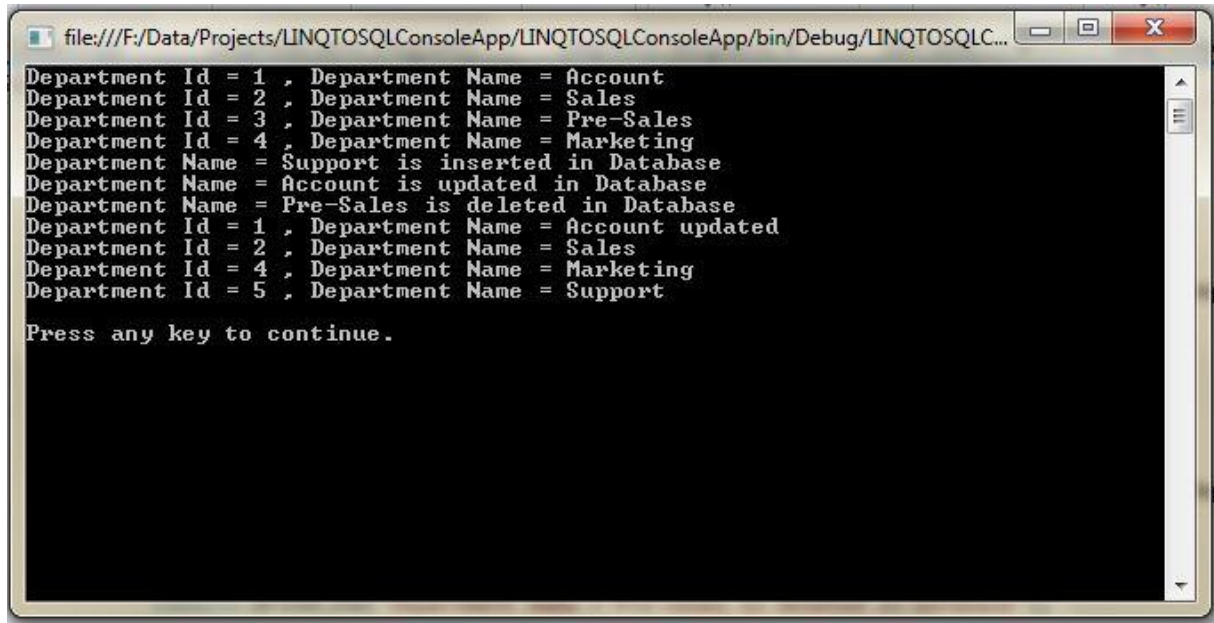
        //Get the Updated List of Departments from Database
        departmentList = from d in context.Departments
        select d;

        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:



The screenshot shows a Windows console window titled "file:///F:/Data/Projects/LINQTOSQLConsoleApp/LINQTOSQLConsoleApp/bin/Debug/LINQTOSQLC...". The console output displays a series of LINQ queries and their results, including inserting, updating, and deleting records in a database. The output is as follows:

```
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.
```


9. 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 –

(Input parameters) \Rightarrow Expression or statement block

Here is an example of a 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  $\Rightarrow$  y * y;
            int j = myDelegate(5);
            Console.WriteLine(j);
            Console.ReadLine();
        }
    }
}
```

VB

```
Module Module1

    Private Delegate Function del(ByVal i As Integer) As Integer

    Sub Main(ByVal args As String())

        Dim myDelegate As del = Function(y) y * y
        Dim j As Integer = myDelegate(5)
        Console.WriteLine(j)
        Console.ReadLine()

    End Sub

End Module
```

When the above code of C# or VB 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();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()

        Dim fibNum As Integer() = {1, 1, 2, 3, 5, 8, 13, 21, 34}
        Dim averageValue As Double = fibNum.Where(Function(num) num Mod 2 = 1).Average()

        Console.WriteLine(averageValue)
        Console.ReadLine()

    End Sub

End Module
```

End Module

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

7.33333333333333

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.

<code>delegate int Transformer (int i);</code>
--

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.

<pre>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;</pre>
--

```

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. They are 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.

Points to remember while using lambda expressions

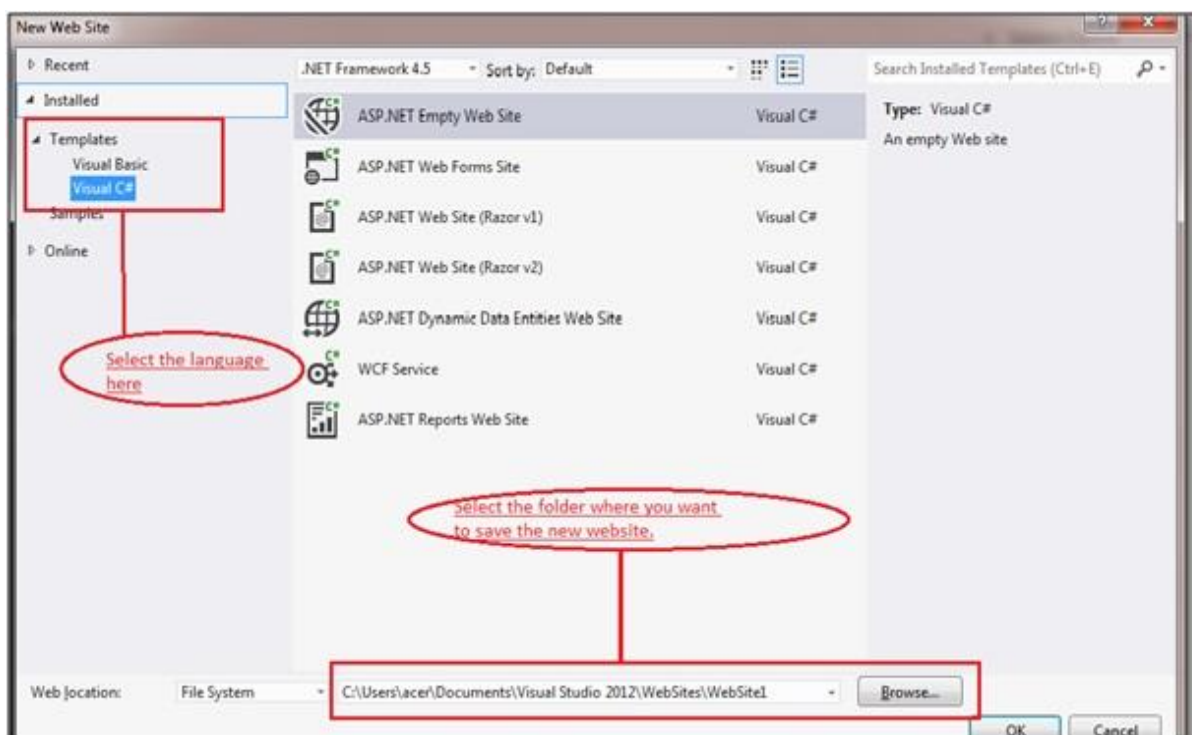
- A lambda expression can return a value and may have parameters.
- Parameters can be defined in a myriad of ways with a lambda expression.
- If there is single statement in a lambda expression, there is no need of curly brackets whereas if there are multiple statements, curly brackets as well as return 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.

10. ASP.NET

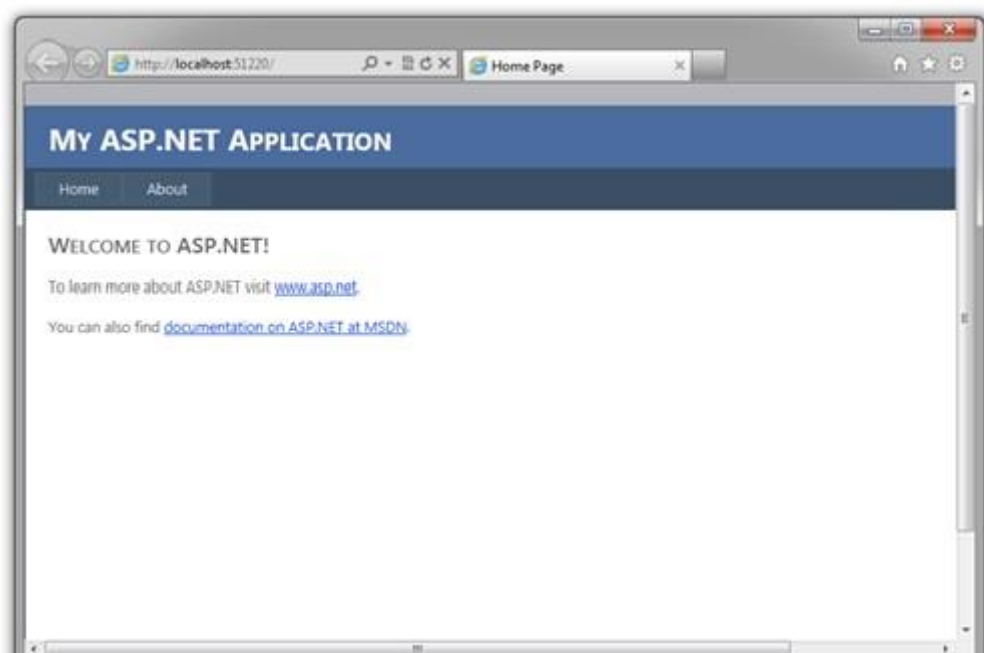
As a set of .NET framework extensions, LINQ is the preferred mechanism for data access by ASP.NET developers. ASP.NET 3.5 has a built-in tool LINQDataSource control that enables usage of LINQ easily in ASP.NET. ASP.NET uses the above-mentioned control as a data source. Real life projects mostly encompass websites or windows applications and so to understand better the concept of LINQ with ASP.NET, let's start with creating a ASP.NET website that make use of the LINQ features.

For this, it is essential to get installed Visual Studio and .NET framework on your system. Once you have opened Visual Studio, go to File → New → Website. A pop up window will open as shown in below figure.



Now, under the templates in the left hand side, there will be two language options to create the website. Choose **Visual C#** and select **ASP.NET Empty Web Site**.

Select the folder where you want to save new website on your system. Then press **OK** and soon **Solution Explorer** appears on your screen containing all the web files. Right click on Default.aspx in the Solution Explorer and choose View in Browser to view the default ASP.NET website in the browser. Soon your new ASP.NET website will open in the web browser, as shown in the following screenshot.



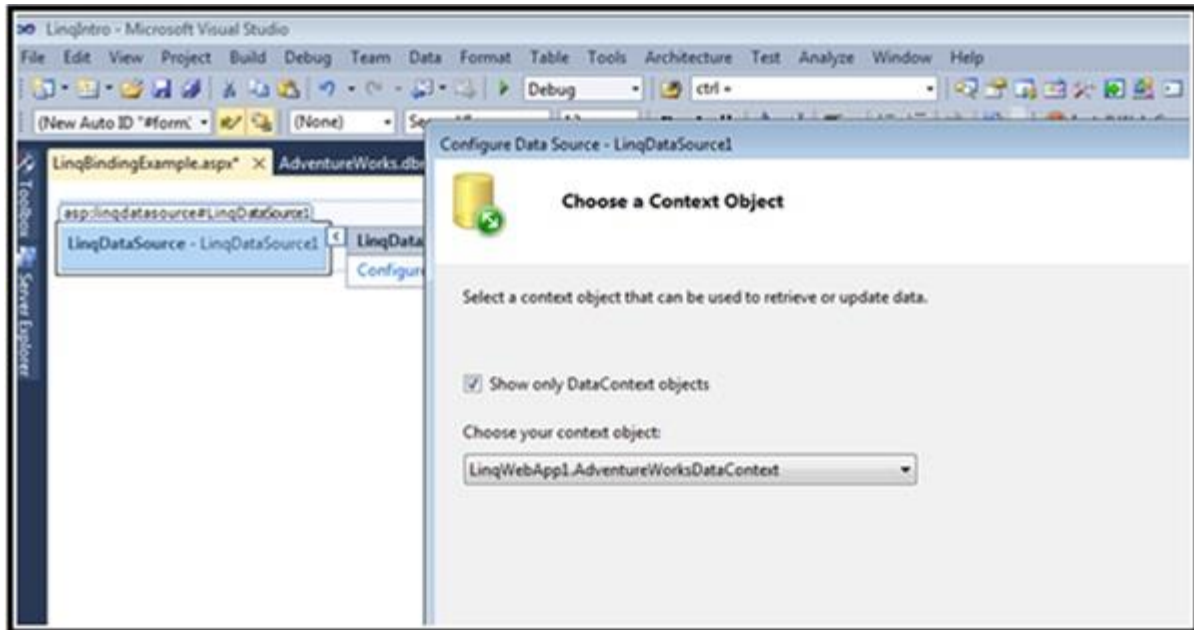
.aspx is in fact the major file extension used in ASP.NET websites. Visual Studio by default creates all the necessary pages for a basic website like **Home page** and **About Us** page where you can place your content conveniently. The code for the website is generated automatically here and can be viewed too.

LINQDataSource Control

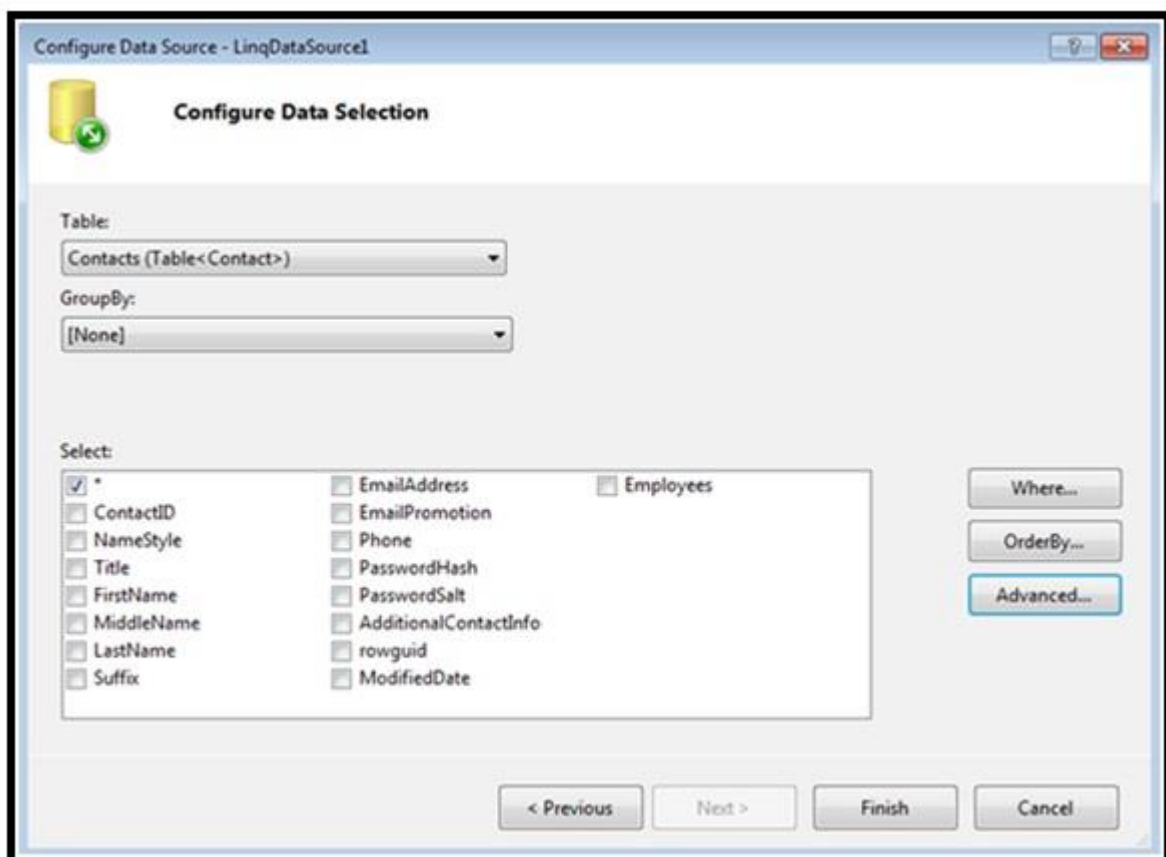
It is possible to **UPDATE**, **INSERT** and **DELETE** data in the pages of ASP.NET website with the help of LINQDataSource control. There is absolutely no need for specification of SQL commands as LINQDataSource control employs dynamically created commands for such operations.

The control enables a user to make use of LINQ in an ASP.NET web page conveniently by property setting in the markup text. LINQDataSource is very similar to that of controls like **SqlDataSource** as well as **ObjectDataSource** as it can be used in binding other ASP.NET controls present on a page to a data source. So, we must have a **database** to explain the various functions invoked by the LINQDataSource Control.

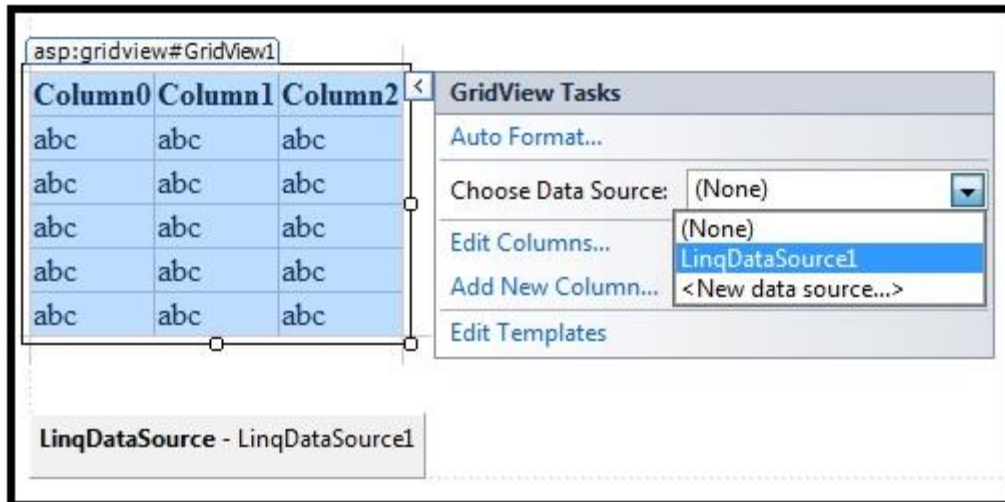
Before going to start explanation of the control usage in ASP.NET web page form, it is essential to open the Microsoft Visual Studio Toolbox and drag and drop LINQDataSource control to .aspx page of ASP.NET website like below figure.



The next step is to configure LINQDataSource by selecting all the columns for the employee record.



Now add a GridView Control to the .aspx page and configure it like shown in below figure. The GridView control is powerful and offers flexibility to work with the data. Soon after configuring the control, it will appear in the browser.



The coding that can be viewed now on your screen for the .aspx page will be –

```
<!DOCTYPE html>

<html>

    <head runat="server">
        <title></title>
    </head>

    <body>

        <form id="form1" runat="server">

            <div>

                <asp:GridView ID="GridView1" runat="server"
AutoGenerateColumns="False"
                DataKeyNames="ContactID" DataSourceID="LINQDataSource1">

                    <Columns>

                        <asp:BoundField DataField="ContactID" HeaderText="ContactID"
                            InsertVisible="False" ReadOnly="True" SortExpression="ContactID" />

                        <asp:CheckBoxField DataField="NameStyle"
                            HeaderText="NameStyle" SortExpression="NameStyle" />

                    </Columns>

                </asp:GridView>

            </div>

        </form>

    </body>

</html>
```

```

        <asp:BoundField DataField="Title" HeaderText="Title"
SortExpression="Title" />
        <asp:BoundField DataField="FirstName" HeaderText="FirstName"
SortExpression="FirstName" />
        <asp:BoundField DataField="MiddleName"
HeaderText="MiddleName" SortExpression="MiddleName" />
        <asp:BoundField DataField="LastName" HeaderText="LastName"
SortExpression="LastName" />
        <asp:BoundField DataField="Suffix" HeaderText="Suffix"
SortExpression="Suffix" />
        <asp:BoundField DataField="EmailAddress"
HeaderText="EmailAddress" SortExpression="EmailAddress" />
    </Columns>

</asp:GridView>

<br />

</div>

<asp:LINQDataSource ID="LINQDataSource1" runat="server"
    ContextTypeName="LINQWebApp1.AdventureWorksDataContext"
    EntityTypeName="" TableName="Contacts">
</asp:LINQDataSource>

</form>

</body>

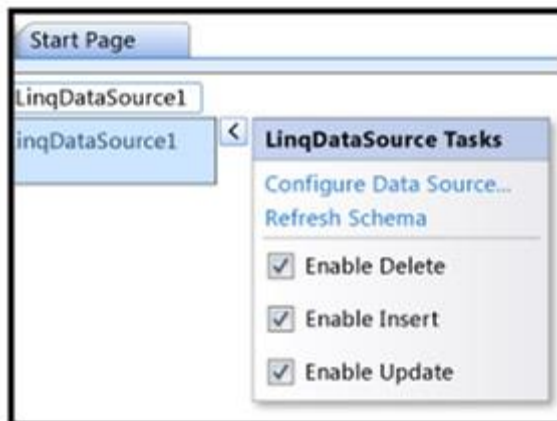
</html>

```

Here it should be noted that it is vital to set the property `ContextTypeName` to that of the class representing the database. For example, here it is given as `LINQWebApp1.AdventureWorksDataContext` as this action will make the needed connection between `LINQDataSource` and the database.

INSERT, UPDATE, and DELETE data in ASP.NET Page using LINQ

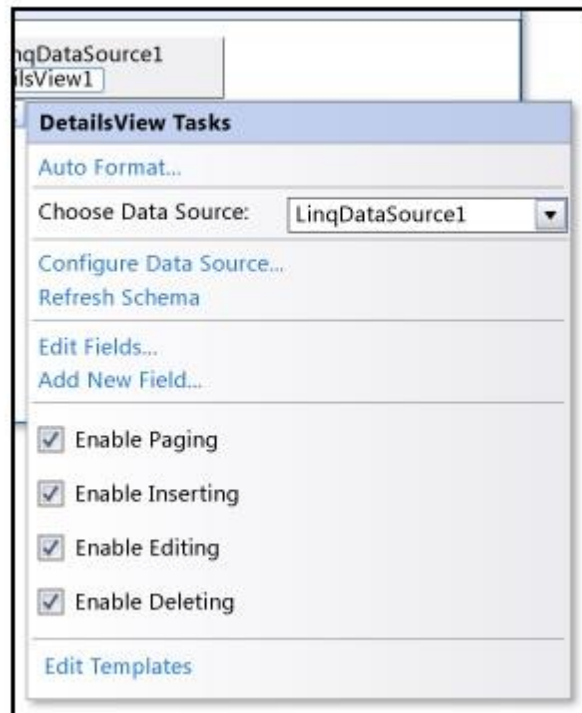
After completing all the above steps rigorously, choose the **LINQDataSource Tasks** from the **LINQDataSource Control** and choose all the three boxes for enable insert, enable update and enable delete from the same, as shown in the following screenshot.



Soon the declarative markup will get displayed on your screen as the following one.

```
<asp:LINQDataSource
    ContextTypeName="LINQWebApp1.AdventureWorksDataContext" TableName="Contacts"
    EnableUpdate="true"
    EnableInsert="true"
    EnableDelete="true"
    ID="LINQDataSource1"
    runat="server">
</asp:LINQDataSource>
```

Now since there are multiple rows and columns, it is better to add another control on your .aspx form named as Detail View or Master control below the Grid View control to display only the details of a selected row of the grid. Choose the Detail View Tasks from the Detail View control and select the check boxes as shown below.



Now, just save the changes and press Ctrl + F5 to view the page in your browser where it is now possible to delete, update, insert any record on the detail view control.