

Beginning LINQ

A new fashion to write Queries and address the impedance mismatch between programming languages and database.



Abhimanyu Kumar Vatsa

Microsoft MVP, ASP.NET/IIS

Blog: www.itorian.com

Twitter: <http://twitter.com/itorian>

Facebook: <https://www.facebook.com/2050.itorian>

This e-book is based on my series of LINQ posts and I'm just consolidating all blog posts here.

Contents

Part 1

In this part you will learn what is LINQ, its benefits and comparison with previous approach?

Part 2

In this part you will learn various approaches used in LINQ like LINQ to Array, XML and SQL.

Part 3

In this part you will learn something on 'Generic' like what are generic types, why we need it in LINQ?

Part 4

In this part you will learn how to setup a demo project to explore LINQ queries.

Part 5

In this part you will learn how to select records using LINQ Query and will explore its internals.

Part 6

In this part you will learn something like filtering, ordering, grouping and joining using LINQ.

Part 7

In this part you will learn how to use 'concat' key in LINQ query to do joins on multiple tables.

Part 8

In this part you will learn how to customize the LINQ's 'select' statement in various ways.

Part 9

In this part you will learn how to transform data source objects into XML.

Part 10

In this part you will learn how to perform some calculations inside LINQ Query.

Part 11

In this part you will look at some differences between LINQ Query Syntax and Method Syntax used in LINQ.

Part – 1

In this part you will learn what is LINQ, its benefits and comparison with previous approach?

What is LINQ?

In short, Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language (also in Visual Basic and potentially any other .NET language).

LINQ is a technique **used to retrieve data from any object** that implements the `IEnumerable<T>` interface. In LINQ, arrays, collections, relational data, and XML are all potential data sources and we can query over it to get only those data that meets our criteria and all is done using C# and VB. In Microsoft's PDC (Professional Developers Conference) 2005, Anders Hejlsberg and his team presented LINQ approach and its component released with .NET 3.5 Framework.

Benefits of LINQ

1. LINQ is integrated into C# and VB languages and it provides syntax highlighting and IntelliSense features and even **you can debug your queries using integrated debugger** in Visual Studio.
2. Using LINQ it is possible to **write codes much faster** than older queries and lets you save half of the query writing time.
3. Using LINQ **you can easily see the relationships between tables** and it helps you **compose your query that joins multiple tables**.
4. **Transformational** features of LINQ make it **easy to convert data of one type into a second type**. For example, you can easily transform SQL data into XML data using LINQ.

Previous Approach

To run a simple SQL query, ADO.NET programmers have to store the SQL in a Command object, associate the Command with a Connection object and execute it on that Connection object, then use a DataReader or other object to retrieve the result set. For example, the following code is necessary to retrieve the single row.

```
SqlConnection c = new SqlConnection(...); //DB Connection
c.Open(); //Open Connection
SqlCommand cmd = new SqlCommand(@"SELECT * FROM Employees e WHERE e.ID = @p0"); //SQL
Query
cmd.Parameters.AddWithValue("@p0", 1); //Add value to parameter
DataReader dr = c.Execute(cmd); //Execute the command
while (dr.Read()) {
    string name = dr.GetString(0);
} //Get name column value
```

```
// Update record using another Command object
.....
c.Close(); //Close connection
```

This is not only huge code, but there's also no way for the C# compiler to check our query against our use of the data it returns. When 'e' retrieve the employee's name we have to know the column's position in the database table to find it in the result. It's a common mistake to retrieve the wrong column and get a type exception or bad data at run time.

But, with LINQ you just need to perform three distinct actions:

1. Obtain the data source.
2. Create the query.
3. Execute the query.

And you all done.

Part – 2

In this part you will learn various approaches used in LINQ like LINQ to Array, XML and SQL.

As I said in Part 1, with LINQ you just need to perform three distinct actions: Obtain the Data Source, Create the query and Execute the query, you will notice it in sample programs given below. We will create Console Apps and test various LINQ concepts. I will walk through very simple programs here and in coming part will dig in depth.

LINQ has a great power of querying on any source of data that could be the collections of objects (in memory data, like array), SQL Database or XML files. We can easily retrieve data from any object that implements the `IEnumerable<T>` interface.

LINQ to Array

Let's look at a Program to find Even Number using LINQ to Array.

```
using System;
using System.Linq;

namespace ThreeActions
{
    class Program
    {
        static void Main(string[] args)
        {
            //Action 1. Data source
            int[] numbers = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            //Action 2. Query creation
            var queryNumber =
                from num in numbers
                where (num % 2) == 0
                select num;

            //Action 3. Query execution
            foreach (int num in queryNumber)
            {
                Console.WriteLine("{0,1} ", num);
            }

            Console.ReadKey();
        }
    }
}

// Output
// 2 4 6 8 10
```

Data Source

Query

Executing Query

If you execute above program, you will get '2 4 6 8 10' as output. You will notice 3 actions above, these actions will be changed always depending upon our project requirement. In above example, 'data source' was an array that implicitly supports the generic `IEnumerable<T>` interface.

LINQ to XML

Now, let's move on to take a look at LINQ to XML example and find student names from XML file. LINQ to XML loads an XML document into a query-able XElement type and then IEnumerable<XElement> loads the query result and using foreach loop we access it.

Student.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<students>
  <student id="1">
    <name>Abhimanyu K Vatsa</name>
    <address>Bokaro Steel City</address>
  </student>
  <student id="2">
    <name>Deepak Kumar</name>
    <address>Dhanbad</address>
  </student>
  <student id="3">
    <name>Rohit Ranjan</name>
    <address>Chas</address>
  </student>
  <student id="4">
    <name>Rahul Kumar</name>
    <address>Muzaffarpur</address>
  </student>
</students>
```

Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Xml.Linq;
```

```
namespace ThreeActions
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            //Action 1. Data Source
```

```
            XElement xmlFile = XElement.Load(@"c:\users\abhimanyukumar\student.xml");
```

```
            //Action 2. Query creation
```

```
            IEnumerable<XElement> nqs = from c in xmlFile.Elements("student").Elements("name")
                                         //where (string)c.Attribute("id") == "1"
                                         select c;
```

```
            //Action 3. Query execution
```

```
            foreach (string data in nqs)
```

```
            {
```

```
                Console.WriteLine("{0} ", data);
```

```
            }
```

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

```
        }
```

```
    }
```

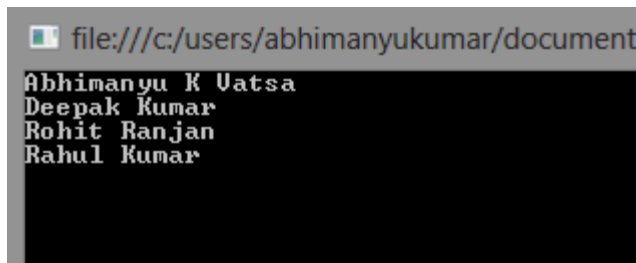
```
}
```

Data Source

Query

Executing Query

Output



```
file:///c:/users/abhimanyukumar/document
Abhimanyu K Uatsa
Deepak Kumar
Rohit Ranjan
Rahul Kumar
```

LINQ to SQL

In LINQ to SQL, firstly we create an object-relational mapping at design time either manually or by using the Object Relational Designer. Then, we write queries against the objects, and at run-time LINQ to SQL handles the communication with the database.

Now, let's look at the sample program:

```
using System;
using System.Linq;

namespace ThreeActions
{
    class Program
    {
        static void Main(string[] args)
        {
            //Action 1. Data Source
            StudentDataContext std = new StudentDataContext();

            //Action 2. Query creation
            var stdNames = from student in std.StudentTables
                           select student.Name;

            //Action 3. Query execution
            foreach (string n in stdNames)
            {
                Console.WriteLine("{0}", n);
            }

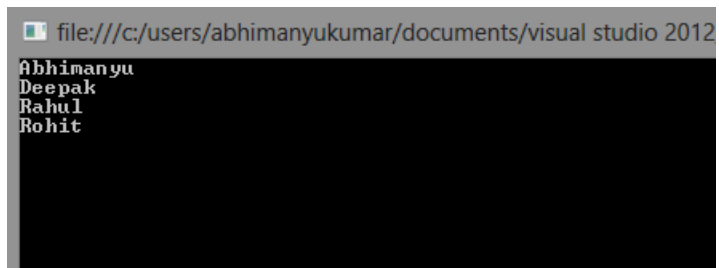
            Console.ReadKey();
        }
    }
}
```

Data Source

Query

Executing Query

Output



```
file:///c:/users/abhimanyukumar/documents/visual studio 2012
Abhimanyu
Deepak
Rahul
Rohit
```

Very simple examples on LINQ so far, find more in next part.

Part – 3

In this part you will learn something on 'Generic' like what are generic types, why we need it in LINQ?

First of all let's understand what are 'Arrays', '.NET Collections' and '.NET Generic Collections'. You will see following advantages and disadvantages in examples given below.

Collection Types	Advantages	Disadvantages
Arrays	Strongly Typed (Type Safe)	<ul style="list-style-type: none">• Zero Index Based• Cannot grow the size once initialized
.NET Collections (like ArrayList, Hashtable etc)	<ul style="list-style-type: none">• Can grow in size• Convenient to work with Add, Remove	Not Strongly Typed
.NET Generic Collections (like List<T>, GenericList<T> etc)	<ul style="list-style-type: none">• Type Safe like arrays• Can grow automatically like ArrayList• Convenient to work with Add, Remove	

Now, let's talk on each of the collection type given in above table to feel real pain of developments.

Array

```
using System;

namespace GenericDemo1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numbers = new int[4];

            numbers[0] = 1001;
            numbers[1] = 1002;
            numbers[2] = 1003;
            numbers[3] = 1004;
            //numbers[4] = 1005;           //Run Time Error: Index was outside the bounds of the array.
            //numbers[4] = "Abhimanyu"; //Compile Time Error: Cannot implicitly convert type 'string' to 'int'

            foreach (int n in numbers)
            {
                Console.WriteLine(n);
            }

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

If you use array:

1. You can't enter more values beyond the initialized size otherwise you will see a runtime error saying 'index was outside the bounds of the array'.
2. If you declare array as an int type then, you are limited to enter integer values only otherwise you will see a compile time error saying 'cannot implicitly convert type string to int', all because array is type safe.
3. Array is index based, so you always need to write index number to insert value in collection.
4. You don't have to add or remove methods here.

ArrayList (System.Collection)

```
using System;
using System.Collections;

namespace GenericDemo2
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList numbers = new ArrayList(4);

            numbers.Add(1001);
            numbers.Add(1002);
            numbers.Add(1003);
            numbers.Add(1004);
            numbers.Add(1005);           //No Error: Because ArrayList is auto grow
            //numbers.Add("Abhimanyu");   //Run Time Error: Specified cast is not valid.
            numbers.Remove(1005);

            foreach (int n in numbers)
            {
                Console.WriteLine(n);
            }

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

If you use ArrayList:

- You will get auto grow feature, means initial size will not limit you.
- You don't option to declare the data type at declaration time on the top, so you can add any value like string, Boolean, int etc. Remember to care it in foreach loop at runtime.
- You will have some very useful methods like Remove, RemoveAt, RemoveRange etc. You don't have any option like index.

List<T> (System.Collections.Generic)

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

namespace GenericDemo3
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> numbers = new List<int>(4);

            numbers.Add(1001);
            numbers.Add(1002);
            numbers.Add(1003);
            numbers.Add(1004);
            numbers.Add(1005);

            foreach (int n in numbers)
            {
                Console.WriteLine(n);
            }

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

Now, in short we can say, list has both features that we have in array or arraylist.

Why we need List (which is generic type) in LINQ?

LINQ queries are based on generic types, which were introduced in version 2.0 of the .NET Framework. LINQ queries returns collection and we don't even know what type of data we will get from database and that is the reason we use generic type like List<T>, IEnumerable<T>, IQueryable<T> etc. For example:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Xml.Linq;

namespace ThreeActions
{
    class Program
    {
        static void Main(string[] args)
        {
            //Action 1. Data Source
            XElement xmlFile = XElement.Load(@"c:\users\abhimanyukumar\student.xml");

            //Action 2. Query creation
            IEnumerable<XElement> nqs = from c in xmlFile.Elements("student").Elements("name")
                                     //where (string)c.Attribute("id") == "1"
                                     select c;

            //Action 3. Query execution
            foreach (string data in nqs)
            {
                Console.WriteLine("{0} ", data);
            }

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

Data Source

Query

Executing Query

In above example, we have used IEnumerable<T> interface and LINQ query and then by using foreach loop I'm able to get my data.

One more example:

```
IEnumerable<Customer> customerQuery =  
    from cust in customers  
    where cust.City == "Bokaro"  
    select cust;  
  
foreach (Customer customer in customerQuery)  
{  
    Console.WriteLine(customer.LastName + " = " + customer.Age);  
}
```

In above example you can see, using LINQ query I'm not only limited to select the string value that is LastName, I'm also able to select integer value that is Age.

So, this could be the great example of generic classes.

Here is some counterpart of Generic Collections over non-generic collections:

Collections (System.Collection)	Generic Collection (System.Collection.Generic)
ArrayList	List<T>
Hashtable	Dictionary<Tkey, Tvalue>
Stack	Stack<T>
Queue	Queue<T>

Now, go ahead and learn rest in next part.

Part – 4

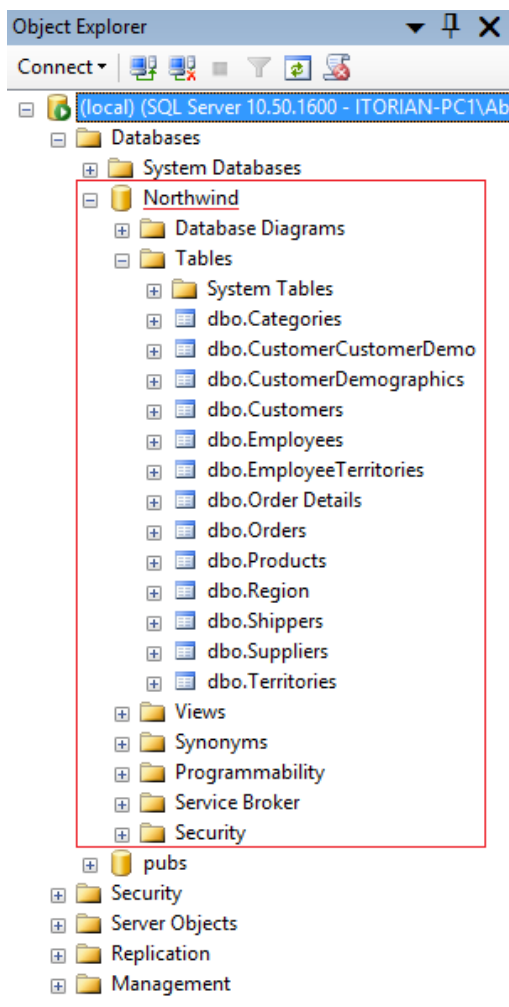
In this part you will learn how to setup a demo project to explore LINQ queries.

First of all, I'll recommend you to watch this video post on YouTube.

http://www.youtube.com/watch?v=yQB4HGmuwY8&feature=player_embedded

I will use my preinstalled 'Northwind' database that you can download from [here](#) and install it in SQL Server Management Studio.

Now, follow the steps to setup a demo project to explore LINQ queries. Before creating new project, setup a 'Northwind' database in SQL Server Management Studio.

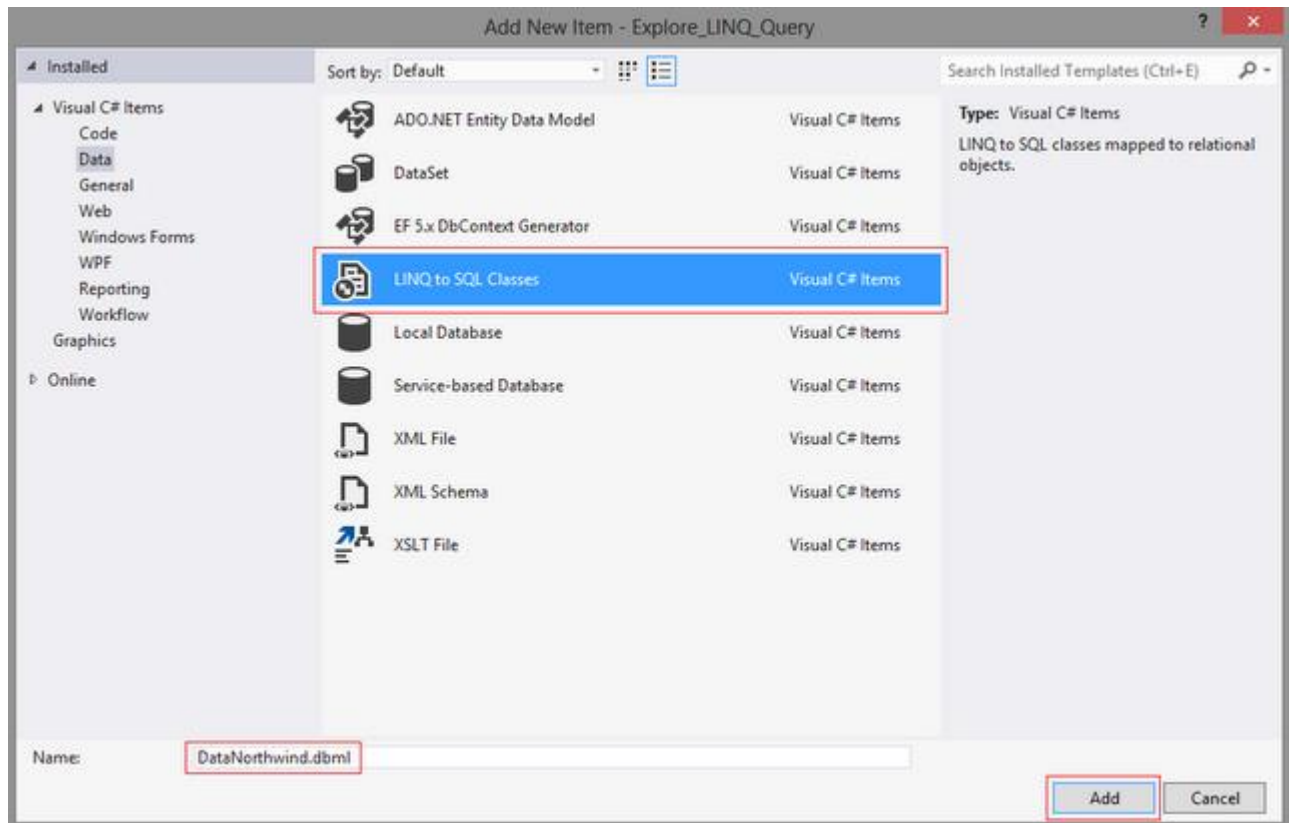


Step 1: Creating New Project

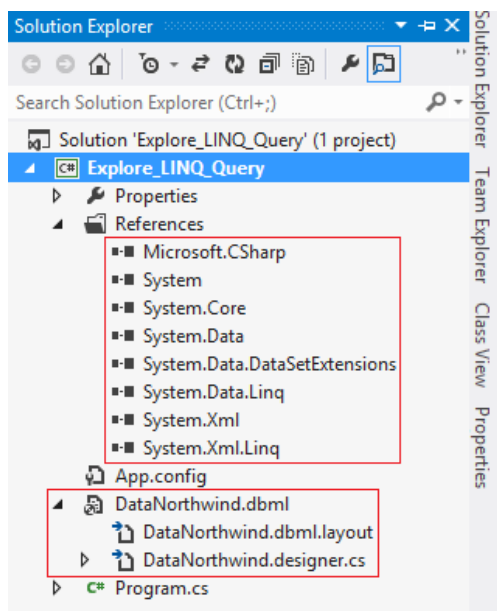
Open the Visual Studio instance and create a new project New > Project > Console Application and name the project whatever you wish.

Step 2: Adding LINQ to SQL Classes

Now, in the solution explorer add a new item 'LINQ to SQL Classes' by a nice name 'DataNorthwind.dbml'.

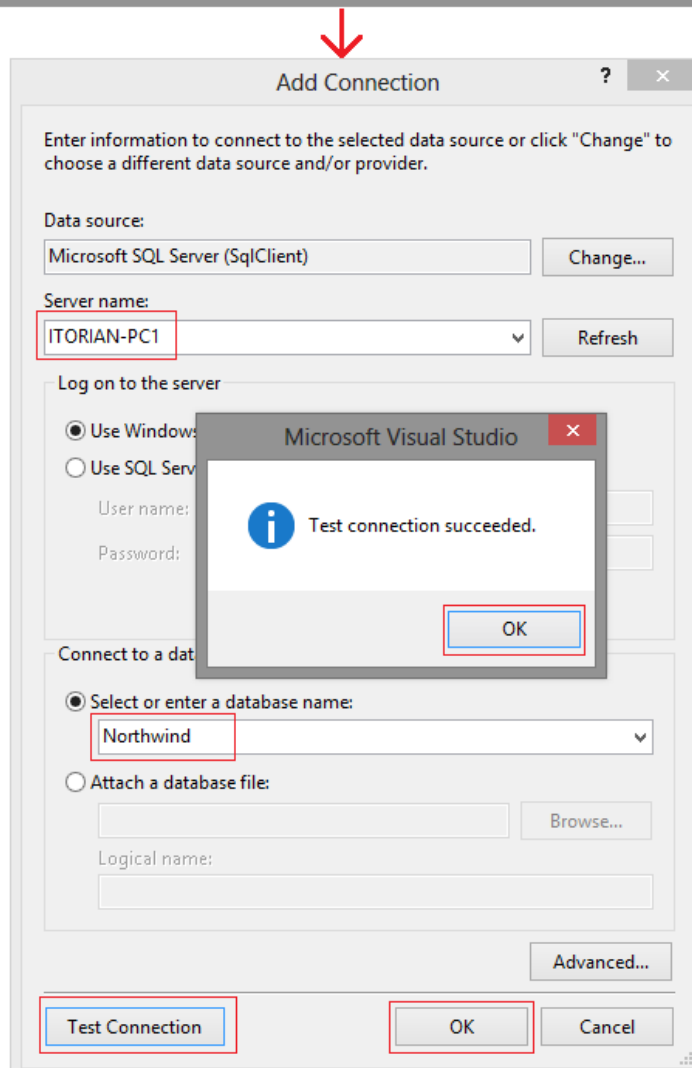
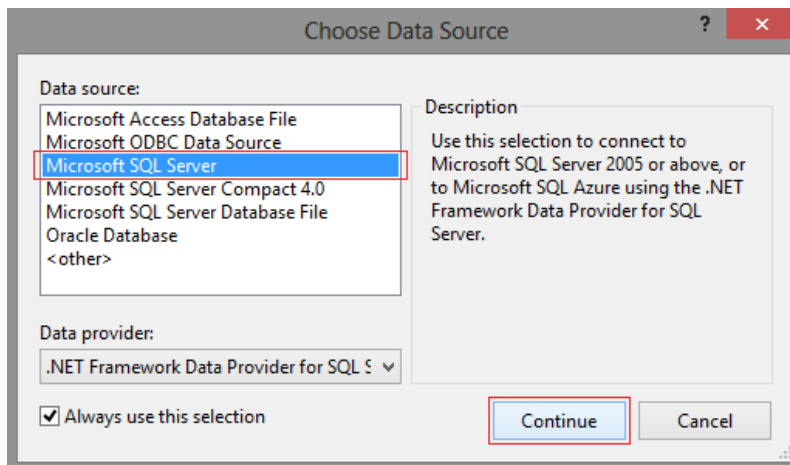


This will add all required library as well as classes in your project for database communication. Actually, this dbml file will become a layer between DB and your project and provide you all the IntelliSense features.

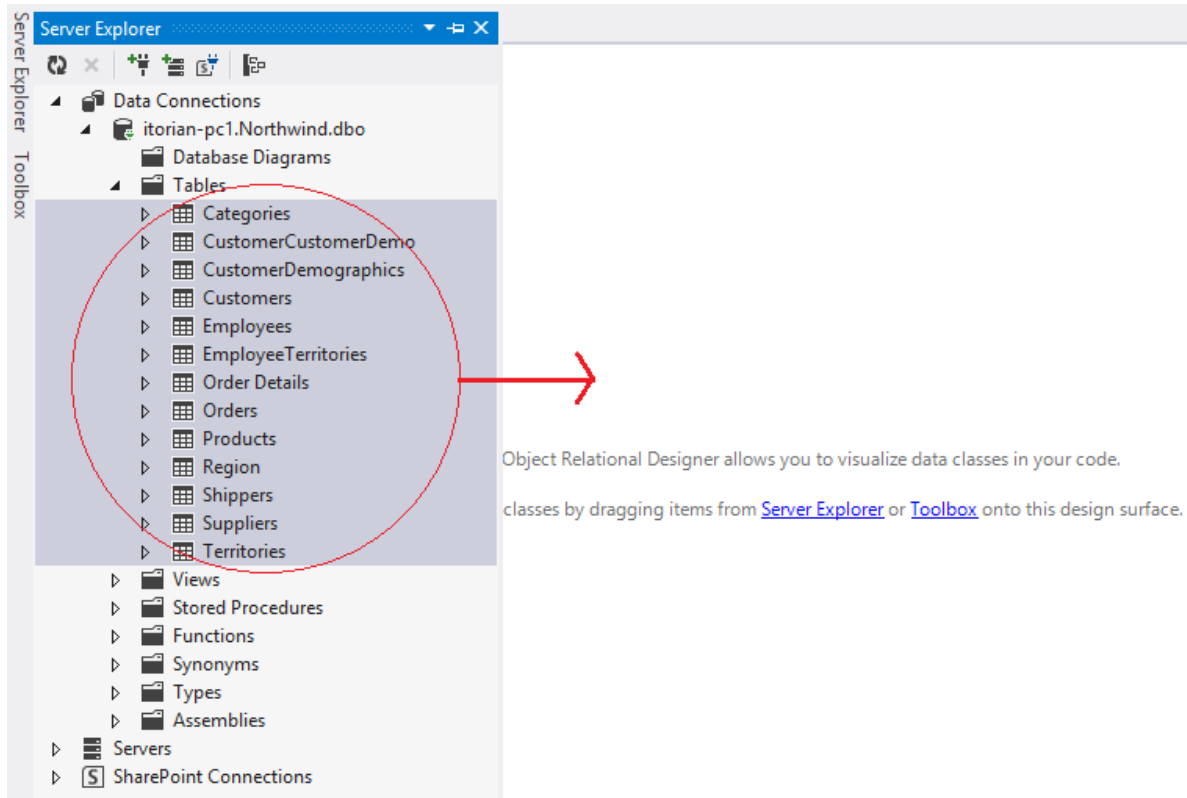


Step 3: Adding New Connection

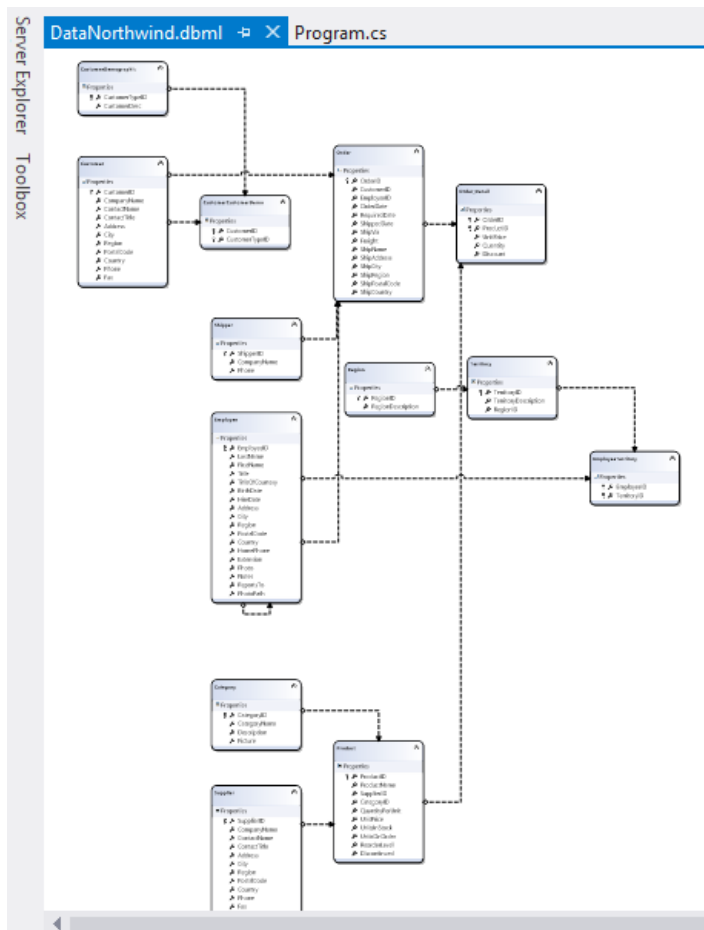
Open 'Server Explorer' and add a new SQL Server connection, follow the image.



Once you done with above, now drag the tables from 'Server Explorer' to 'DataNorthwind.dbml' file designer.



Now, you will get following DB diagram on designer.

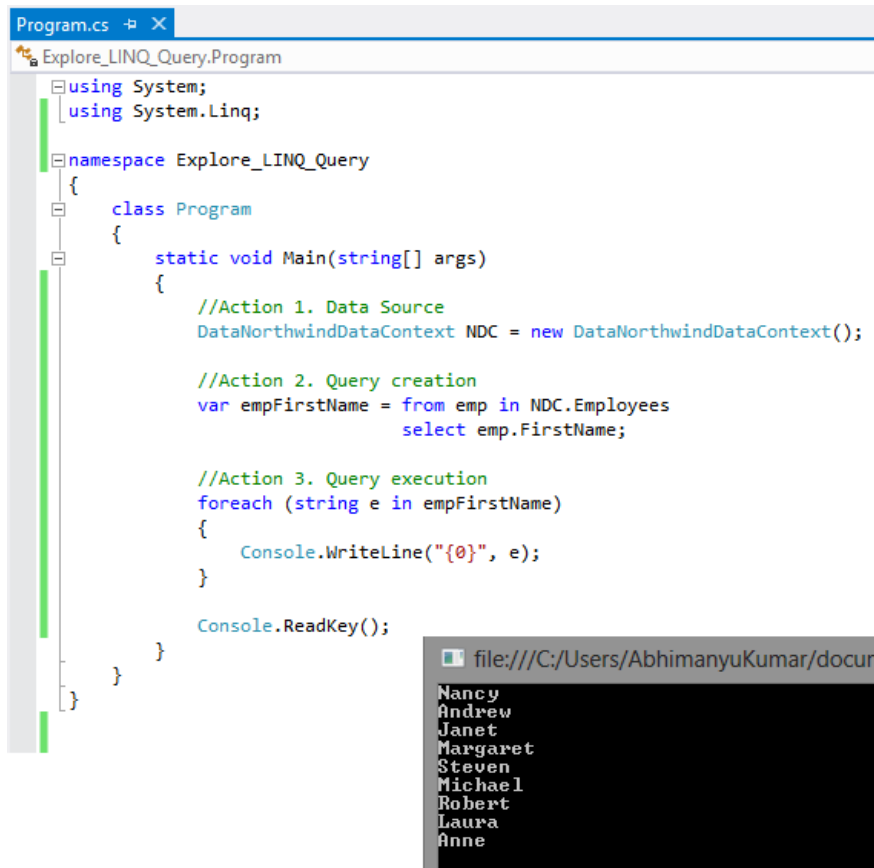


You will notice following things here:

1. Got a new connection string in App.config file.
2. Got every setup required to communicate with DB like DataNorthwindDataContext, TableAttribute, EntitySet etc in DataNorthwind.designer.cs file.

Step 4: Coding and Running Project

Now, setup the project code as given blow and run it.



The screenshot shows a Visual Studio IDE with a C# program named 'Program.cs' in a project called 'Explore_LINQ_Query'. The code defines a 'Program' class with a 'Main' method. The 'Main' method performs three actions: 1. Data Source setup, 2. Query creation, and 3. Query execution. The query selects the first names of employees from the 'Employees' table. The console output shows the first names of the employees: Nancy, Andrew, Janet, Margaret, Steven, Michael, Robert, Laura, and Anne.

```
Program.cs
Explore_LINQ_Query.Program
using System;
using System.Linq;

namespace Explore_LINQ_Query
{
    class Program
    {
        static void Main(string[] args)
        {
            //Action 1. Data Source
            DataNorthwindDataContext NDC = new DataNorthwindDataContext();

            //Action 2. Query creation
            var empFirstName = from emp in NDC.Employees
                              select emp.FirstName;

            //Action 3. Query execution
            foreach (string e in empFirstName)
            {
                Console.WriteLine("{0}", e);
            }

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

file:///C:/Users/AbhimanyuKumar/docum
Nancy
Andrew
Janet
Margaret
Steven
Michael
Robert
Laura
Anne

We have already seen this code in one of the previous post. Now, we have a demo project setup completely and ready to try all the LINQ queries.

In upcoming part, we will look at various LINQ queries and will test it in above project.

Part – 5

In this part you will learn how to select records using LINQ Query and will explore its internals.

Selecting Records

Selecting data using LINQ queries is very simple, you will get full intellisense support while querying database and even compile time error check that adds great advantage to LINQ, once you learn it, you will love to use this in your every DB projects. I love this, it's great.

In the image given below, you will find the sample code for selecting employees 'FirstName' and 'LastName' from 'Employee' table of 'Northwind' Database:

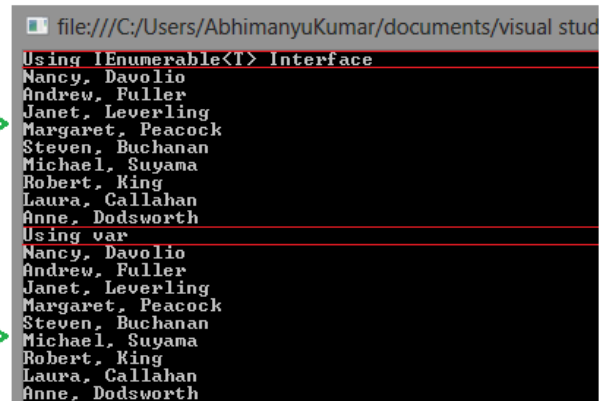
```
DataNorthwindDataContext NDC = new DataNorthwindDataContext();
```

```
Console.WriteLine("Using IEnumerable<T> Interface");
```

```
IEnumerable<Employee> empQuery1 = from emp in NDC.Employees  
    select emp;  
  
foreach (Employee e in empQuery1)  
{  
    Console.WriteLine(e.FirstName + ", " + e.LastName);  
}
```

```
Console.WriteLine("Using var");
```

```
var empQuery2 = from emp in NDC.Employees  
    select emp;  
  
foreach (var e in empQuery2)  
{  
    Console.WriteLine(e.FirstName + ", " + e.LastName);  
}
```



```
file:///C:/Users/AbhimanyuKumar/documents/visual stud  
Using IEnumerable<T> Interface  
Nancy, Davolio  
Andrew, Fuller  
Janet, Leverling  
Margaret, Peacock  
Steven, Buchanan  
Michael, Suyama  
Robert, King  
Laura, Callahan  
Anne, Dodsworth  
Using var  
Nancy, Davolio  
Andrew, Fuller  
Janet, Leverling  
Margaret, Peacock  
Steven, Buchanan  
Michael, Suyama  
Robert, King  
Laura, Callahan  
Anne, Dodsworth
```

In above image, you can see, I'm using same queries two times, but there is difference, don't go on output screen only. If you look on the code carefully, you will find I'm using `IEnumerable<Employee>` in the first block of code and just a `'var'` in second block of code. Also note, for `IEnumerable<Employee>` I'm using `'Employee'` in foreach loop and for `'var'` I'm using `'var'` in foreach loop, you should care it always.

Now, what is the difference between 'IEnumerable<Employee>' and 'var'?

IEnumerable<Employee>

When you see a query variable that is typed as `IEnumerable<T>`, it just means that the query, when it is executed, will produce a sequence of zero or more `T` objects, as given in image blow (just hover the mouse on `'empQuery1'` when compiler is on break). When you expand the `index[]`, you will find entire data associated with that row/record.

```

IEnumerable<Employee> empQuery1 = from emp in NDC.Emp1
    select emp;
foreach (Employee emp in empQuery1)
{
    Console.WriteLine(emp.LastName);
}

```

Results View: Expanding the Results View

- [0] {Explore_LINQ_Query.Employee}
- [1] {Explore_LINQ_Query.Employee}
- [2] {Explore_LINQ_Query.Employee}
- [3] {Explore_LINQ_Query.Employee}
- [4] {Explore_LINQ_Query.Employee}
- [5] {Explore_LINQ_Query.Employee}
- [6] {Explore_LINQ_Query.Employee}
- [7] {Explore_LINQ_Query.Employee}
- [8] {Explore_LINQ_Query.Employee}

Employee Details:

_Address	Q - "507 - 20th Ave. E.\r\nApt. 2A"
_BirthDate	Q - {12/8/1948 12:00:00 AM}
_City	Q - "Seattle"
_Country	Q - "USA"
_Employee1	{System.Data.Linq.EntityRef<Explore_LINQ_Query.Employee>}
_EmployeeID	1
_Employees	{System.Data.Linq.EntitySet<Explore_LINQ_Query.Employee>}
_EmployeeTerritories	{System.Data.Linq.EntitySet<Explore_LINQ_Query.EmployeeTerritory>}
_Extension	Q - "5467"
_FirstName	Q - "Nancy"
_HireDate	Q - {5/1/1992 12:00:00 AM}
_HomePhone	Q - "(206) 555-9857"
_LastName	Q - "Davolio"
_Notes	Q - "Education includes a BA in psychology from Colorado State University in 1970."
_Orders	{System.Data.Linq.EntitySet<Explore_LINQ_Query.Order>}

var

If you prefer, you can avoid generic syntax like `IEnumerable<T>` by using the 'var' keyword. The 'var' keyword instructs the compiler to infer the type of a query variable by looking at the data source specified in from clause, no big difference you still get all benefits.

Now, let's talk on the query part.

```

var empQuery1 = from emp in NDC.Employees
    select emp;

```

In C# as in most programming languages a variable must be declared before it can be used. In a LINQ query, the 'from' clause comes first in order to introduce the data source (`NDC.Employees`) and the range variable (`emp`). `NDC` is just an instance of the 'Northwind Data Context' that we have created in Part 4 and inside this Data Context, we got `Employees` table and that is why I have used '`NDC.Employees`'.

Executing Query (using foreach loop)

Once you done with query your next step will be executing it. Let's go ahead and understand it too.

```
foreach (Employee e in empQuery1)
{
    Console.WriteLine(e.FirstName + ", " + e.LastName);
}



---



foreach (var e in empQuery2)
{
    Console.WriteLine(e.FirstName + ", " + e.LastName);
}
```

As we have bunch of rows/columns in 'empQuery1' or 'empQuery2' returned by the LINQ select statement, we need to specify which column we want to see in output and that's why, we have used e.FirstName and e.LastName, where 'e' is instance of 'empQuery1' or 'empQuery2'.

Alternatively, you can bind 'empQuery1' or 'empQuery2' directly to any Data Control like GridView.

In upcoming part, you will learn how to filter, order, group, and join using LINQ.

Part – 6

In this part you will learn something like filtering, ordering, grouping and joining using LINQ.

Filter

Filter is the most common query operation to apply a filter in the form of a Boolean expression. The filter causes the query to return only those elements for which the expression is true. The result is produced by using the where clause. The filter in effect specifies which elements to exclude from the source sequence. Let's look at sample query:

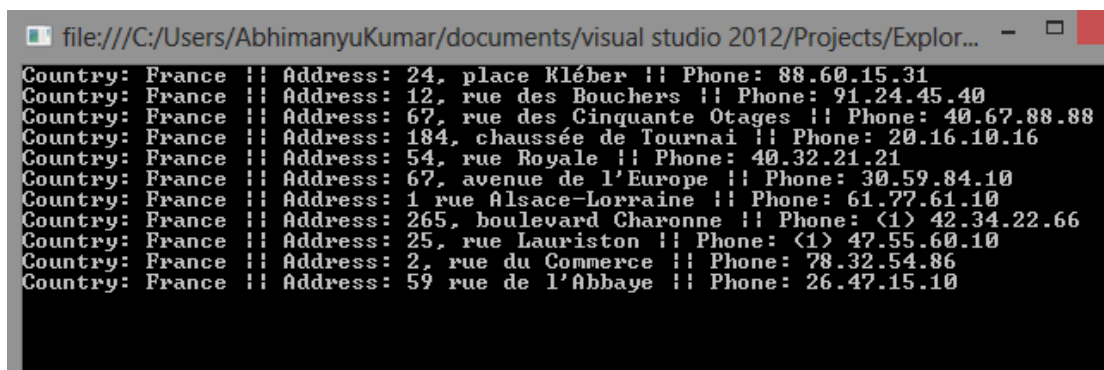
```
DataNorthwindDataContext NDC = new DataNorthwindDataContext();

var custQuery = from cust in NDC.Customers
                where cust.Country == "France"
                select cust;

foreach (var e in custQuery)
{
    Console.WriteLine("Country: " + e.Country + " || Address: " + e.Address + " || Phone: " + e.Phone);
}

Console.ReadKey();
```

In above query, I'm asking for only those records who's 'Country' is 'France'. And in the foreach loop, 'Country', 'Address' and 'Phone' separated by '||' and the same in output.



```
file:///C:/Users/AbhimanyuKumar/documents/visual studio 2012/Projects/Explor...
Country: France || Address: 24, place Kléber || Phone: 88.60.15.31
Country: France || Address: 12, rue des Bouchers || Phone: 91.24.45.40
Country: France || Address: 67, rue des Cinquante Otages || Phone: 40.67.88.88
Country: France || Address: 184, chaussée de Tournai || Phone: 20.16.10.16
Country: France || Address: 54, rue Royale || Phone: 40.32.21.21
Country: France || Address: 67, avenue de l'Europe || Phone: 30.59.84.10
Country: France || Address: 1 rue Alsace-Lorraine || Phone: 61.77.61.10
Country: France || Address: 265, boulevard Charonne || Phone: <1> 42.34.22.66
Country: France || Address: 25, rue Lauriston || Phone: <1> 47.55.60.10
Country: France || Address: 2, rue du Commerce || Phone: 78.32.54.86
Country: France || Address: 59 rue de l'Abbaye || Phone: 26.47.15.10
```

In the same way, if you want to select records where 'Country' is 'France' and 'ContactName' starts with 'A', then use

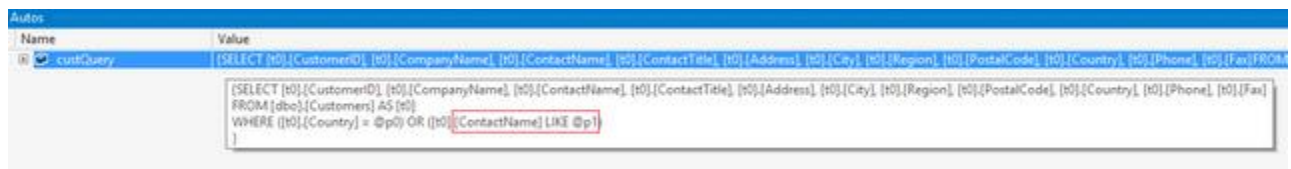
```
var custQuery = from cust in NDC.Customers
                where cust.Country == "France" && cust.ContactName.StartsWith("a")
                select cust;
```

And, if you want to select records where 'Country' is 'France' or 'ContactName' starts with 'A', then use

```
var custQuery = from cust in NDC.Customers
    where cust.Country == "France" || cust.ContactName.StartsWith("a")
    select cust;
```

So, in both query, '&&' is being used for 'And' and '||' is being used for 'Or'.

Now, 'StartsWith' is a LINQ level key that is equivalent to LIKE operator in SQL. You can see it in generated query here.



We will look more only such available 'keys' in coming post.

Order

The orderby clause will cause the elements in the returned sequence to be sorted according to the default comparer for the type being sorted. For example the following query can be extended to sort the results based on the ContactName property. Because ContactName is a string, the default comparer performs an alphabetical sort from A to Z.

```
var custQuery = from cust in NDC.Customers
    orderby cust.ContactName descending //orderby cust.ContactName ascending
    select cust;
```

Group

The group clause enables you to group your results based on a key that you specify. For example you could specify that the results should be grouped by the City.

```
var custQuery = from cust in NDC.Customers
    where cust.ContactName.StartsWith("a")
    group cust by cust.City;
```

When you end a query with a group clause, your results take the form of a list of lists. Each element in the list is an object that has a key member and a list of elements that are grouped under that key. When you iterate over a query that produces a sequence of groups, you must use a nested foreach loop. The outer loop iterates over each group, and the inner loop iterates over each group's members.


```

foreach (var e in custQuery)
{
    int x = e.Key.Length;
    Console.WriteLine("\n");
    Console.WriteLine(e.Key);
    Console.WriteLine(Repeat('-', x));

    foreach (Customer c in e)
    {
        Console.WriteLine("Contact Name : " + c.ContactName);
    }
}

```

And the output will be organized as



```

file:///C:/Users/AbhimanyuKumar/documents/visual studio 2012/Projects/Explor

Campinas
-----
Contact Name : André Fonseca

Lander
-----
Contact Name : Art Braunschweiger

Leipzig
-----
Contact Name : Alexander Feuer

London
-----
Contact Name : Ann Devon

Madrid
-----
Contact Name : Alejandra Camino

México D.F.
-----
Contact Name : Ana Trujillo
Contact Name : Antonio Moreno

Sao Paulo
-----
Contact Name : Aria Cruz
Contact Name : Anabela Domingues

Toulouse
-----
Contact Name : Annette Roulet

```

If you must refer to the results of a group operation, you can use the 'into' keyword to create an identifier that can be queried further. The following query returns only those groups that contain more than two customers:

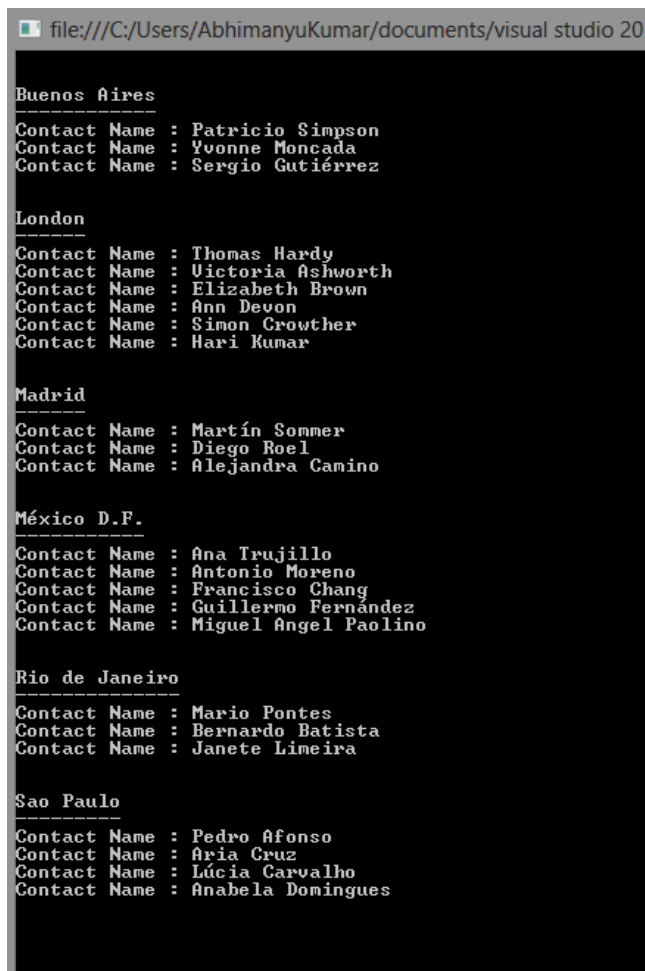

```
var custQuery = from cust in NDC.Customers
                group cust by cust.City into custGroup
                where custGroup.Count() > 2
                select custGroup;
```

And the foreach loop will be same.

```
foreach (var e in custQuery)
{
    int x = e.Key.Length;
    Console.WriteLine("\n");
    Console.WriteLine(e.Key);
    Console.WriteLine(Repeat('-', x));

    foreach (Customer c in e)
    {
        Console.WriteLine("Contact Name : " + c.ContactName);
    }
}
```

You will get following output:



```
file:///C:/Users/AbhimanyuKumar/documents/visual studio 2017/...
Buenos Aires
-----
Contact Name : Patricio Simpson
Contact Name : Yvonne Moncada
Contact Name : Sergio Gutiérrez

London
-----
Contact Name : Thomas Hardy
Contact Name : Victoria Ashworth
Contact Name : Elizabeth Brown
Contact Name : Ann Devon
Contact Name : Simon Crowther
Contact Name : Hari Kumar

Madrid
-----
Contact Name : Martín Sommer
Contact Name : Diego Roel
Contact Name : Alejandra Canino

México D.F.
-----
Contact Name : Ana Trujillo
Contact Name : Antonio Moreno
Contact Name : Francisco Chang
Contact Name : Guillermo Fernández
Contact Name : Miguel Angel Paolino

Rio de Janeiro
-----
Contact Name : Mario Pontes
Contact Name : Bernardo Batista
Contact Name : Janete Limeira

Sao Paulo
-----
Contact Name : Pedro Afonso
Contact Name : Aíria Cruz
Contact Name : Lúcia Carvalho
Contact Name : Anabela Domingues
```

Join

Join operations create associations between sequences that are not explicitly modeled in the data sources. For example you can perform a join to find all the customers and distributors who have the same location. In LINQ the join clause always works against object collections instead of database tables directly.

Question: What query we should write to select names from two different tables 'Customer' and 'Employee' depending upon matching city.

Answer: Query will be:

```
var custQuery = from cust in NDC.Customers
                join emp in NDC.Employees on cust.City equals emp.City
                select new { CityName = cust.City, CustomerName = cust.ContactName, EmployeeName =
emp.FirstName };
```

And in foreach loop, will write

```
foreach (var e in custQuery)
{
    Console.WriteLine(e.CityName + " : " + e.CustomerName + ", " + e.EmployeeName);
}
```

Output:



```
file:///C:/Users/AbhimanyuKumar/documents/
Seattle : Karl Jablonski, Nancy
Kirkland : Helvetius Nagy, Janet
London : Thomas Hardy, Steven
London : Victoria Ashworth, Steven
London : Elizabeth Brown, Steven
London : Ann Devon, Steven
London : Simon Crowther, Steven
London : Hari Kumar, Steven
London : Thomas Hardy, Michael
London : Victoria Ashworth, Michael
London : Elizabeth Brown, Michael
London : Ann Devon, Michael
London : Simon Crowther, Michael
London : Hari Kumar, Michael
London : Thomas Hardy, Robert
London : Victoria Ashworth, Robert
London : Elizabeth Brown, Robert
London : Ann Devon, Robert
London : Simon Crowther, Robert
London : Hari Kumar, Robert
Seattle : Karl Jablonski, Laura
London : Thomas Hardy, Anne
London : Victoria Ashworth, Anne
London : Elizabeth Brown, Anne
London : Ann Devon, Anne
London : Simon Crowther, Anne
London : Hari Kumar, Anne
```

We can use 'Group by' here to group the output.

Part – 7

In this part you will learn how to use 'concat' key in LINQ query to do joins on multiple tables.

Assume if we have following data source information:

Student.cs

```
class Student
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public List<int> Marks { get; set; }
}
```

Teacher.cs

```
class Teacher
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
}
```

Program.cs

//DataSource1

```
List<Student> students = new List<Student>()
{
    new Student { ID=1, Name="Abhimanyu K Vatsa", Address="Bokaro", Marks= new List<int> {97, 92, 81, 90}},
    new Student { ID=2, Name="Deepak Kumar", Address="Dhanbad", Marks= new List<int> {70, 56, 87, 69}},
    new Student { ID=3, Name="Mohit Kumar", Address="Dhanbad", Marks= new List<int> {78, 76, 81, 56}},
    new Student { ID=4, Name="Geeta K", Address="Bokaro", Marks= new List<int> {95, 81, 54, 67}}
};
```

//DataSource2

```
List<Teacher> teachers = new List<Teacher>()
{
    new Teacher {ID=1, Name="Ranjeet Kumar", Address = "Bokaro"},
    new Teacher {ID=2, Name="Gopal Chandra", Address = "Dhanbad"}
};
```

And I want to select those Students' and Teachers' name who are from 'Bokaro' by concat. So, we can use LINQ query to create an output sequence that contains elements from more than one input sequence. Here it is:

//Query using Concat

```
var query = (from student in students
              where student.Address == "Bokaro"
              select student.Name)
.Concat(from teacher in teachers
        where teacher.Address == "Bokaro"
        select teacher.Name);
```

And output is:

Abhimanyu K Vatsa
Geeta K
Ranjeet Kumar

Remember, this can be done using join also that we have already seen in one of the previous post.

Part – 8

In this part you will learn how to customize the LINQ's 'select' statement in various ways.

Assume following as our data source:

Student.cs

```
class Student
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public List<int> Marks { get; set; }
}
```

Program.cs

```
List<Student> students = new List<Student>()
{
    new Student { ID=1, Name="Abhimanyu K Vatsa", Address="Bokaro", Marks= new List<int> {97, 92, 81, 90}},
    new Student { ID=2, Name="Deepak Kumar", Address="Dhanbad", Marks= new List<int> {70, 56, 87, 69}},
    new Student { ID=3, Name="Mohit Kumar", Address="Dhanbad", Marks= new List<int> {78, 76, 81, 56}},
    new Student { ID=4, Name="Geeta K", Address="Bokaro", Marks= new List<int> {95, 81, 54, 67}}
};
```

Study 1

Now, if you want to select only name of students' then use following LINQ query:

```
var query = from student in students
            select student.Name;
```

```
foreach (var e in query)
{
    Console.WriteLine(e);
}
```

Output:

```
Abhimanyu K Vatsa
Deepak Kumar
Mohit Kumar
Geeta K
```

Study 2

Now, if you want to select name and address of the student then use following LINQ query:

```
var query = from student in students
            select new { Name = student.Name, Address = student.Address};

foreach (var e in query)
{
    Console.WriteLine(e.Name + " from " + e.Address);
}
```

Output:

```
Abhimanyu K Vatsa from Bokaro
Deepak Kumar from Dhanbad
Mohit Kumar from Dhanbad
Geeta K from Bokaro
```

Study 3

Now, if you want to select name, marks and even want to add it then use following query:

```
var query = from student in students
            select new { Name = student.Name, Mark = student.Marks};

int sum = 0;
foreach (var e in query)
{
    Console.Write(e.Name + " : ");

    foreach (var m in e.Mark)
    {
        sum = sum + m;
        Console.Write(m + " ");
    }

    Console.WriteLine(": Total- " + sum);

    Console.WriteLine();
}
```

Output:

```
Abhimanyu K Vatsa : 97 92 81 90 : Total- 360
Deepak Kumar : 70 56 87 69 : Total- 642
Mohit Kumar : 78 76 81 56 : Total- 933
Geeta K : 95 81 54 67 : Total- 1230
```

In above query, we have Mark field that will produce a list and that's why we need to use nested foreach loop to get the list item and at the same time we are adding mark items.

Part – 9

In this part you will learn how to transform data source objects into XML.

In-Memory Data-Source

Let's have some in-memory data:

```
List<Student> students = new List<Student>()
{
    new Student { ID=1, Name="Abhimanyu K Vatsa", Address="Bokaro", Marks= new List<int> {97, 92, 81, 90}},
    new Student { ID=2, Name="Deepak Kumar", Address="Dhanbad", Marks= new List<int> {70, 56, 87, 69}},
    new Student { ID=3, Name="Mohit Kumar", Address="Dhanbad", Marks= new List<int> {78, 76, 81, 56}},
    new Student { ID=4, Name="Geeta K", Address="Bokaro", Marks= new List<int> {95, 81, 54, 67}}
};
```

Once we have in-memory data, we can create query on it to generate xml tags and its data as given below:

```
var studentsToXML = new XElement("School",
    from student in students
    let x = String.Format("{0},{1},{2},{3}", student.Marks[0], student.Marks[1], student.Marks[2], student.Marks[3])
    select new XElement("Student",
        new XElement("ID", student.ID),
        new XElement("Name", student.Name),
        new XElement("Address", student.Address),
        new XElement("Marks", x)
    ) //end of "student"
); //end of "Root"
```

Remember to use 'System.Xml.Linq' namespace that supports XElement in LINQ. Now, it's time to execute the above query as:

```
string XMLFileInformation = "<?xml version=\"1.0\"?>\n" + studentsToXML;
Console.WriteLine(XMLFileInformation);
```


Output on Console: Here is what my above code generated for me.

```
<?xml version="1.0"?>
<School>
  <Student>
    <ID>1</ID>
    <Name>Abhimanyu K Vatsa</Name>
    <Address>Bokaro</Address>
    <Marks>97,92,81,90</Marks>
  </Student>
  <Student>
    <ID>2</ID>
    <Name>Deepak Kumar</Name>
    <Address>Dhanbad</Address>
    <Marks>70,56,87,69</Marks>
  </Student>
  <Student>
    <ID>3</ID>
    <Name>Mohit Kumar</Name>
    <Address>Dhanbad</Address>
    <Marks>78,76,81,56</Marks>
  </Student>
  <Student>
    <ID>4</ID>
    <Name>Geeta K</Name>
    <Address>Bokaro</Address>
    <Marks>95,81,54,67</Marks>
  </Student>
</School>
```

If you want to build an real xml file then use following line

```
File.AppendAllText("C:\\Database.xml", XMLFileInformation);
```

Remember to use 'System.IO' namespace to create xml file on disk. Now, open 'C drive' and file a new xml file named Database.xml.

Same way you can do this for outer data sources (not in-memory) too.

Part – 10

In this part you will learn how to perform some calculations inside LINQ Query.

Let's assume we have following data in data source:

ProductName	SupplierID	UnitPrice	UnitsInStock
Chai	1	18.0000	39
Chang	1	19.0000	17
Aniseed Syrup	1	10.0000	13

And we want to find-out Product's Total Price by multiplying UnitPrice and UnitInStock data. So, what would be the LINQ query to find the calculated data?

Let's write a query to solve this:

```
var query = from std in DNDC.Products
             where std.SupplierID == 1
             select new { ProductName = std.ProductName, ProductTotalPrice = String.Format("Total Price = {0}", (std.UnitPrice) * (std.UnitsInStock)) };
```

To understand it, look at the above underlined query, you will find multiplication is being done on two different columns that is on UnitPrice and UnitInStock. Once we get the multiplied value that will be casted as string equivalent data and then will assign this to ProductTotalPrice variable. So, we have ProductName and ProductTotalPrice in query that will be executed in foreach loop to get entire data as given below:

```
foreach (var q in query)
{
    Console.WriteLine("Product Name: " + q.ProductName + ", Total Price: " + q.ProductTotalPrice + "");
}
```

When you execute above loop, you will get following output:

```
Product Name: Chai, Total Price: Total Price = 702.000000
Product Name: Chang, Total Price: Total Price = 323.000000
Product Name: Aniseed Syrup, Total Price: Total Price = 130.000000
```

So, using this you can find calculated information from LINQ query too.

Part – 11

In this part you will look at some differences between LINQ Query Syntax and Method Syntax used in LINQ.

All the queries we learned in this series so far are known as LINQ Query Syntax that was introduced with C# 3.0 release. Actually, at compile time queries are translated into something that the CLR understands. So, it makes a method call at runtime to produce standard SQL queries that uses real SQL keys like WHERE, SELECT, GROUPBY, JOIN, MAX, AVERAGE, ORDERBY and so on. So, we have a direct way, to avoid such method calls to produce standard SQL and that is known as Method Syntaxes.

If you ask me, I'll recommend query syntax because it is usually simpler and more readable; however there is no semantic difference between method syntax and query syntax. In addition, some queries, such as those that retrieve the number of elements that match a specified condition, or that retrieve the element that has the maximum value in a source sequence, can only be expressed as method calls. The reference documentation for the standard query operators in the System.Linq namespace generally uses method syntax. Therefore, even when getting started writing LINQ queries, it is useful to be familiar with how to use method syntax in queries and in query expressions themselves. [Source: MSDN]

Let's look at the example, which will produce same result using Query Syntax and Method Syntax.

```
int[] numbers = new int[10] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

var query1 = from num in numbers
              where (num % 2) == 0
              select num;

var query2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);

Console.WriteLine("Query Syntax");
foreach (int i in query1)
{
    Console.Write(i + " ");
}

Console.WriteLine("\nMethod Syntax");
foreach (int i in query2)
{
    Console.Write(i + " ");
}
```

If you run above code, you will get following output:

Query Syntax

2 4 6 8 10

Method Syntax

2 4 6 8 10

In query2, I'll be using Method Syntax and you can see it also produce same output. You can also see I'm using where, orderby clauses like a method and in the method I'm using Lambda syntaxes.

I hope you will find this ebook useful. Thanks for reading.

Abhimanyu Kumar Vatsa

Microsoft MVP, ASP.NET/IIS

Blog: www.itorian.com

Twitter: <http://twitter.com/itorian>

Facebook: <https://www.facebook.com/2050.itorian>