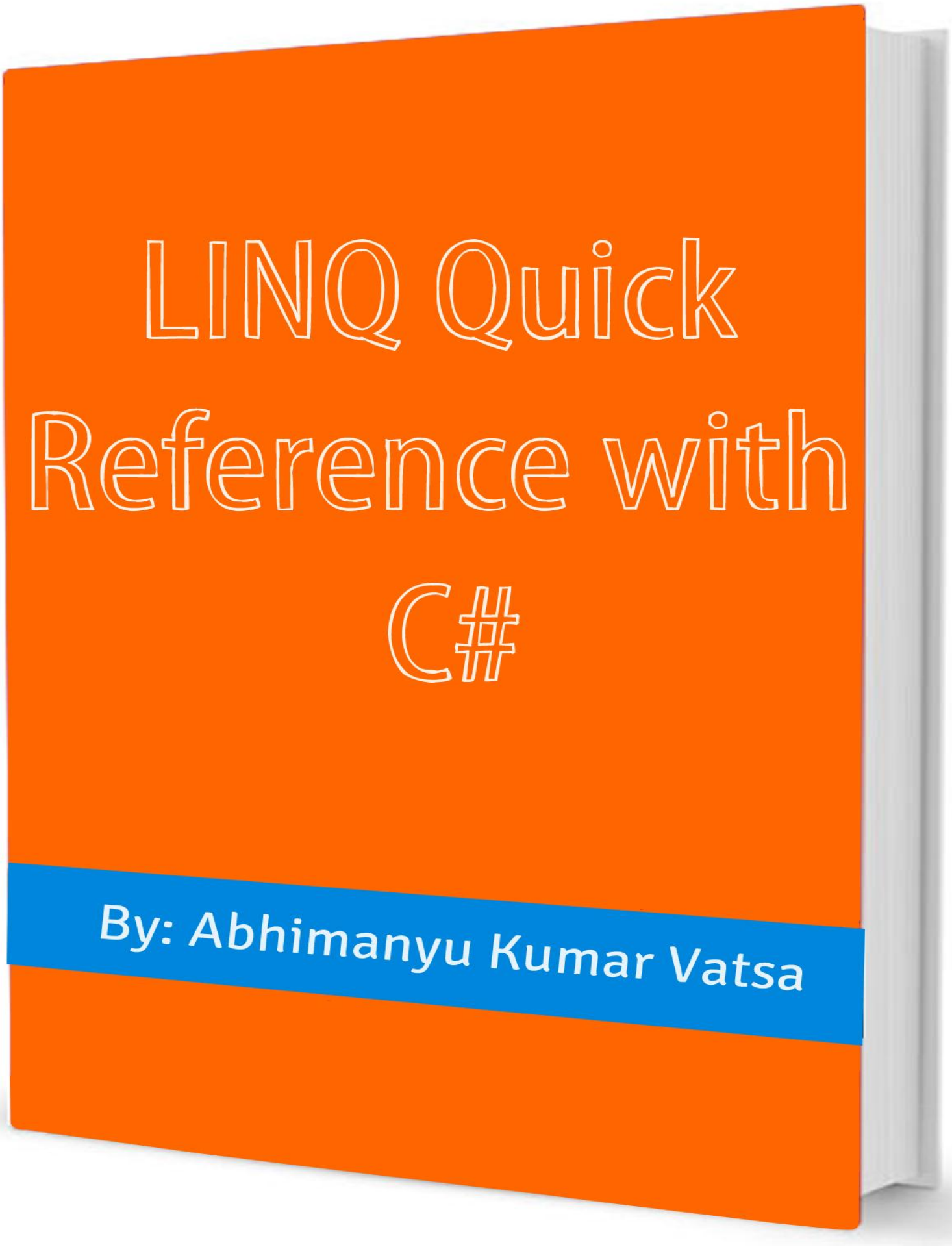

LINQ Quick Reference with C#
By: Abhimanyu Kumar Vatsa

LINQ Quick Reference with C#

Abhimanyu Kumar Vatsa
Microsoft & C# Corner MVP,
Blogs at ITORIAN.COM

Sam Hobbs
Editor, C# Corner

This book is a basic introduction to **LINQ (Language Integrated Query)** basically for beginners who want to learn complete basic with example of **LINQ**.

Table of Contents

1 Basic Introduction to LINQ

1.1 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. At Microsoft's PDC (Professional Developers Conference) 2005, Anders Hejlsberg and his team presented the LINQ approach and its components released with the .NET 3.5 Framework.

1.2 Benefits of LINQ

Some of the major benefits of LINQ are:

1. LINQ is integrated into the C# and VB languages and it provides syntax highlighting and IntelliSense features and even you can debug your queries using the integrated debugger in Visual Studio.
2. Using LINQ it is possible to write code much faster than older queries and lets you save half of the time spent writing queries.
3. Using LINQ you can easily see the relationships among 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.

1.3 Older 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(aÂ¦);//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" retrieves 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.

2 Examples Related LINQ

You will see 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 subsequent parts will dig in depth.

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

2.1 LINQ to Array

Let's look at a Program to find Even Numbers 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.Write("{0,1} ", num);
            }

            Console.ReadKey();
        }
    }
}

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

Data Source

Query

Executing Query

If you execute the above program, you will get "2 4 6 8 10" as the output. You will notice 3 actions above, these actions will always be changed depending upon our project requirements. In the above example, the "data source" was an array that implicitly supports the generic IEnumerable<T> interface.

2.2 LINQ to XML

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

Student.XML File

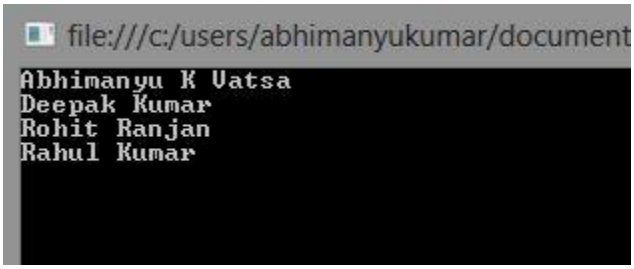
```
<?xml version="1.0" encoding="utf-8" ?>
<students>
   <student id="1">
      <name>Abhimanyu K Vatsa</name>
      <address>Bokaro Steel City</address>
   </student>
   <student id="2">
      <name>Deepak Kumar</name>
      <address>Dhanbad</address>
   </student>
   <student id="3">
      <name>Rohit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

```
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



2.3 LINQ to SQL

In LINQ to SQL, we first 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.

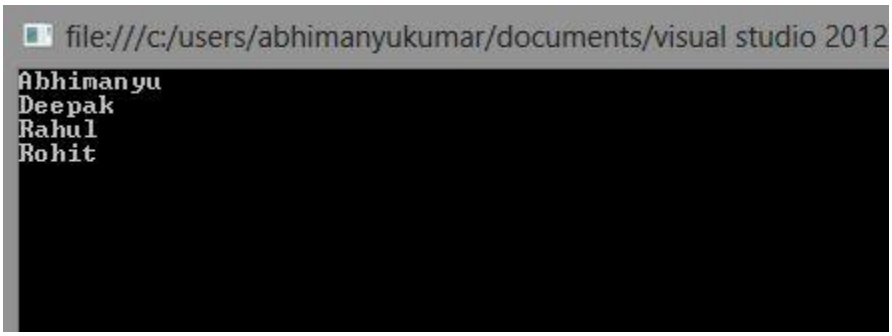
Now, let's look at the sample program.

```
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



Very simple examples on LINQ so far, stay tuned, you will find more stuff.

3 Generic Collection of LINQ

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

Collection Types	Advantages	Disadvantages
Arrays	Strongly Typed (Type Safe)	• Zero Index Based • Cannot grow in size once initialized
.NET Collections (like ArrayList, Hashtable etc)	• Can grow in size • Convenient to work with Add, Remove	Not Strongly Typed
.NET Generic Collections (like List<T>, GenericList<T> etc)	• Type Safe like arrays • Can grow automatically like ArrayList • Convenient to work with Add, Remove	

Now, let's discuss each of the collection types given in the above table to feel the real pain of development.

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:

- 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".
- If you declare an 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 the array is type-safe.
- The array is an index based, so you always need to write index number to insert value in the collection.
- You aren't adding and 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.rem
                    Remove
                    RemoveAt
                    RemoveRange
            foreach (int n in numbers)
            {
                Console.WriteLine(n);
            }

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

If you use ArrayList:

- You will get the auto grow feature, which means the initial size will not limit you.
- You don't have the option to declare the data type at declaration time initially, so you can add any value type 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 however have any property like index.

List<T>(System.Collection.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, a list has both the features that we have in arrays and arraylists.

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 return collections and we don't even know what type of data we will get from the database and that is the reason we use a 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 the above example, we have used the IEnumerable<T> interface and a LINQ query and then by using a 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 the above example you can see, using a LINQ query I'm not only limited to select the string value that is LastName, I'm also able to select an integer value that is Age.

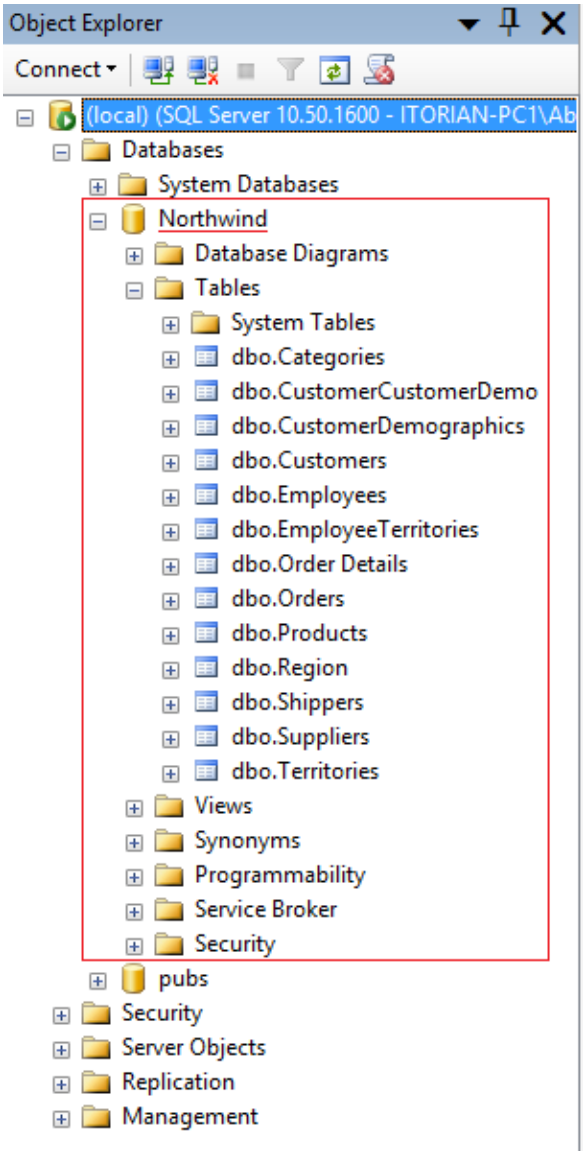
So, this could be the great example of generic classes.

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

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

4 Demo Project Explore LINQ queries

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

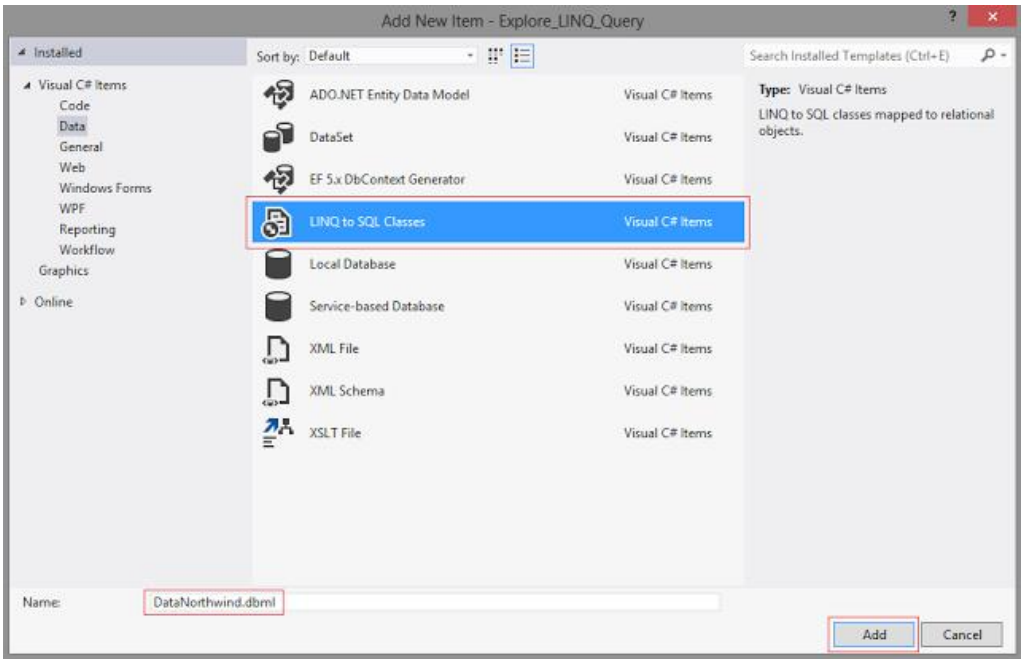


Step 1: Creating the New Project

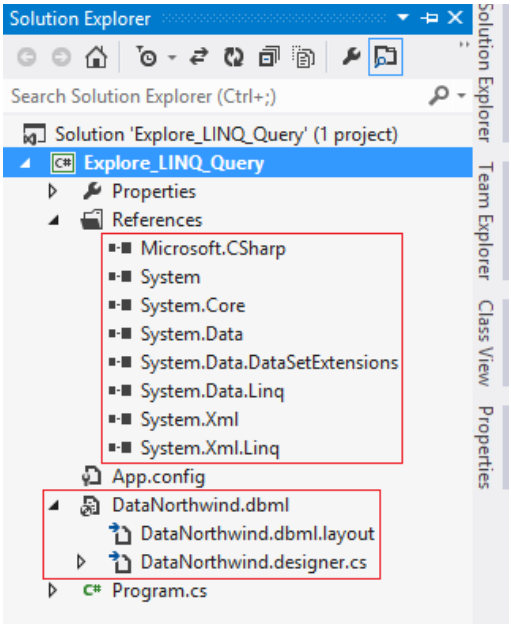
Open the Visual Studio instance and create a new project New > Project > Console Application and name the project whatever you wish.

Step 2: Adding LINQ to SQL Classes

Now, in the Solution Explorer add a new item "LINQ to SQL Classes" using a nice name like "DataNorthwind.dbml".

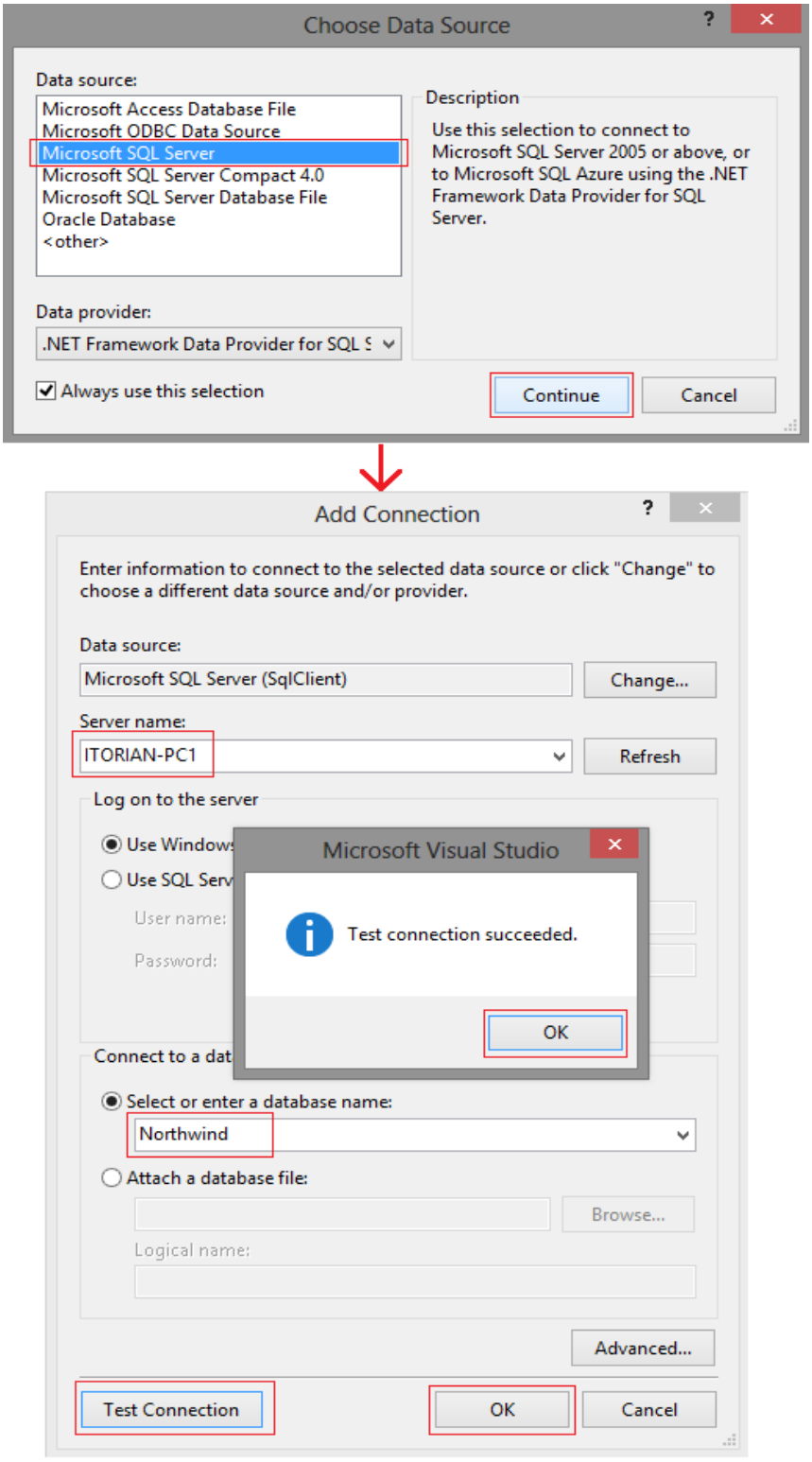


This will add all required libraries as well as classes to your project for database communication. Actually, this DBML file will become a layer between the DB and your project and provide you all the IntelliSense features.

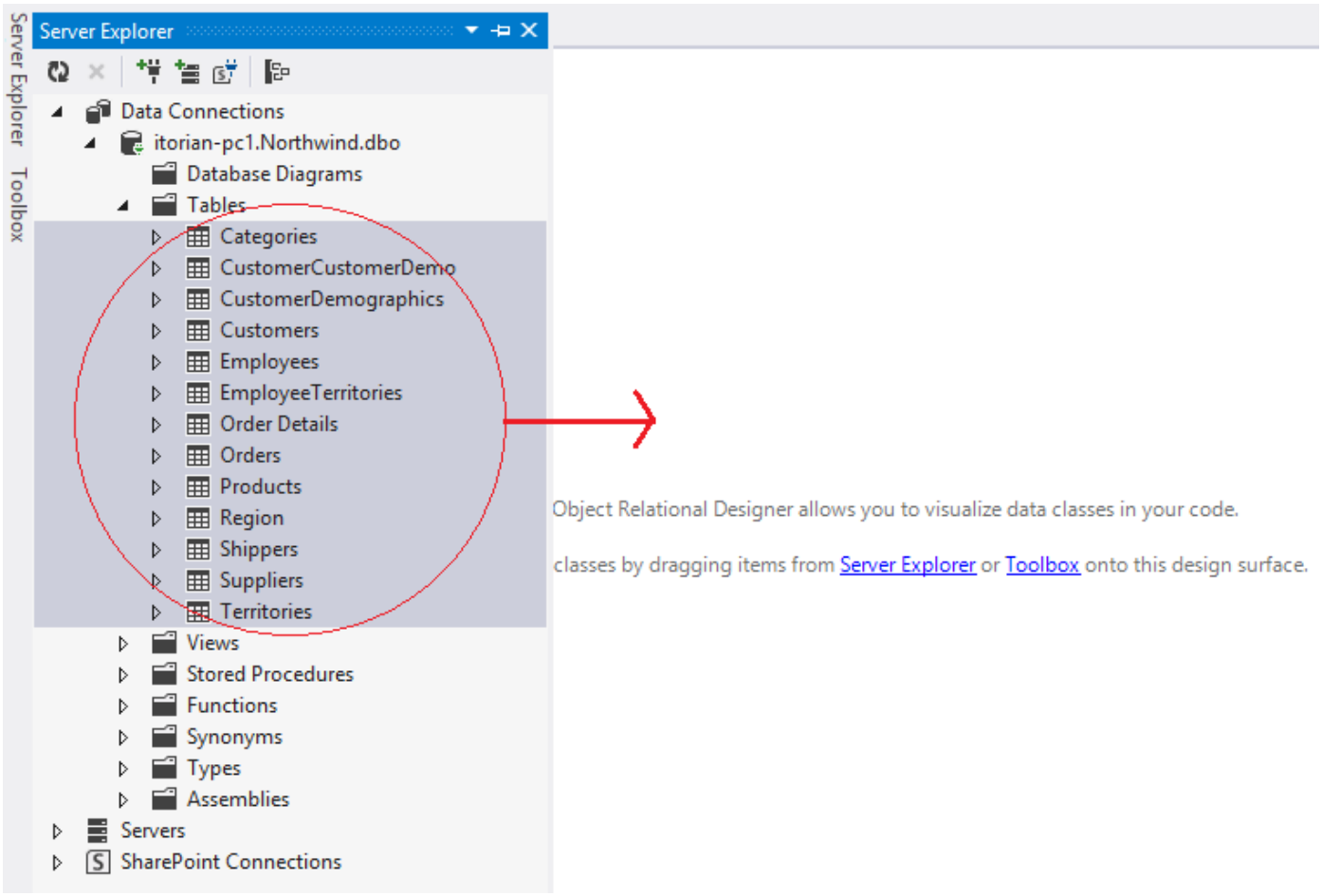


Step 3: Adding New Connection

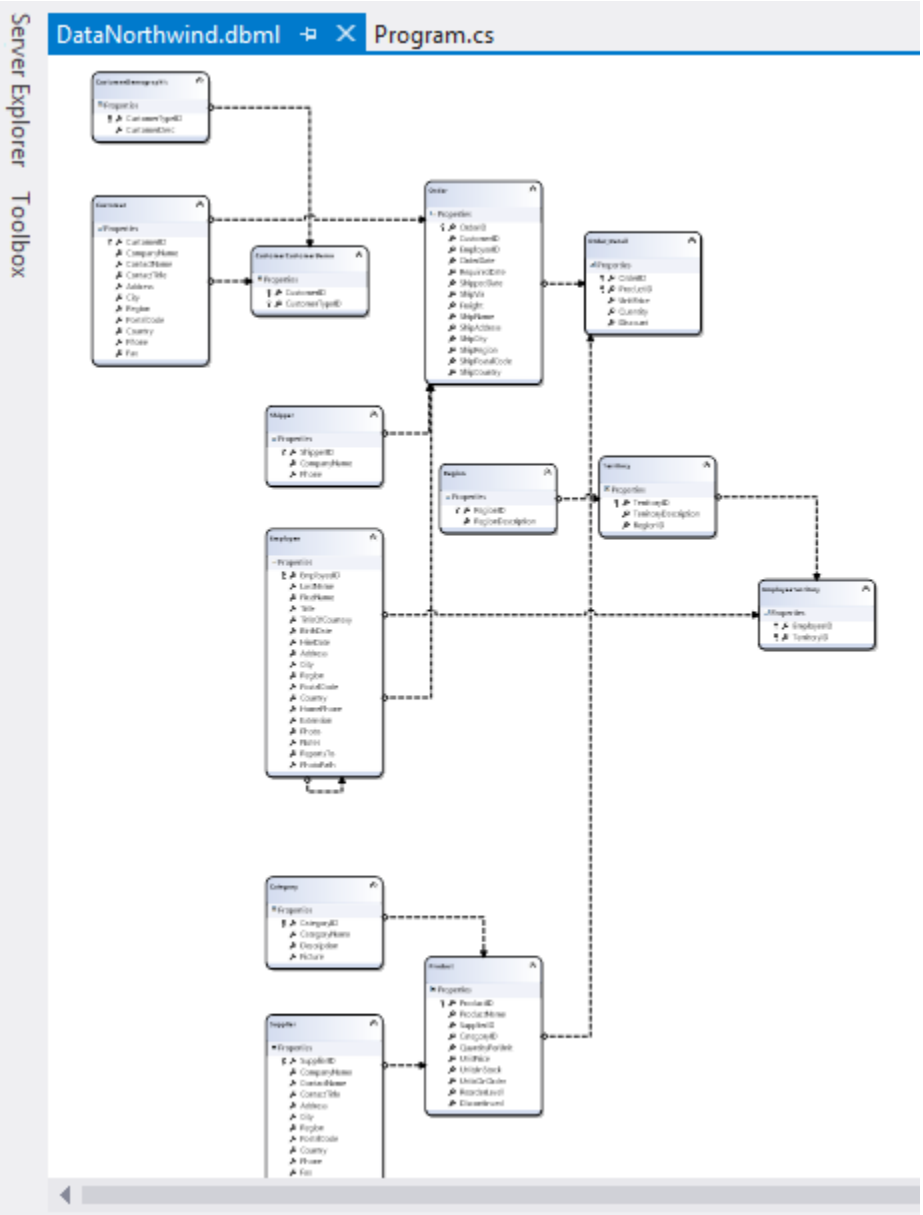
Open "Server Explorer" and add a new SQL Server connection, as in the following image:



Once you are done with the above, drag the tables from the "Server Explorer" to the "DataNorthwind.dbml" file designer; see:



Now, you will get following DB diagram in the designer:



You will notice the following things here:

1. A new connection string in the App.config file.
2. Every setup required to communicate with the DB like DataNorthwindDataContext, TableAttribute, EntitySet etc in was created in the DataNorthwind.designer.cs file.

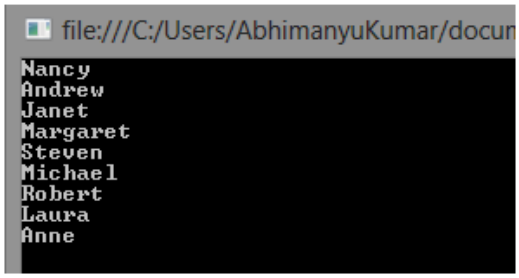
Step 4: Coding and Running Project

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

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();
        }
    }
}
```



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

In an upcoming article, we will look at various LINQ queries and will test it using the preceding project.

5 Selecting Data Using LINQ

Selecting/Querying

Selecting data using LINQ queries is very simple. You will get full intellisense support while querying a database and even compile-time error checking that adds significant advantages to LINQ. Once you have learned it, you will love to use this in all your DB projects. I'm loving this, it is great.

In the image given below, you will find the sample code for selecting the employee's "FirstName" and "LastName" from the "Employee" table of the "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);
}
```

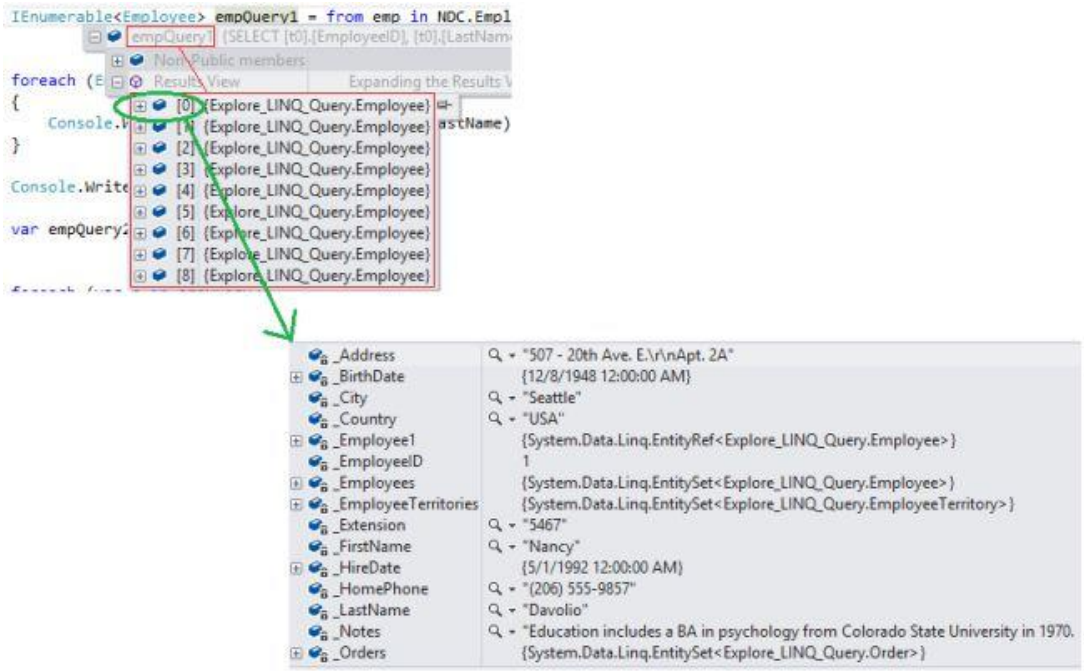
```
file:///C:/Users/AbhimanyuKumar/documents/visual stud
Using IEnumerable<T> Interface
Nancy, Davolio
Andrew, Fuller
Janet, Leverling
Margaret, Peacock
Steven, Buchanan
Michael, Suyama
Robert, King
Laura, Callahan
Anne, Dodsworth
Using var
Nancy, Davolio
Andrew, Fuller
Janet, Leverling
Margaret, Peacock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

In the above image, you can see, I'm using the same queries two times, but there is a difference, don't go on output screen only. If you look at the code carefully, you will find I'm using IEnumerable<Employee> in the first block of code and just a "var" in the second block of code. Also note, for IEnumerable<Employee> I'm using "Employee" in the foreach loop and for "var" I'm using "var" in the 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 the following image (just hover the mouse over "empQuery1" when the compiler is on break). When you expand the index[], you will find all the data associated with that row/record.



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 the from clause, no big difference; you still get all the benefits.

Now, let's discuss the query part.

```
= 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 is first, 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 an Employees table and that is why I have used "NDC.Employees".

Executing Query (using foreach loop)

When you are 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);
}
```

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

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

6 Query Operations in LINQ

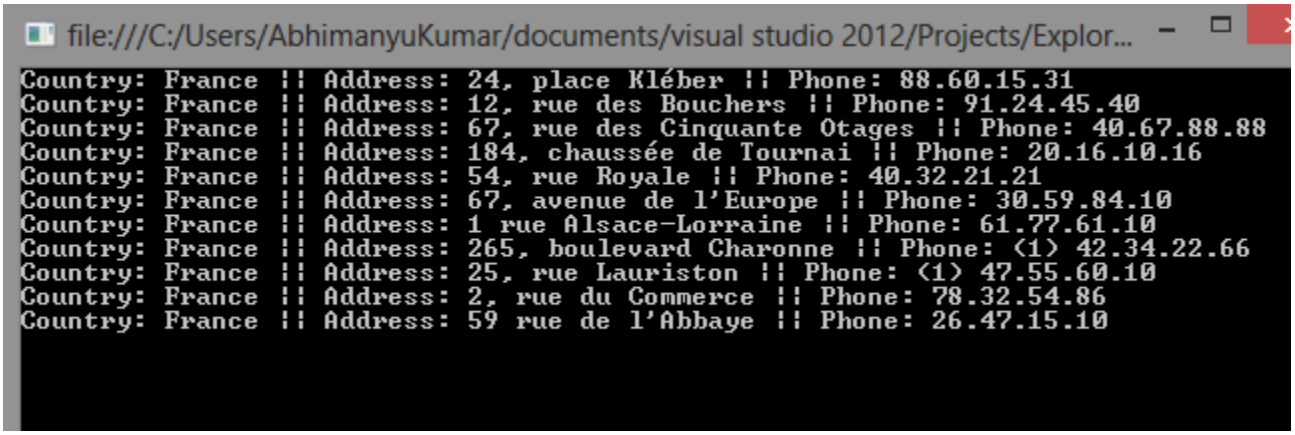
6.1 Filter

Filter is the most common query operation. A filter is applied 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 a 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 the above query, I'm asking for only those records who's "Country" is "France". And in the foreach loop, "Country", "Address" and "Phone" are separated by "||" and the same in output.



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

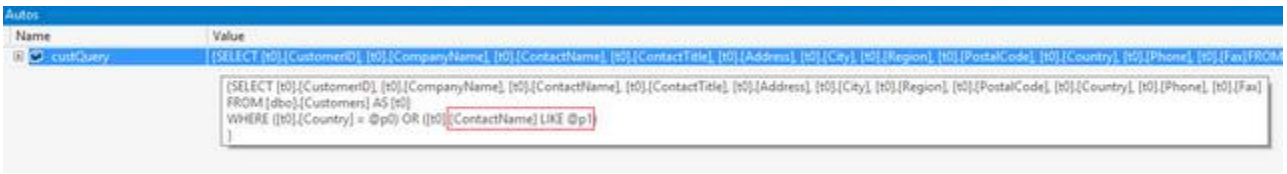
```
var custQuery = from cust in NDC.Customers
            where cust.Country == "France" && cust.ContactName.StartsWith("a")
            select cust;
```

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 queries, "&&" is being used for "And" and "||" is being used for "Or".

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



We will look more only such available "keys" in a subsequent article.

6.2 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;
```

6.3 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:

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:

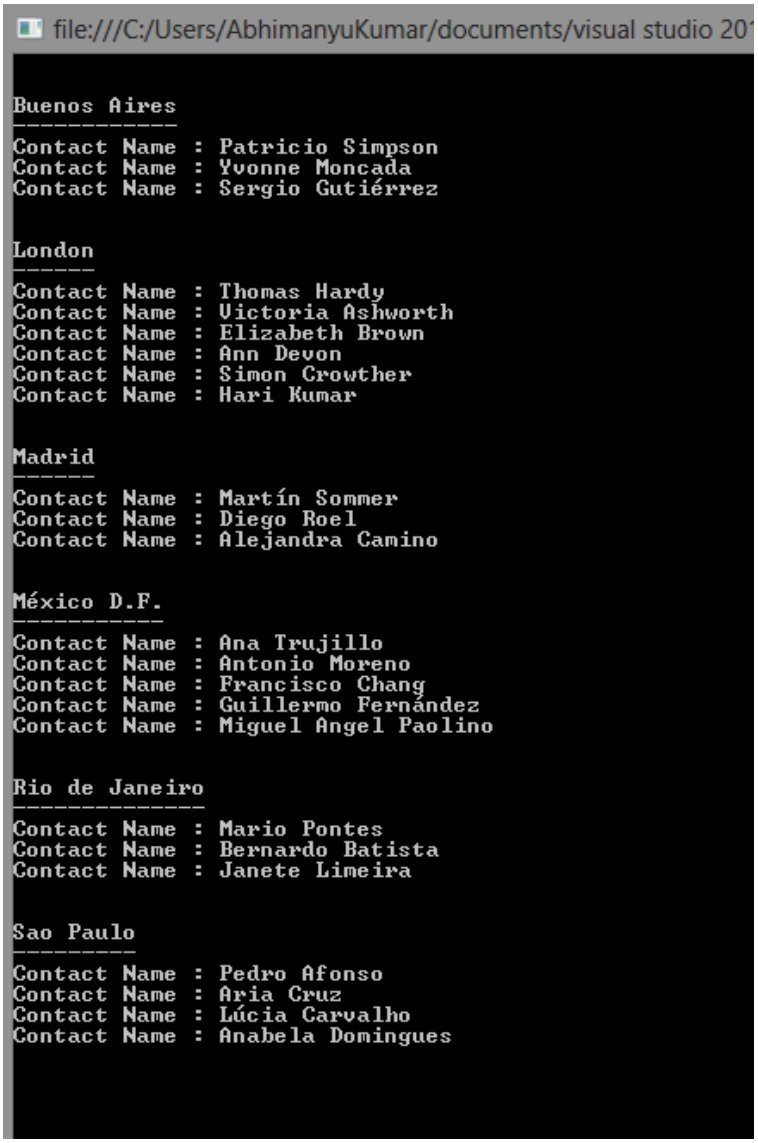
```
var custQuery = from cust in NDC.Customers
                group cust by cust.City into custGroup
                where custGroup.Count() > 2
                select custGroup;
```

And the foreach loop will be the same; see:

```
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 the following output:



Join

Join operations create associations among 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 should we write to select names from the two tables "Customer" and "Employee" depending upon matching city?

Answer: The 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 the foreach loop, we will write:

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

Output:

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

7 Joining Multiple Data Sources Using “Concate” Key in LINQ

Now, we will take a look at doing joins using Concate key in a LINQ query.

Assume we have the 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 Student's names and Teachers from "Bokaro" by Concat. So, we can use a 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);
```

Output:

Abhimanyu K Vatsa
Geeta K
Ranjeet Kumar

8 Customizing LINQ's "Select" Statement

Now, you will learn how to customize the LINQ's "select" statement to select a subset of each Source Element.

Assuming the following as the 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 the name of the students then use the 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 the name and address of the student then use the 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 the 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.Write(": 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 the above query, we have the Mark field that will produce a list and that's why we need to use a nested foreach loop to get the list item and at the same time we are adding mark items.

9 Transforming Data Sources objects into XML Using LINQ

In this Topic of LINQ you will learn how to transform data sources into XML

In-Memory Data Sources

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 a 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 the "System.Xml.Linq" namespace that supports XElement in LINQ. Now, it's time to execute the preceding query as:

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

Output on Console:

```
<?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 a real XML file then use the following line:

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

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

In the same manner you can do this for outer data sources (not in-memory) too.

10 Performing Calculation in LINQ

You learned how to transform source data or objects into a XML file and this article will teach how to perform some calculations in a LINQ query.

Let's assume we have the following data in a 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 determine the Product's Total Price by multiplying UnitPrice and UnitInStock data. So, what would be the LINQ query to evaluate the 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 preceding underlined query, you will see that a multiplication is performed on the two columns UnitPrice and UnitInStock. Once we get the result it will be converted to a string and then it will be assigned to the ProductTotalPrice variable. So, we have ProductName and ProductTotalPrice in the query that will be executed in a foreach loop to get the 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 the 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 way you can find the calculated information from a LINQ query too.

11 LINQ Query syntax and Method syntax

Now you will look at some differences between LINQ Query Syntax and Method Syntax used in LINQ.

All the queries we have learned in this series so far are known as LINQ Query Syntax that was introduced with the 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 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 the use of Method Syntax in queries and in query expressions themselves. [Source: MSDN]

Let's look at the example, which will produce the 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 the above code, you will get the following output:

Query Syntax

2 4 6 8 10

Method Syntax

2 4 6 8 10

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