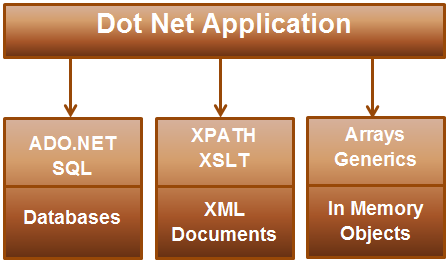
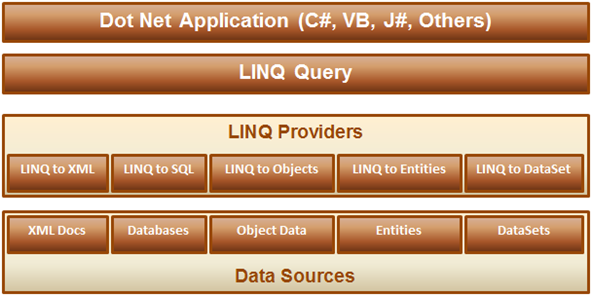
**Why should we use LINQ and what are the benefits of using LINQ**   
   
  
**If the .NET application that is being developed**  
**a) Requires data from SQL Server -**Then the developer has to understand ADO.NET code and SQL specific to SQL Server Database  
**b) Requires data from an XML document -**Then the developer has to understand XSLT & XPATH queries  
  
**c) Need to query objects in memory (List<Customer>, List<Order> etc) -**Then the developer has to understand how to work with objects in memory  
LINQ enables us to work with these different data sources using a similar coding style without having the need to know the syntax specific to the data source.   
It provides intellisense and compile time error checking.   
  
**LINQ Architecture & LINQ Providers**   
  
  
**1.** LINQ query can be written using any .NET supported programming language  
**2.** LINQ provider is a component between the LINQ query and the actual data source, which converts the LINQ query into a format that the underlying data source can understand. For example LINQ to SQL provider converts a LINQ query to T-SQL that SQL Server database can understand.  
  
**For example**, the application that we are developing should display male students in a GridView control as shown below.    
   
  
**To achieve this**  
**Step 1:**We first create the required table

Create Table Students

(

     ID int primary key identity,

     FirstName nvarchar(50),

     LastName nvarchar(50),

     Gender nvarchar(50)

)

GO

Insert into Students values ('Mark', 'Hastings', 'Male')

Insert into Students values ('Steve', 'Pound', 'Male')

Insert into Students values ('Ben', 'Hoskins', 'Male')

Insert into Students values ('Philip', 'Hastings', 'Male')

Insert into Students values ('Mary', 'Lambeth', 'Female')

GO

**Step 2:** Write the required ADO.NET code to retrieve data from SQL Server database as shown below.

using System;

using System.Collections.Generic;

using System.Configuration;

using System.Data.SqlClient;

namespace Demo

{

    public partial class WebForm1 : System.Web.UI.Page

    {

        protected void Page\_Load(object sender, EventArgs e)

        {

            string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;

            SqlConnection con = new SqlConnection(cs);

            SqlCommand cmd = new SqlCommand

                ("Select ID, FirstName, LastName, Gender from Students where Gender='Male'", con);

            List<Student> listStudents = new List<Student>();

            con.Open();

            SqlDataReader rdr = cmd.ExecuteReader();

            while (rdr.Read())

            {

                Student student = new Student();

                student.ID = Convert.ToInt32(rdr["ID"]);

                student.FirstName = rdr["FirstName"].ToString();

                student.LastName = rdr["LastName"].ToString();

                student.Gender = rdr["Gender"].ToString();

                listStudents.Add(student);

            }

            con.Close();

            GridView1.DataSource = listStudents;

            GridView1.DataBind();

        }

    }

    public class Student

    {

        public int ID { get; set; }

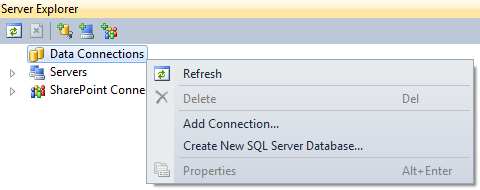
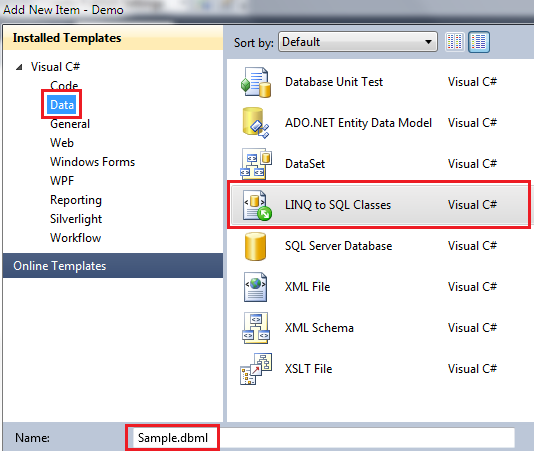
        public string FirstName { get; set; }

        public string LastName { get; set; }

        public string Gender { get; set; }

    }

}

If we misspell table or column names in the SQL Query, we will not know about it at compile time. At run time the page crashes and that's when we will know about this error. Also notice that there is no **intellisense**when typing table and column names. Misspelled column names when reading from the reader will also cause the same problem. With LINQ we will have intellisense and compile time error checking.  
  
**Now let's achieve the same thing using LINQ to SQL.**  
**Step 1:** Create a new empty asp.net web application and name it **Demo**  
  
**Step 2:**Click on **"View"**menu item and select **"Server Explorer"**  
  
**Step 3:**In **"Server Explorer"**window, right click on **"Data Connections"**and select **"Add Connection"**option   
   
  
**Step 4:** Specify your SQL Server name and the credentials to connect to SQL Server. At this point we should be connected to SQL Server from Visual Studio.  
  
**Step 5:** Adding **LINQ to SQL Classes**  
**a)** Right click on the **"Demo"**project in solution explorer and select **"Add New Item"**option  
**b)** In the **"Add New Item"**dialog box, select **"Data"**under **"Installed Templates"**  
**c)** Select **"LINQ to SQL Classes"**  
**d)** Set **Name = Sample.dbml**  
**e)** Finally click **"Add"**button   
   
  
**Step 6:** From **"Server Explorer"**window drag and drop **"Students"**table onto **"Sample.dbml"**designer file.  
  
**Step 7:**Add a webform. Drag and drop a gridview control.  
  
**Step 8:**Copy and paste the following code in the code-behind file

using System;

using System.Linq;

namespace Demo

{

    public partial class WebForm1 : System.Web.UI.Page

    {

        protected void Page\_Load(object sender, EventArgs e)

        {

            SampleDataContext dataContext = new SampleDataContext();

            GridView1.DataSource = from student in dataContext.Students

                                   where student.Gender == "Male"

                                   select student;

            GridView1.DataBind();

        }

    }

}

Notice that, with LINQ we are getting intellisense. If we misspell the table or column names we will get to know about them at compile time. Open SQL Profiler. Run the application, and notice the SQL Query that is generated.

Part-2

**There are 2 ways to write LINQ queries using these Standard Query Operators**  
**1.** **Using Lambda Expressions.**  **2.** **Using SQL like query expressions**  
  
The **Standard Query Operators**are implemented as extension methods on IEnumerable<T> interface  
  
LINQ queries written using SQL like query expressions are translated into their lambda expressions before they are compiled.   
  
We will use the following Student class in this demo. GetAllStudents() is a static method that returns List<Student>. Since List<T> implements IEnumerable<T>, the LINQ Standard Query Operators will be available and can be applied on List<Student>.

public class Student

{

    public int ID { get; set; }

    public string Name { get; set; }

    public string Gender { get; set; }

    public static List<Student> GetAllStudents()

    {

        List<Student> listStudents = new List<Student>();

        Student student1 = new Student

        {

            ID = 101,

            Name = "Mark",

            Gender = "Male"

        };

        listStudents.Add(student1);

        Student student2 = new Student

        {

            ID = 102,

            Name = "Mary",

            Gender = "Female"

        };

        listStudents.Add(student2);

        Student student3 = new Student

        {

            ID = 103,

            Name = "John",

            Gender = "Male"

        };

        listStudents.Add(student3);

        Student student4 = new Student

        {

            ID = 104,

            Name = "Steve",

            Gender = "Male"

        };

        listStudents.Add(student4);

        Student student5 = new Student

        {

            ID = 105,

            Name = "Pam",

            Gender = "Female"

        };

        listStudents.Add(student5);

        return listStudents;

    }

}

The **LINQ query**should return just the **Male**students.   
  
**LINQ query using Lambda Expressions.**

IEnumerable<Student>students=Student.GetAllStudents**()**

.Where**(**student=>student.Gender=="Male"**);**  
  
**LINQ query using using SQL like query expressions**

IEnumerable<Student>students=fromstudentinStudent.GetAllStudents**()**

wherestudent.Gender=="Male"

selectstudent**;**  
  
**To bind the results of this LINQ query to a GridView**  
GridView1.DataSource = students;  
GridView1.DataBind();

part3-

**What are Extension Methods**  
Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.   
**Extension methods are a special kind of static method**, but they are called as if they were instance methods on the extended type.   
  
**For client code written in C# and Visual Basic**, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.  
  
**Let us understand what this definition actually means.**  
**LINQ's standard query**operators (select, where etc ) are implemented in Enumerable class as extension methods on the IEnumerable<T> interface.  
  
**Now look at the following query**

List<int> Numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

IEnumerable<int> EvenNumbers = Numbers.Where(n => n % 2 == 0);

In spite of **Where()**method not belonging to **List<T>**class, we are still able to use it as though it belong to **List<T>**class. This is possible because **Where()**method is implemented as extension method in **IEnumerable<T>**interface and **List<T>**implements **IEnumerable<T>**interface.  
  
**How to implement extension methods**  
We want to define a method in the string class (let's call it ChangeFirstLetterCase), which will change the case of the first letter of the string. For example, if the first letter of the string is lowercase the function should change it to uppercase and viceversa.  
  
We want to be able to call this function on the string object as shown below.

string result = strName.ChangeFirstLetterCase();

Defining **ChangeFirstLetterCase**() method directly in the **string**class is not possible as we don't own the string class. It belongs to .NET framework. Another alternative is to write a wrapper class as shown below.

public class StringHelper

{

    public static string ChangeFirstLetterCase(string inputString)

    {

        if (inputString.Length > 0)

        {

            char[] charArray = inputString.ToCharArray();

            charArray[0] = char.IsUpper(charArray[0]) ?

                char.ToLower(charArray[0]) : char.ToUpper(charArray[0]);

            return new string(charArray);

        }

        return inputString;

    }

}

Wrapper class works, but the problem is, we cannot call **ChangeFirstLetterCase**() method using the following syntax.  
string result = strName.ChangeFirstLetterCase();  
  
Instead we have to call it as shown below.  
string result = StringHelper.ChangeFirstLetterCase(strName);  
  
Convert **ChangeFirstLetterCase**() method to an extension method to be able to call it using the following syntax, as though it belongs to string class.  
string result = strName.ChangeFirstLetterCase();  
  
To **convert ChangeFirstLetterCase() method to an extension method**, make the following 2 changes  
**1.** Make StringHelper static class  
**2.** The type the method extends should be passed as a first parameter with this keyword preceeding it.  
  
With these 2 changes, we should be able to call this extension method in the same way we call an instance method. Notice that the extension method shows up in the intellisense as well, but with a different visual clue.  
string result = strName.ChangeFirstLetterCase();  
  
Please note that, we should still be able to call this extension method using wrapper class style syntax. In fact, behind the scene this is how the method actually gets called. Extension methods are just a syntactic sugar.  
string result = StringHelper.ChangeFirstLetterCase(strName);  
  
So, this means we should also be able to call LINQ extension methods (select, where etc), using wrapper class style syntax. Since all LINQ extension methods are defined in Enumerable class, the syntax will be as shown below.

List<int> Numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

IEnumerable<int> EvenNumbers = Enumerable.Where(Numbers, n => n % 2 == 0);

Linq in c#-

In a LINQ query, you are always working with objects.

LINQ stands for Language Integrated Query. LINQ enables us to query any type of data store (SQL Server, XML documents, Objects in memory etc).

All LINQ query operations consist of three distinct actions:

Obtain the data source.

Create the query.

Execute the query.

ex-1

class IntroToLINQ

{

static void Main()

{

// The Three Parts of a LINQ Query:

// 1. Data source.

int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

// 2. Query creation.

// numQuery is an IEnumerable<int>

var numQuery =

from num in numbers

where (num % 2) == 0

select num;

// 3. Query execution.

foreach (int num in numQuery)

{

Console.Write("{0,1} ", num);

}

}

}

//A query is executed in a foreach statement, and foreach requires IEnumerable or IEnumerable<T>.

LINQ and Generic Types (C#)

LINQ queries are based on generic types, which were introduced in version 2.0 of the .NET Framework.

You do not need an in-depth knowledge of generics before you can start writing queries. However, you may want to understand two basic concepts:

1-When you create an instance of a generic collection class such as List<T>, you replace the "T" with the type of objects that the list will hold.

For example, a list of strings is expressed as List<string>, and a list of Customer objects is expressed as List<Customer>.

A generic list is strongly typed and provides many benefits over collections that store their elements as Object.

If you try to add a Customer to a List<string>, you will get an error at compile time.

It is easy to use generic collections because you do not have to perform run-time type-casting.

2-IEnumerable<T> is the interface that enables generic collection classes to be enumerated by using the foreach statement.

Generic collection classes support IEnumerable<T> just as non-generic collection classes such as ArrayList support IEnumerable.

IEnumerable<T> variables in LINQ Queries

LINQ query variables are typed as IEnumerable<T> or a derived type such as IQueryable<T>.

When you see a query variable that is typed as IEnumerable<Customer>, it just means that the query, when it is executed, will produce a sequence of zero or more Customer objects.

IEnumerable<Customer> customerQuery =

from cust in customers

where cust.City == "London"

select cust;

foreach (Customer customer in customerQuery)

{

Console.WriteLine(customer.LastName + ", " + customer.FirstName);

}

Letting the Compiler Handle Generic Type Declarations

If you prefer, you can avoid generic syntax 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.

The following example produces the same compiled code as the previous example:

C#

var customerQuery2 =

from cust in customers

where cust.City == "London"

select cust;

foreach(var customer in customerQuery2)

{

Console.WriteLine(customer.LastName + ", " + customer.FirstName);

}

The var keyword is useful when the type of the variable is obvious or when it is not that important to explicitly specify nested generic types such as those that are produced by group queries. In general,

we recommend that if you use var, realize that it can make your code more difficult for others to read. For more information, see Implicitly Typed Local Variables.