LINQ Tutorial

Language-Integrated Query (LINQ) is a powerful query language introduced with .Net 3.5 & Visual Studio 2008.

LINQ can be used with C# to query different data sources.

Why LINQ?

To understand why we should use LINQ, let's look at some examples.

Suppose you want to find list of teenage students from an array of Student objects.

Before C# 2.0, we had to use a 'foreach' or a 'for' loop to traverse the collection to find a particular object.

For example, we had to write the following code to find all Student objects from an array of Students where the age is between 12 and 20 (for teenage 13 to 19):

Example: Use for loop to find elements from the collection in C# 1.0

```
class Student
{
    public int StudentID { get; set; }
    public String StudentName { get; set; }
    public int Age { get; set; }
}

class Program
{
    static void Main(string[] args)
    {
        Student[] studentArray = {
            new Student() { StudentID = 1, StudentName = "John", Age = 18 },
            new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 },
            new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 },
            new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 },
            new Student() { StudentID = 5, StudentName = "Ron" , Age = 31 },
            new Student() { StudentID = 6, StudentName = "Chris",  Age = 17 },
            new Student() { StudentID = 7, StudentName = "Rob",Age = 19  },
        };

        Student[] students = new Student[7];

        int i = 0;
        foreach (Student std in studentArray)
        {
            if (std.Age > 12 && std.Age < 20)
            {
                students[i] = std;
                i++;
            }
        }
    }
}
```

Use of for loop is cumbersome, not maintainable and readable.

C# 2.0 introduced **delegate**, which can be used to handle this kind of a scenario, as shown below.

Example: Use delegates to find elements from the collection in C# 2.0

```
public class Student{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }

}

delegate bool FindStudent(Student std);

class StudentExtension
{
    public static Student[] where(Student[] stdArray, FindStudent del)
    {
        int i=0;
        Student[] result = new Student[7];
        foreach (Student std in stdArray)
            if (del(std))
            {
                result[i] = std;
                i++;
            }

        return result;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Student[] studentArray = {
            new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
            new Student() { StudentID = 2, StudentName = "Steve",  Age = 21 } ,
            new Student() { StudentID = 3, StudentName = "Bill",  Age = 25 } ,
            new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
            new Student() { StudentID = 5, StudentName = "Ron" , Age = 31 } ,
            new Student() { StudentID = 6, StudentName = "Chris",  Age = 17 } ,
            new Student() { StudentID = 7, StudentName = "Rob",Age = 19  } ,
        };

        Student[] students = StudentExtension.where(studentArray, delegate(Student std){
                return std.Age > 12 && std.Age < 20;
            });
        }
    }
}
```

So, with C# 2.0, you got the advantage of **delegate** in finding students with any criteria.

You don't have to use a for loop to find students using different criteria.

For example, you can use the same delegate function to find a student whose StudentId is 5 or whose name is Bill, as below:

```
Student[] students = StudentExtension.where(studentArray, delegate(Student std) {
        return std.StudentID == 5;
    });

//Also, use another criteria using same delegate
Student[] students = StudentExtension.where(studentArray, delegate(Student std) {
        return std.StudentName == "Bill";
    });
```

The C# team felt that they still needed to make the code even more compact and readable.

So they introduced LINQ to query to the different types of collection and get the resulted element(s) in a single statement.

The example below shows how you can use LINQ query with lambda expression to find a particular student(s) from the student collection.

C# 3.0 onwards:

```
public class Student{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }

}
class Program
{
    static void Main(string[] args)
    {
        Student[] studentArray = {
                    new Student() { StudentID = 1, StudentName = "John", age = 18 } ,
                    new Student() { StudentID = 2, StudentName = "Steve",  age = 21 } ,
                    new Student() { StudentID = 3, StudentName = "Bill",  age = 25 } ,
                    new Student() { StudentID = 4, StudentName = "Ram" , age = 20 } ,
                    new Student() { StudentID = 5, StudentName = "Ron" , age = 31 } ,
                    new Student() { StudentID = 6, StudentName = "Chris",  age = 17 } ,
                    new Student() { StudentID = 7, StudentName = "Rob",age = 19  } ,
                };

        // Use LINQ to find teenager students
        Student[] teenAgerStudents = studentArray.Where(s => s.age > 12 && s.age < 20).ToArray();


    }
}
```

As you can see in the above example, we specify different criteria using LINQ operator and lambda expression in a single statement.

Advantages of LINQ:

- **Familiar language:** Developers don’t have to learn a new query language for each type of data source or data format.
- **Less coding:** It reduces the amount of code to be written as compared with a more traditional approach.
- **Readable code:** LINQ makes the code more readable so other developers can easily understand and maintain it.
- **Standardized way of querying multiple data sources:** The same LINQ syntax can be used to query multiple data sources.
- **Compile time safety of queries:** It provides type checking of objects at compile time.
- **IntelliSense Support:** LINQ provides IntelliSense for generic collections.
- **Shaping data:** You can retrieve data in different shapes.

LINQ API

LINQ is nothing but the collection of extension methods for classes that implements IEnumerable and IQueryable interface.

System.Linq namespace includes the necessary classes & interfaces for LINQ.

Enumerable and Queryable are two main static classes of LINQ API that contain extension methods.

Enumerable:

Enumerable class includes extension methods for the classes that implement IEnumerable<T> interface,

this include all the collection types in System.Collections.Generic namespaces such as List<T>, Dictionary<T>, SortedList<T>, Queue<T>, HashSet<T>, LinkedList<T> etc.

LINQ Syntax

There are two basic ways to write a LINQ query to IEnumerable collection

1. Query Syntax or Query Expression Syntax
2. Method Syntax or Method extension syntax or Fluent

LINQ Query Syntax

Query syntax is similar to SQL (Structured Query Language) for the database. It is defined within the C#.

The LINQ query syntax starts with from keyword and ends with select keyword. The following is a sample LINQ query that returns a collection of strings which contains a word "Tutorials".

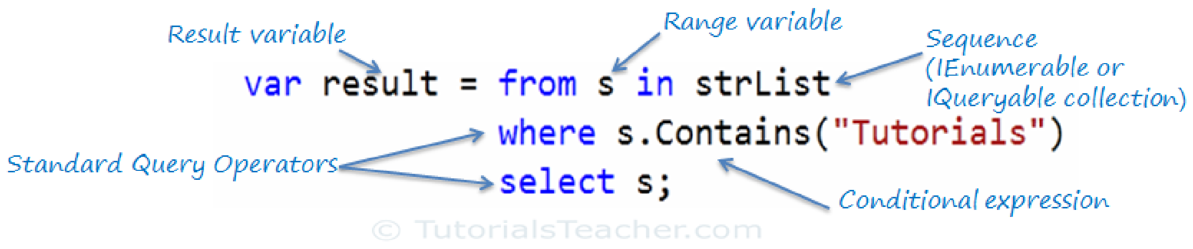
Example: LINQ Query Syntax in C#

```
public class Program
{
    public static void Main()
    {
    // string collection
        IList<string> stringList = new List<string>() {
            "C# Tutorials",
            "VB.NET Tutorials",
            "Learn C++",
            "MVC Tutorials" ,
            "Java"
        };

        // LINQ Query Syntax
        var result = from s in stringList
                    where s.Contains("Tutorials")
                    select s;

        foreach(string str in result){
            Console.WriteLine(str);
        }
    }
}
```

Result:

```
C# Tutorials
VB.NET Tutorials
MVC Tutorials
```

The following figure shows the structure of LINQ query syntax.



```
var result = from s in strList
             where s.Contains("Tutorials")
             select s;
```

Query syntax starts with a ***From*** clause followed by a ***Range*** variable. The ***From*** clause is structured like

`"From rangeVariableName in IEnumerablecollection"`.

In English, this means, from each object in the collection. It is similar to a foreach loop: `foreach(Student s in studentList)`.

After the From clause, you can use different Standard Query Operators to filter, group, join elements of the collection. There are around 50 Standard Query Operators available in LINQ. In the above figure, we have used "where" operator (aka clause) followed by a condition. This condition is generally expressed using lambda expression.

LINQ query syntax always ends with a Select or Group clause. The Select clause is used to shape the data. You can select the whole object as it is or only some properties of it. In the above example, we selected the each resulted string elements.

In the following example, we use LINQ query syntax to find out teenager students from the Student collection (sequence).

Example: LINQ Query Syntax in C#

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


public class Program
{
    public static void Main()
    {
            // Student collection
        IList<Student> studentList = new List<Student>>() {
                new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
                new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,
                new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,
                new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
                new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
            };

        // LINQ Query Syntax to find out teenager students
        var teenAgerStudent = from s in studentList
                              where s.Age > 12 && s.Age < 20
                              select s;
        Console.WriteLine("Teen age Students:");

        foreach(Student std in teenAgerStudent){
            Console.WriteLine(std.StudentName);
        }
    }
}
public class Student{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }

}
```

Result:

```
Teen age Students:
John
Bill
Ron
```

LINQ Method Syntax

Method syntax (also known as fluent syntax) uses extension methods included in the **Enumerable** or **Queryable** static class, similar to how you would call the extension method of any class.

The following is a sample LINQ method syntax query that returns a collection of strings which contains a word "Tutorials".

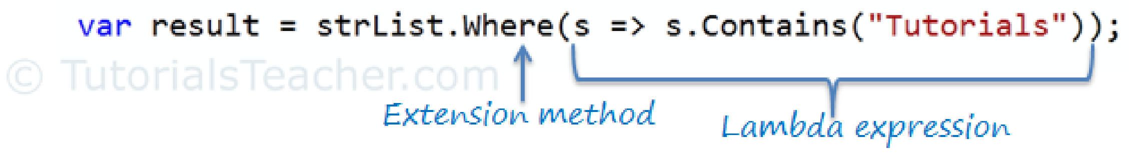
Example: LINQ Method Syntax in C#

```
public class Program
{
    public static void Main()
    {
        // string collection
        IList<string> stringList = new List<string>() {
            "C# Tutorials",
            "VB.NET Tutorials",
            "Learn C++",
            "MVC Tutorials" ,
            "Java"
        };

        // LINQ Query Syntax
        var result = stringList.Where(s => s.Contains("Tutorials"));


        foreach(string str) in result){
            Console.WriteLine(str);
        }
    }
}
```

Result:

```
C# Tutorials
VB.NET Tutorials
MVC Tutorials
```

The following figure illustrates the structure of LINQ method syntax.



As you can see in the above figure, method syntax comprises of extension methods and Lambda expression.

The extension method **Where()** is defined in the Enumerable class.

If you check the signature of the Where extension method, you will find the Where method accepts a predicate delegate as Func<Student, bool>.

This means you can pass any delegate function that accepts a Student object as an input parameter and returns a Boolean value. The lambda expression works as a delegate passed in the Where clause.

The following example shows how to use LINQ method syntax query with the IEnumerable<T> collection.

Example: Method Syntax in C#

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


public class Program
{
    public static void Main()
    {
        IList<Student> studentList = new List<Student>() {
                new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
                new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,
                new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,
                new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
                new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
            };

        // LINQ Method Syntax to find out teenager students
        var teenAgerStudents = studentList.Where(s => s.Age > 12 && s.Age < 20)
```

```
                                        .ToList<Student>();


        Console.WriteLine("Teen age Students:");

        foreach(Student std in teenAgerStudent){
            Console.WriteLine(std.StudentName);
        }
    }
}
public class Student{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }

}
```

Result:

```
Teen age Students:
John
Bill
Ron
```

Filtering Operators - Where

The Where operator (Linq extension method) filters the collection based on a given criteria expression and returns a new collection. The criteria can be specified as lambda expression or Func delegate type.

The following query sample uses a Where operator to filter the students who is teen ager from the given collection (sequence).

It uses a lambda expression as a predicate function.

Example: Where clause - LINQ query syntax C#

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



public class Program
{
    public static void Main()
    {
        // Student collection
        IList<Student> studentList = new List<Student>() {
                new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
                new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,
                new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,
                new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
                new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
            };

        // LINQ Query Syntax to find out teenager students
        var teenAgerStudent = from s in studentList
                              where s.Age > 12 && s.Age < 20
                              select s;
        Console.WriteLine("Teen age Students:");

        foreach(Student std in teenAgerStudent){
            Console.WriteLine(std.StudentName);
        }
    }
}

public class Student{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }

}
```

Result:

```
Teen age Students:
John
Bill
Ron
```

In the above sample query, the lambda expression body

`s.Age > 12 && s.Age < 20`

is passed as a predicate function `Func<TSource, bool>` that evaluates every student in the collection.

Alternatively, you can also use a Func type delegate with an anonymous method to pass as a predicate function as below (output would be the same):

Exapmle: Where clause

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


public class Program
{
    public static void Main()
    {
        // Student collection
        IList<Student> studentList = new List<Student>() {
                new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
                new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,
                new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,
                new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
                new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
            };

        Func<Student,bool> isTeenAger = delegate(Student s) {
            return s.Age > 12 && s.Age < 20;
        };

        var filteredResult = from s in studentList
                             where isTeenAger(s)
                             select s;

        foreach (var std in filteredResult)
            Console.WriteLine(std.StudentName);
    }
}

public class Student{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }

}
```

Result:

```
John
Bill
Ron
```

Where extension method in Method Syntax:

Unlike the query syntax, you need to pass whole lambda expression as a predicate function instead of just body expression in LINQ method syntax.

Example: Where in method syntax in C#

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

```
public class Program
{
    public static void Main()
    {
        // Student collection
        IList<Student> studentList = new List<Student>() {
                new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
                new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,
                new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,
                new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
                new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
            };

        // LINQ Query Method to find out teenager students
        var teenAgerStudent = studentList.Where(s => s.Age > 12 && s.Age < 20);

        Console.WriteLine("Teen age Students:");

        foreach(Student std in teenAgerStudent){
            Console.WriteLine(std.StudentName);
        }
    }
}

public class Student{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }

}
```

Result:

```
Teen age Students:
John
Bill
Ron
```

Projection Operators: Select

The Select operator always returns an IEnumerable collection which contains elements based on a transformation function.

It is similar to the Select clause of SQL that produces a flat result set.

Select in Query Syntax:

LINQ query syntax must end with a **Select** or **GroupBy** clause.

The following example demonstrates select operator that returns a string collection of StudentName.

Example: Select in query syntax C#

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


public class Program
{
    public static void Main()
    {
        // Student collection
        IList<Student> studentList = new List<Student>() {
                new Student() { StudentID = 1, StudentName = "John" } ,
                new Student() { StudentID = 2, StudentName = "Moin" } ,
                new Student() { StudentID = 3, StudentName = "Bill" } ,
                new Student() { StudentID = 4, StudentName = "Ram" } ,
                new Student() { StudentID = 5, StudentName = "Ron"  }
            };

        var selectResult = from s in studentList
                           select s.StudentName;

        foreach(var name in selectResult){
            Console.WriteLine(name);
        }
    }
}

public class Student{

    public int StudentID { get; set; }
    public string StudentName { get; set; }

}
```

Result:

```
John
Moin
Bill
Ram
Ron
```

The select operator can be used to formulate the result as per our requirement. It can be used to return a collection of custom class or anonymous type which includes properties as per our need.

The following example of the select clause returns a collection of anonymous type containing the Name and Age property.

Example: Select operator in query syntax C#

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


public class Program
{
    public static void Main()
    {
        IList<Student> studentList = new List<Student>() {
            new Student() { StudentID = 1, StudentName = "John", Age = 13 } ,
            new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,
            new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,
            new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
            new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
        };

        // returns collection of anonymous objects with Name and Age property
        var selectResult = from s in studentList
                           select new { Name = "Mr. " + s.StudentName, Age = s.Age };

        // iterate selectResult
        foreach (var item in selectResult)
            Console.WriteLine("Student Name: {0}, Age: {1}", item.Name, item.Age);
    }
}

public class Student{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }
}
```

Result:

```
Student Name: Mr. John, Age: 13
Student Name: Mr. Moin, Age: 21
Student Name: Mr. Bill, Age: 18
Student Name: Mr. Ram, Age: 20
Student Name: Mr. Ron, Age: 15
```

Select in Method Syntax:

The Select operator is optional in method syntax. However, you can use it to shape the data.

In the following example, Select extension method returns a collection of anonymous object with the Name and Age property:

Example: Select in method syntax C#

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


public class Program
{
    public static void Main()
    {
        IList<Student> studentList = new List<Student>() {
            new Student() { StudentID = 1, StudentName = "John", Age = 13 } ,
            new Student() { StudentID = 2, StudentName = "Moin",  Age = 21 } ,
            new Student() { StudentID = 3, StudentName = "Bill",  Age = 18 } ,
            new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
            new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
        };

        // returns collection of anonymous objects with Name and Age property
        var selectResult = studentList.Select(s => new { Name = s.StudentName ,
                                                      Age = s.Age  });

        // iterate selectResult
        foreach (var item in selectResult)
            Console.WriteLine("Student Name: {0}, Age: {1}", item.Name, item.Age);
    }
}

public class Student{

    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }
}
```

Result:

```
Student Name: John, Age: 13
Student Name: Moin, Age: 21
Student Name: Bill, Age: 18
Student Name: Ram, Age: 20
Student Name: Ron, Age: 15
```

In the above example, selectResult would contain anonymous objects with Name and Age property as shown below in the debug view.