

Unit 1.4: LINQ

Lecturer: Le Thi Bich Tra



Content

- › Lesson 1: What is LINQ?
- › Lesson 2: Writing LINQ Queries
- › Lesson 3: LinQ to SQL



Lesson 1: What is LINQ?

- § What is LINQ?
- § Benefits of using LINQ?
- § LINQ Architecture
- § LINQ Providers

What is LINQ

- › LINQ standards for Language Integrated Query.
- › LINQ is a structured query syntax built in C# and VB.NET used to save and retrieve data from different types of data sources like an Object Collection, SQL server database, XML, web service etc.

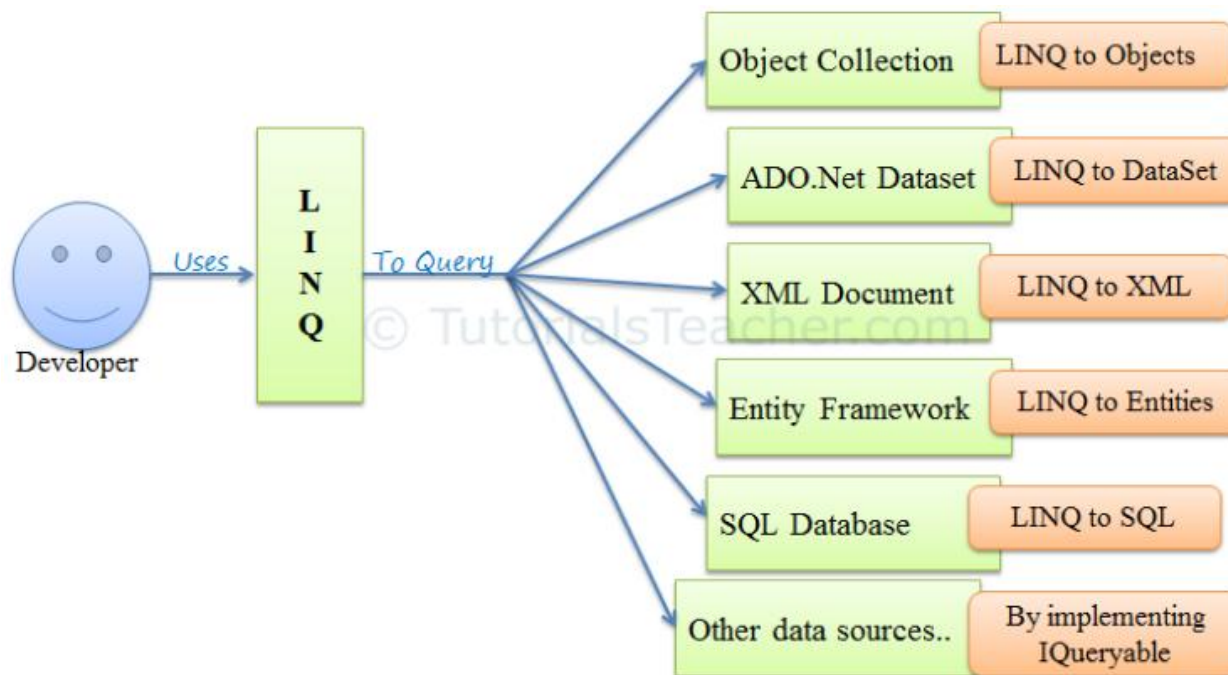
LINQ Architecture



- › LINQ query can be written using any .NET supported programming language
- › LINQ provider is a component between the LINQ query and the actual data source, which converts the LINQ query into a format that underlying data source can understand.

What is LINQ

› Why should we use LINQ



What is LINQ

› Benefits:

- 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.

Why use LINQ

```
static private string GetStringFromDb( SqlConnection sqlConnection, string sqlQuery)
{
    if (sqlConnection.State != ConnectionState.Open)
    {
        sqlConnection.Open();
    }
    SqlCommand sqlCommand = SqlCommand(sqlQuery, sqlConnection);
    SqlDataReader sqlDataReader = sqlCommand.ExecuteReader();
    string result = null;
    try
    {
        if (!sqlDataReader.Read())
        {
            throw (new Exception(
                String.Format("Unexpected exception executing query [{0}].", sqlQuery)));
        }
        else
        {
            {
                if (!sqlDataReader.IsDBNull(0))
                {
                    result = sqlDataReader.GetString(0);
                }
            }
        }
    }
    finally
    {
        // always call Close when done reading.
        sqlDataReader.Close();
    }
    return (result);    }}

```


Why use LINQ

```
static private void ExecuteStatementInDb(string cmd, string connection)
{
    SqlConnection sqlConn = new SqlConnection(connection);
    SqlCommand sqlComm = new SqlCommand(cmd);
    sqlComm.Connection = sqlConn;
    try
    {
        sqlConn.Open();
        Console.WriteLine("Executing SQL statement against database with ADO.NET ...");
        sqlComm.ExecuteNonQuery();
        Console.WriteLine("Database updated. ");
    }
    finally
    {
        // Close the connection.
        sqlComm.Connection.Close();
    }
}
```

Anonymous type

› Use var keyword we can create anonymous data types.

```
var st = "Hello";  
var x = 5;  
var intArray = new[] { 1, 2, 3 };
```

```
var s = new { Name = "Van A", Address = "HCM" };  
Console.WriteLine("Name= {0}, Address= {1}", s.Name, s.Address);
```

Extension methods

- › Extension Methods enable us to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.
- › Extension Methods are static methods that can be invoked using the instance method syntax.
- › Extension methods are contained in a static class and they have a this parameter of type class.

Extension methods

› Example:

```
public class Student
{
    public int StudentID { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
}

public static class MyExtension
{
    public static bool IsAdult(this Student st)
    {
        return st.Age >= 18;
    }
}
```

Lambda Expression

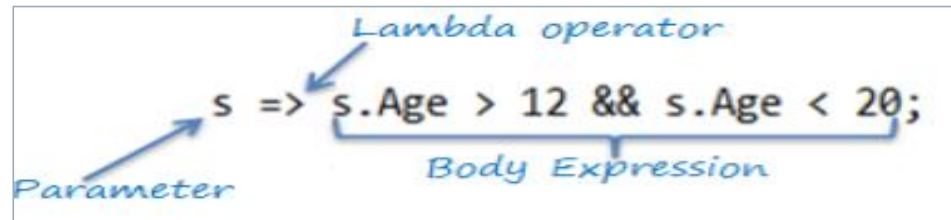
- › Lambda expression is actually a syntax change for anonymous methods. It is just a new way of writing anonymous methods.
- › Get rid of Delegate (old fashion)
- › Use LINQ fashion(Lambda express):
 - Lambda expressions are specified as a comma-delimited list of parameters followed by the lambda operator, followed by an expression or statement block.

`(param1, param2, ...paramN) => expression`

```
(param1, param2, ...paramN) => {  
    statement1;  
    return (lambda_expression_return_type);  
}
```

Lambda Expression

- › For example:



- › You can wrap the parameters in the parenthesis if you need to pass more than one parameter:

```
(s, youngAge) => s.Age >= youngage;
```

Or give type of each parameters if parameters are confusing:

```
(Student s, int youngAge) => s.Age >= youngage;
```

- › Statements block:

```
(s, youngAge) =>
{
    Console.WriteLine("Lambda expression with
multiple statements in the body");
    return s.Age >= youngAge;
}
```

Func Delegate

- › A lot of Linq operators take a Func<> parameter .
- › For example, check the Where operator:

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> source,  
    Func<T, bool> predicate);
```

- › This type allow to wrap a delegate function, in lambda expression form.

```
List<Product> listP = new List<Product>{ ...};  
Func<Product, bool> isValidProductID = p => p.ProductID > 10 && p.ProductID < 100;  
var validProducts = from p in listP  
                    where isValidProductID(p)  
                    select p;  
var validProducts2 = listP.Where(isValidProductID);  
//Or shorthand  
var validProducts3 = listP.Where(p => p.ProductID > 10 && p.ProductID < 100);
```



Lesson 2: Writing LINQ Queries

- § How to write LINQ Query?
- § Extension methods
- § Query Syntax and Method Syntax
- § Func delegate
- § Standard query operators

Writing LINQ Queries

- › To write LINQ queries, we use the LINQ Standard Query Operators: select, from, where, orderby, join, groupby...
- › There are 2 ways to write LINQ queries using these Standard Query Operators:
 - Using Lambda Expression.
 - Using Query Syntax or Method syntax (or fluent)
- › 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.

Writing LINQ Queries

› Example:

```
List<int> list = new List<int>() { 1, 2, -3, 4, 7, 8, 100, 20 };
List<int> listResult = new List<int>();
foreach (var num in list)
{
    if (num % 2 == 0)
    {
        listResult.Add(num);
    }
}
```

› With LINQ:

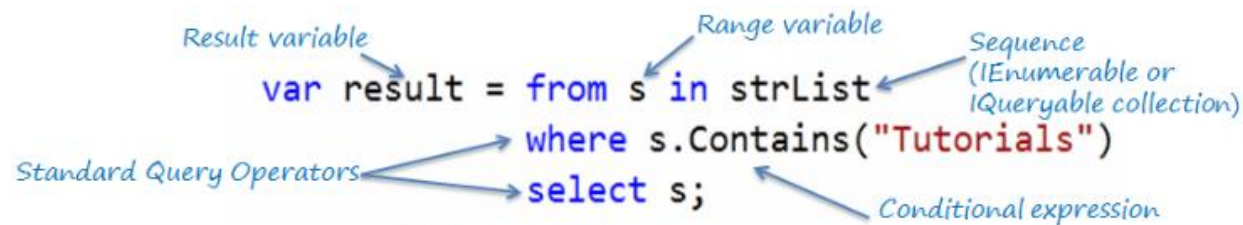
```
var listResult = from num in list
                  where num % 2 == 0
                  select num;
```

Query Syntax

- › Query syntax is similar to SQL (Structured Query Language) for the database. It is defined within the C# or VB code.
- › Syntax:

```
from <range variable> in <IEnumerable<T> or IQueryable<T> Collection>  
<Standard Query Operators> <lambda expression>  
<select or groupBy operator> <result formation>
```

- › Query Syntax starts with *from* clause and can be end with *Select* or *GroupBy* clause.



The diagram shows a LINQ query with annotations pointing to its components:

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

- Result variable**: Points to `var result`.
- Range variable**: Points to `s` in the `from` clause.
- Sequence (IEnumerable or IQueryable collection)**: Points to `strList`.
- Standard Query Operators**: Points to the `where` and `select` clauses.
- Conditional expression**: Points to `s.Contains("Tutorials")`.

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.

```
List<string> stringList = new List<string>() { "C# Tutorials", "VB. NET Tutorials", "Learn C++", "MVC Tutorials" , "Java" };
```

```
var result = strList.Where(s => s.Contains("Tutorials"));
```

Extension method

Lambda expression

Standard Query Operators

- › Standard Query Operators can be classified based on the functionality they provide.

Classification	Standard Query Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList

Filtering Operators

- › Filtering operators in LINQ filter the sequence (collection) based on some given criteria.

```
List<Student> studentList = new List<Student> {  
    new Student { StudentID = 1, StudentName = "John", Age = 18 } ,  
    new Student { StudentID = 2, StudentName = "Steve", Age = 15 } ,  
    new Student { StudentID = 3, StudentName = "Bill", Age = 25 } ,  
    new Student { StudentID = 4, StudentName = "Ram" , Age = 20 } ,  
    new Student { StudentID = 5, StudentName = "Ron" , Age = 19 }  
};
```

//using Query syntax

```
var filteredResult = from s in studentList  
                     where s.Age > 12 && s.Age < 20  
                     select s.StudentName;
```

//using Method syntax

```
var filteredResult = studentList.Where(s => s.Age > 12 && s.Age < 20);
```

Projection Operators

› Projection Operators:

- Select
- SelectMany

› They are used to transform the results of a query, allow us:

- to specify what properties we want to retrieve
- to perform calculations.

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

› Ex:

```
var products = from s in studentList  
                select new CustomStudent { Name = s.Name, Address = s.Address };
```

› Select clause can returns a collection of anonymous type

```
var selectResult = from s in studentList  
                    select new { Name = s.StudentName, Age = s.Age };
```

Projection Operators

- › Projection Operators:
 - Select
 - SelectMany
- › Loading related data: using navigation property or use the Join operators only if the tables do not have any navigational properties defined on them.

Sorting Operator

- › A sorting operator arranges the elements of the collection in ascending or descending order

//using Query syntax

```
var orderByResult = from s in studentList
                    orderby s.StudentName
                    select s;
```

//using Method syntax

```
var studentsInAscOrder = studentList.OrderBy(s => s.StudentName);
```

- › Multiple sorting:

- You can sort the collection on multiple fields separated by comma.

```
var orderByResult = from s in studentList
                    orderby s.StudentName, s.Age
                    select new { s.StudentName, s.Age };
```

Partitioning Operators

- › Partitioning operators split the sequence (collection) into two parts and returns one of the parts.
- › Partitioning Operators:
 - Take: returns the specified number of elements starting from the first element.
 - TakeWhile: returns elements from the given collection until the specified condition is true
 - Skip: skips the specified number of element starting from first element and returns rest of the elements.
 - SkipWhile:

```
List<string> strList = new List<string>() { "One", "Two",  
"Three", "Four", "Five" };  
var newList = strList.Take(2);  
var result = strList.TakeWhile(s => s.Length < 3);
```

Exercises

- › Exercise 1:
 - Paging using Skip & Take
 - Prompt the user to enter a page number. Once a valid page number is entered, the program should display the correct set of Students.
- › Exercise 2: Working with Max, Min, Sum, Average operators
- › Exercise 3: Working with Any, All operators.

GroupBy

- › This operator takes a flat sequence of items, organize that sequence into groups base on a specific key and return groups of sequences.

```
var employeeGroup = from e in Employee.getEmployees()
                    group e by e.DepartmentID;
foreach (var group in employeeGroup)
{
    Console.WriteLine("DepartmentID: {0}- #{1}",
                      group.Key, group.Count());
    foreach (var em in group)
    {
        Console.WriteLine(em.Name);
    }
}
```

Join

› Different type of joins in LINQ

- Group Join
- Inner Join
- Left Outer Join
- Cross Join

```
public class Department
{
    public int ID { get; set; }
    public string Name { get; set; }
    public static List<Department> getDepartments()
    {
        return new List<Department>() {
            new Department { ID=1, Name="IT"},
            new Department { ID=2, Name="HR"},
            new Department { ID=3, Name="Marketing"}
        };
    }
}

public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int DepartmentID { get; set; }
    public static List<Employee> getEmployees()
    {
        return new List<Employee>()
        {
            new Employee{ ID=1, Name= "A", DepartmentID=1},
            new Employee{ ID=2, Name= "D", DepartmentID=2},
            new Employee{ ID=3, Name= "C", DepartmentID=3},
            new Employee{ ID=4, Name= "B", DepartmentID=1},
            new Employee{ ID=5, Name= "E", DepartmentID=2}
        };
    }
}
```

Group join

- › Group Join: Each element from the first collection is paired with a set of correlated elements from the second collection.
- › Using Method syntax or Query syntax:
 - Query syntax is easy to write than Method syntax:

```
//using Query syntax: Use join operator and into keyword
var employeesByDepartment = from d in Department.getDepartments()
                             join e in Employee.getEmployees()
                             on d.ID equals e.DepartmentID into eGroup
                             select new {
                                 Department = d,
                                 Employee = eGroup
                             };
```

```
foreach (var department in employeesByDepartment)
{
    Console.WriteLine(department.Department.Name);
    foreach (var employee in department.Employee)
    {
        Console.WriteLine(" " + employee.Name);
    }
    Console.WriteLine();
}
```

Group join

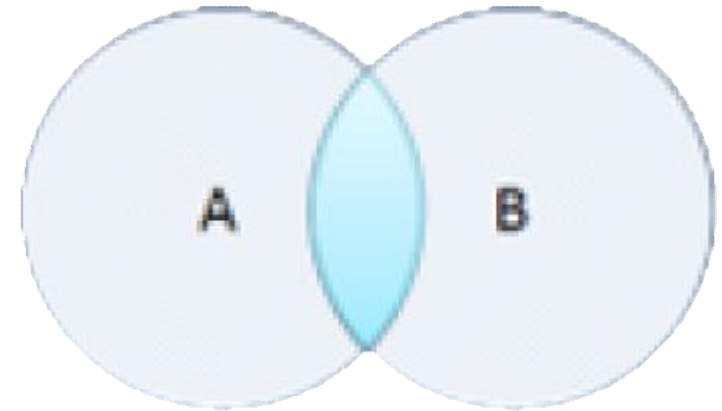
› Using Method syntax

```
var employeesByDepartment = Department.getDepartments()  
    . GroupJoin(Employee.getEmployees(),  
    d => d.ID,  
    e => e.DepartmentID,  
    (department, employees) => new  
    {  
        Department = department,  
        Employee = employees  
    });
```

- › Group Join is similar to Outer Join in SQL
- › Group join comes in handy when you want to create a hierarchical data structure.

Inner Join

- › If you have 2 collections, and when you perform an inner join, only the matching elements between the 2 collections are included in the result set.
- › Non- matching elements are excluded from the result set
- › Like relational database joins, joins can be performed on more than two sources.



Inner Join

> Example

//Using Query syntax

```
var result = from e in Employee.getEmployees()
              join d in Department.getDepartments()
              on e.DepartmentID equals d.ID
              select new
              {
                  EmployeeName = e.Name,
                  DepartmentName = d.Name
              };
```

//print result

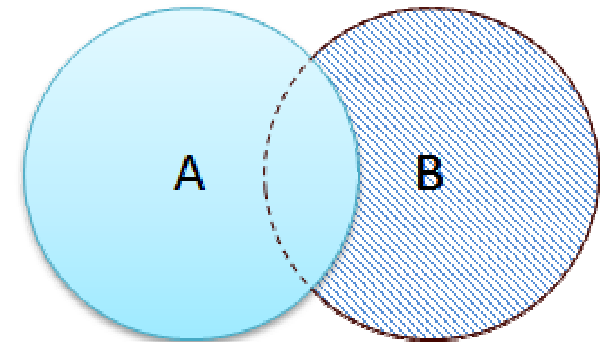
```
foreach (var employee in result)
{
    Console.WriteLine("{0}-{1}",
        employee.EmployeeName,
        employee.DepartmentName);
}
```

//using Method syntax

```
var result2 = Employee.getEmployees()
               .Join(Department.getDepartments(),
                   e => e.DepartmentID, d => d.ID,
                   (employee, department) => new
                   {
                       EmployeeName = employee.Name,
                       DepartmentName = department.Name
                   });
```

Left Outer Join

- › With Left outer Join, all the matching elements + all the non matching elements from the left collection are included in the result set.



```
var result = from e in Employee.getEmployees()
              join d in Department.getDepartments()
              on e.DepartmentID equals d.ID into eGroup
              from d in eGroup.DefaultIfEmpty()
              select new
              {
                  EmployeeName = e.Name,
                  DepartmentName = d == null ? "No Department" : d.Name
              };
foreach (var employee in result)
{
    Console.WriteLine("{0}-{1}", employee.EmployeeName, employee.DepartmentName);
}
```

Exercise

- › Working with ToArray & ToList operators

Thank You !