Language Integrated Query (LINQ)

Windows Programming Course

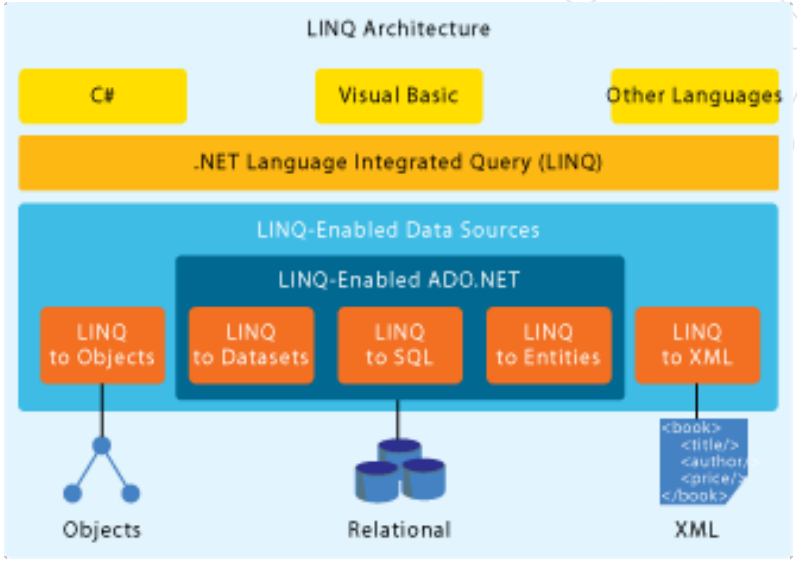
Agenda

1. LINQ Overview
2. LINQ Query
3. Standard Query Operators

1.
LINQ Overview

What is LINQ?

LINQ provides us common query syntax which allows us to query the data from various data sources such as: SQL Server database, XML documents, ADO.NET Datasets, and any other in-memory objects such as Collections, Generics, etc.



LINQ Providers

LINQ to Objects provider allows us to query an in-memory object such as an array, collection, and generics types. It provides many built-in functions that we can use to perform many useful operations such as filtering, ordering, and grouping with minimum code.

LINQ to XML provider is basically designed to work with an XML document. So, it allows us to perform different operations on XML data sources such as querying or reading, manipulating, modifying, and saving the changes to XML documents.

LINQ Providers (cont.)

LINQ to SQL Provider is designed to work with only the SQL Server database. You can consider this as an object-relational mapping (ORM) framework which allows one to one mapping between the SQL Server database and related .NET Classes.

LINQ to Entities looks like LINQ to SQL. It means it is also an object-relational mapping (ORM) framework that allows one to one, one to many, and many to many mapping between the database tables and .NET Classes. It supports several databases such as SQL Server, Oracle, MySQL, DB2, etc. (ADO.NET Entity Framework).

Advantages of using LINQ

- Don’t need to learn new query language syntaxes for different data source.
- Less code as compared to the traditional approach (SQL)
- Provides Compile time error checking as well as intelligence support in Visual Studio. Avoid runtime error.
- Provides a lot of built-in methods such as filtering, ordering, grouping, etc.

Disadvantages of using LINQ

- It’s very difficult to write complex queries like SQL.
- Doesn’t take the full advantage of SQL features.
- Get the worst performance if don’t write the queries properly.

2.
LINQ Syntax

Query Syntax
Method Syntax

Query Syntax

This is one of the easy ways to write complex LINQ queries in an easy and readable format.

The syntax for this type of query is very much similar to SQL Query.

Query Syntax starts with `from` clause and can be end with `select` or `groupby` clause.

Query Syntax (cont.)

```
List<Student> students = new List<Student>();

// Filtering and sorting the list with List methods
// Require the class Student derive from Icomparable
var result = students.FindAll(s => s.Name == "Alice");
result.Sort();

// LINQ Query
var query = from s in students
            where s.Name == "Alice"
            orderby s.StudentId descending
            select s;
```

Method Syntax

An **extension method** is defined as a static method whose first parameter defines the type it extends, and it is declared in a static class.

Extension methods make it possible to write a method to a class that doesn't already offer the method at first.

```
public static class StringExtension {
  public static void Foo(this string s) {
    Console.WriteLine($"Foo invoked for {s}"); }
}
var s = "Hello World";
s.Foo();
```

Method Syntax (cont.)

Method syntax is extension method in the namespace `System.Linq`.

It uses a lambda expression to define the condition for the query.

```
var result = students.Where(s => s.Name == "Alice")
                     .OrderBy(s => s.StudentId);
```

3.
Standard Query Operators

Filtering Operators

Filtering operators define a restriction to the elements returned:

- `Where`
- `OfType<TResult>`

Filtering Operators (cont.)

e.g.:

```
students.Where(s => s.Class == “18DTHQA1”);

students.Where((s, index) =>
      s.LastName.StartWith(“A”) && index % 2 != 0);

var dogs = animals.OfType<Dog>();
```

Projection Operators

Projection operators are used to transform an object into a new object of a different type.

- `Select`
- `SelectMany`

Projection Operators (cont.)

e.g.:

```
students.Where(s => s.Class == “18DTHQA1”)
        .Select(s => $“{s.FirstName} {s.LastName}”);

classes.Where(c => c.SchoolYear == 2017)
       .SelectMany(c => c.Students);
```

Sorting Operators

Sorting operators change the order of elements returned.

- `OrderBy`
- `ThenBy`
- `OrderByDescending`
- `ThenByDescending`
- `Reverse`

Sorting Operators

e.g.:

```
var result = students.OrderBy(s =>
s.Class.Name)
        .ThenBy(s => s.FirstName)
        .ThenBy(s => s.LastName);

result.Reverse();
```

Join/Grouping Operators

Join operators are used to combine collections that might not be directly related to each other.

- `Join` two collections based on key selector functions.
- `GroupJoin` joins two collections and groups the results.

Grouping operators put the data into groups.

- `GroupBy`
- `ToLookup` groups the elements by creating a one-to-many dictionary.

Join/Grouping Operators (cont.)

```
GroupBy<TSource,TKey,TElement,TResult>(
        Func<TSource,TKey> keySelector,
        Func<TSource,TElement> elementSelector,
        Func<TKey,IEnumerable<TElement>,TResult> resultSelector)
```

SQL:

```
Select Gender, Count(*)
From Students
Group by Gender
```

LINQ:

```
students.GroupBy(
  s => s.Gender,
  s => s,
  (k, v) => new { Gender = k,
                  Number = v.Count()
                }
);
```

Quantifier Operators

Quantifier operators return a Boolean value if elements of the sequence satisfy a specific condition.

- `Any` determines whether any element in the collection satisfies a predicate function.
- `All` determines whether all elements in the collection satisfy a predicate.
- `Contains` checks whether a specific element is in the collection.

Quantifier Operators (cont.)

e.g.:

```
var isAllMale = students.Any(s => s.Gender == “F”);

var isAllStudentHasExamResult =
  students.All(s => s.Exam.Any());
```

Partitioning Operators

Partitioning operators return a subset of the collection.

- `Take`
- `Skip` ignores the specified number of elements and takes the rest.
- `TakeWhile` takes the elements as long as a condition is true.
- `SkipWhile` skips the elements if the condition is true.

Partitioning Operators (cont.)

e.g.:

```
var pageSize = 10;
var page = 2; // Start from 0

var studentPage2 =
      students.Skip(page * pageSize)
              .Take(pageSize);
```

Set Operators

Set operators return a collection set.

- `Distinct`
- `Union` returns unique elements that appear in either of the two collections.
- `Intersect` returns elements that appear in both collections.
- `Except` returns elements that appear in just one collection.
- `Zip` combines two collections into one.

Set Operators (cont.)

e.g.:

```
var studentBirthPlace =
        students.Select(s => s.BirthPlace).Distinct();
```

Element Operators

Element operators return just one element.

- `First / FirstOrDefault`
- `Last / LastOrDefault`
- `ElementAt / ElementAtOrDefault`
- `Single / SingleOrDefault` returns only the one element that satisfies a condition. If more than one element satisfies the condition, an exception is thrown.

Aggregate Operators

Aggregate operators compute a single value from a collection.

- `Count`
- `Sum`
- `Min`
- `Max`
- `Average`

Conversion Operators

Conversion operators convert the collection to an array: IEnumerable, IList, IDictionary, and so on.

- `ToArray`
- `AsEnumerable`
- `ToList`
- `ToDictionary`
- `Cast<TResult>`

Generation Operators

Generation operators return a new sequence.

- `Empty`
- `Range`
- `Repeat`

e.g.:

```
var values = Enumerable.Range(1, 10);
// 1 2 3 4 5 6 7 8 9 10
```

Hands-on Exercise

Practice LINQ operators:

- Download the XML sample data on GitHub and Follow the example to load the sample data to memory
- Query data following conditions:
  - List all students of the class “18DTHQA1”, select FirstName, LastName and StudentId and sort by FirstName
  - Statistic student by gender/class/city
  - Calculate GPA of each student, e.g.: Nguyen A – GPA: 8.1 (round to 1 decimal)
  - Find student which has highest GPA
  - List students who must repeat subject: has a subject less than 5
  - For each subject, find student which has highest score
  - For each subject, statistic by average score/student has highest/lowest score
  - Find good students which GPA > 7 and no subject less than 5

```
class Student {
  Id, FirstName,
  LastName, Email,
  Gender, City,
  StudentId,
  List<Result> Exam;
}
class Result {
  Subject, Score
}
```

Thanks!

Any questions?

You can find me at:

tranminhphuoc@gmail.com