Krishna Nigalye

Software Developer

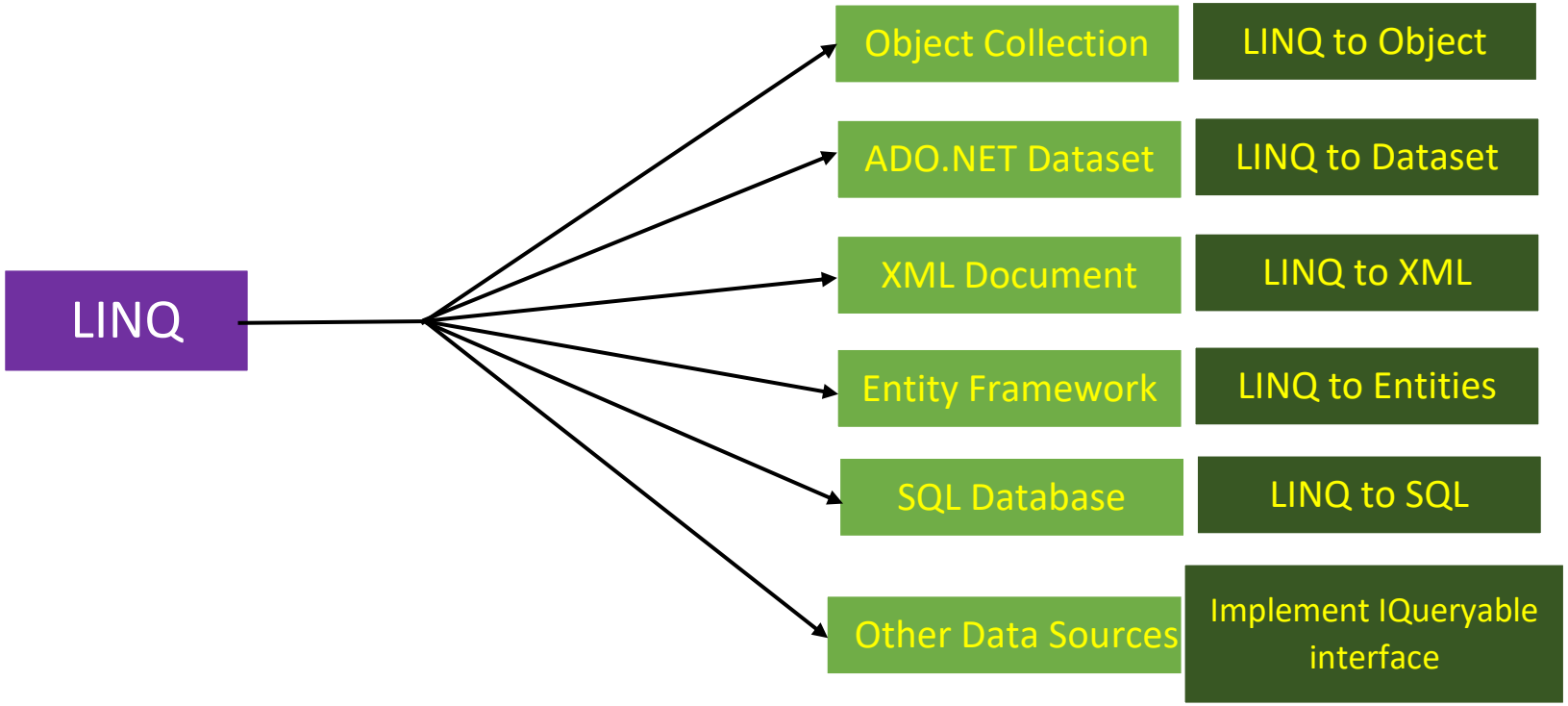
Learning C-Sharp

LANGUAGE INTEGRATED QUERY (LINQ)

WHAT IS LINQ?

- Language-Integrated Query (LINQ) is a set of features that extends powerful query capabilities directly into C#.
- The most important part to focus is **Query Expression** which are written in declarative query syntax.
- With the help of query syntax, you can perform operations like filtering, grouping and ordering on **any kind of data source** with minimum line of code.

- Same operations can be performed on data sources like ADO.NET datasets, SQL databases, XML documents, .NET Collections etc.



HOW LINQ WORKS?

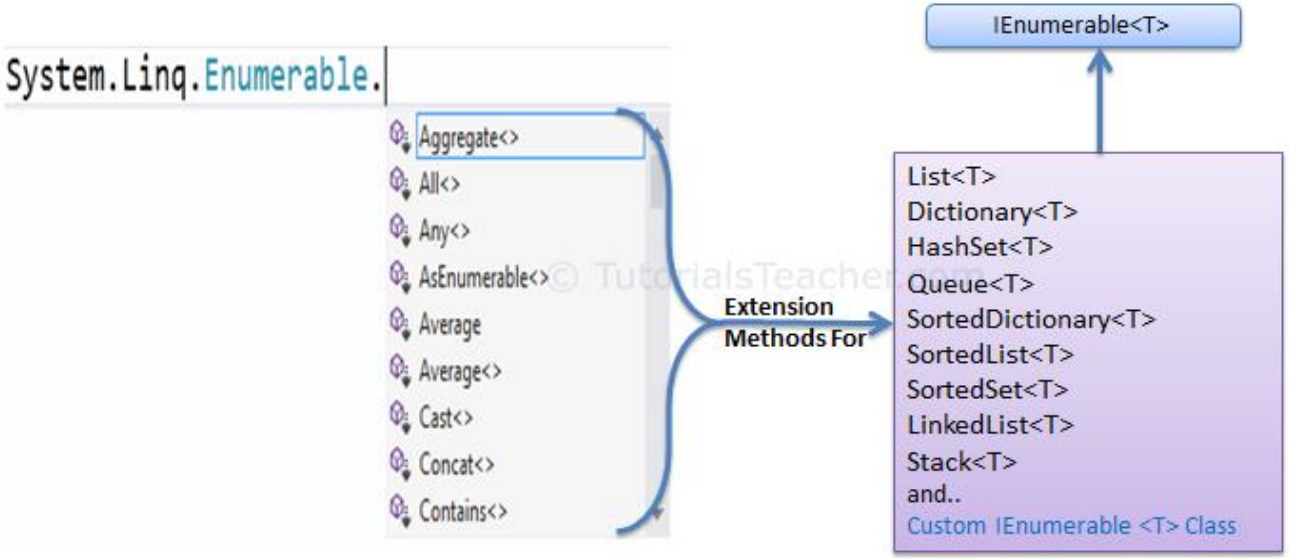
- We will not get result of the LINQ query until we execute it. LINQ query can be executed in many ways. For example, foreach loop. The foreach loop executes the query on the data source, gets the result and then it iterates over that result set. Thus, every LINQ query must query to some kind of data sources whether it can be array, collections, XML or any database. After writing LINQ query, it must be executed to get the result.

WHY WE SHOULD USE LINQ?

- LINQ code is easy, compact and readable.
- Query syntax is very relatable if you have used SQL.
- LINQ can be used in place of writing conditions in for loop or delegates.
- Same query syntax can be used to get data from multiple data sources.
- It provides type checking of objects at compile time.
- LINQ provides INTELLISENSE for generic collections.

LINQ API

- It includes two main classes. **Enumerable** & **Queryable**, both are static classes.
- The **Enumerable** class includes extension methods for classes that implements the **IEnumerable<T>** interface.
- All build-in collection classes such as List, Dictionary, SortedList, Queue, HashSet, LinkedList implements **IEnumerable<T>** interface. We can write LINQ queries to retrieve data from the built-in collections.

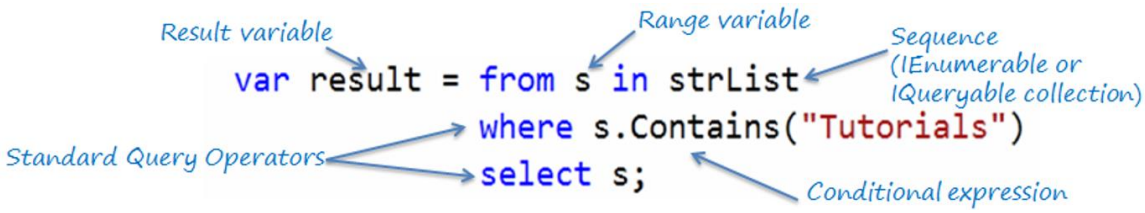


- The **Queryable** class includes extension methods for classes that implements the **IQueryable<T>** interface. The **IQueryable<T>** interface is used to provide querying capabilities against a specific data source where the type of the data is known.

HOW TO WRITE LINQ QUERIES?

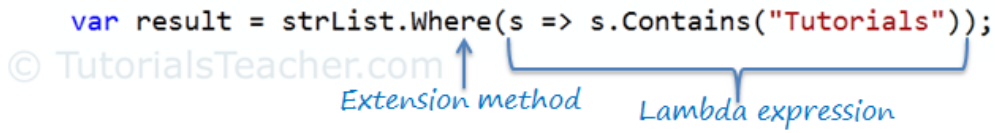
- There are two ways to write LINQ queries.
  - Query Syntax
  - Method Syntax or Fluent
- Query syntax:
  - Similar to SQL and defined within C#
  - Starts with 'from' keyword and ends with 'select' or 'groupBy' clause
  - Following is a general structure of a LINQ query

```
from <range variable> in <IEnumerable<T> or IQueryable<T> Collection>

<Query Operators like where> <lambda expression>

<select or groupBy operator> <result formation>
```



As you can see in the above example, sequence is nothing but your collection, 'select' is used to shape your data. In 'select', you can select the whole object or required parts of the object.

- Method syntax or Fluent syntax:
  - In this, we can use extension methods provided by **Enumerable** and **Queryable** classes.



  - You can call multiple extension functions within a single query.
- Standard query operators in query syntax are converted into extension methods at compile time. So, both are same.

For this demo, we are going to populate some dummy data.

```
public static class ProductStore
{
    public static IEnumerable<string> GetProductTypes()
    {
        return new string[]
        {
            "Electronics",
            "Books",
            "Beuty Products",
            "Fitness",
            "Support",
            "Software",
            "Furniture"
        };
    }

    public static IEnumerable<Product> GetProducts()
    {
        return new Product[]
        {
            new Product { Name = "TV", ProductType = "Electronics" },
            new Product { Name = "Laptop", ProductType = "Electronics" },
            new Product { Name = "E-book", ProductType = "Books" },
            new Product { Name = "Hardcopy", ProductType = "Books" },
            new Product { Name = "Kindle Book", ProductType = "Books" },
            new Product { Name = "Fairness Cream", ProductType = "Beuty Products" },
            new Product { Name = "Grooming Kit", ProductType = "Beuty Products" },
            new Product { Name = "Skin Care", ProductType = "Beuty Products" },
            new Product { Name = "Fragrances", ProductType = "Beuty Products" },
            new Product { Name = "Bath and Shower", ProductType = "Beuty Products" },
            new Product { Name = "Gaming Console", ProductType = "Software" },
            new Product { Name = "Antivirus", ProductType = "Software" },
            new Product { Name = "Video Games", ProductType = "Software" },
        };
    }
}
```

This is the sample data we are going to use now.

```
namespace c_sharp_linq
{
    partial class Program
    {
        static void Main(string[] args)
        {
            var productTypes = ProductStore.GetProductTypes();
            var products = ProductStore.GetProducts();

            ProductStore.ShowDummyData<string>(productTypes, "Product Types");
            ProductStore.ShowDummyData<Product>(products, "\nProducts");

            Console.ReadLine();
        }
    }
}
```

No issues found

e.cs

linq c_sharp_linq.ProductStore

```
        /// Generic Method to display data
        /// </summary>
        /// <typeparam name="T">Generic paramter</typeparam>
        /// <param name="data">List of items</param>
        /// <param name="header">head to display</param>
        public static void ShowDummyData<T>(IEnumerable<T> data, string header)
        {
            Console.WriteLine(header);
            Console.WriteLine("-------------");
            foreach (var item in data)
            {
                if (item is Product product)
                    Console.WriteLine(product.Name);
                else
                    Console.WriteLine(item);
            }
        }
    }
```

C:\Users\kpnigalye\Documents...

```
Product Types
-------------
Electronics
Books
Beuty Products
Fitness
Support
Software
Furniture

Products
-------------
TV
Laptop
E-book
Hardcopy
Kindle Book
Fairness Cream
Grooming Kit
Skin Care
Fragrances
Bath and Shower
Gaming Console
Antivirus
Video Games
```

WHERE OPERATOR

- This operator is used to filter data based on the given criteria.
- Let's see use of where clause.

```
/// <summary>
/// Using 'Where' clause
/// Select Products where ProductType is Software
/// </summary>
public static dynamic ExecuteSimpleQueries(in IEnumerable<Product> products)
{

    var result = from r in products
                 where r.ProductType is "Software"
                 select new { r.Name, Category = r.ProductType };
    return result;
}
```

```
C:\Users\kpnigalye\Documents\_programming\c-sharp-a...
Simple Queries
-------------

Software : Gaming Console
Software : Antivirus
Software : Video Games
```

```
/// <summary>
/// Using 'Where' clause
/// Select Products where ProductType is Software or Books
/// </summary>
public static dynamic ExecuteSimpleQueries(in IEnumerable<Product> products)
{

    var result = from r in products
                 where r.ProductType is "Software" || r.ProductType is "Books"
                 select new { r.Name, Category = r.ProductType };
    return result;
}
```

```
C:\Users\kpnigalye\Document...
-------------

Books : E-book
Books : Hardcopy
Books : Kindle Book
Software : Gaming Console
Software : Antivirus
Software : Video Games
```

OrderBy operator

- This operator helps you sort the result in ascending or descending order.

```
/// <summary>
/// Using 'OrderBy' clause
/// </summary>
public static dynamic ExecuteOrderByClause(in IEnumerable<Product> products)
{
    // Using Method Syntax
    //var result = products.OrderBy(a => a.ProductType);

    var result = from r in products
                 orderby r.ProductType
                 select new { r.Name, Category = r.ProductType };
    return result;
}

/// <summary>
/// Query using Combination of Where and OrderBy clauses
/// </summary>
public static dynamic ExecuteWhereAndOrderByClause(in IEnumerable<Product> products)
{
    // Using Method Syntax
    //var result = products.Where(p => p.ProductType == "Books").OrderByDescending(a => a.Name);

    var result = from r in products
                 where r.ProductType is "Books"
                 orderby r.Name descending
                 select new { r.Name, Category = r.ProductType };
    return result;
}
```

```
C:\Users\kpnigalye\Documen...

Simple Queries
-------------

Beuty Products : Fairness Cream
Beuty Products : Grooming Kit
Beuty Products : Skin Care
Beuty Products : Fragrances
Beuty Products : Bath and Shower
Books : E-book
Books : Hardcopy
Books : Kindle Book
Electronics : TV
Electronics : Laptop
Software : Gaming Console
Software : Antivirus
Software : Video Games

Simple Queries
-------------

Books : Kindle Book
Books : Hardcopy
Books : E-book
```

ThenBy and ThenByDescending operator

- This operator helps you sort on multiple fields either in ascending or descending order.

```
/// <summary>
/// Query using ThenBy clause
/// </summary>
public static dynamic ExecuteThenByClause(in IEnumerable<Product> products)
{
    // Using Method Syntax
    //var result = products.OrderBy(a => a.ProductType).ThenBy(b => b.Name);

    var result = from r in products
                 orderby r.ProductType, r.Name
                 select new { r.Name, Category = r.ProductType };
    return result;
}
```

```
C:\Users\kpnigalye\Do...
Beuty Products : Bath and Shower
Beuty Products : Fairness Cream
Beuty Products : Fragrances
Beuty Products : Grooming Kit
Beuty Products : Skin Care
Books : E-book
Books : Hardcopy
Books : Kindle Book
Electronics : Laptop
Electronics : TV
Software : Antivirus
Software : Gaming Console
Software : Video Games
```

```
/// <summary>
/// Query using ThenBy clause
/// </summary>
public static dynamic ExecuteThenByClause(in IEnumerable<Product> products)
{
    // Using Method Syntax
    //var result = products.OrderBy(a => a.ProductType).ThenByDescending(b => b.Name);

    var result = from r in products
                 orderby r.ProductType, r.Name descending
                 select new { r.Name, Category = r.ProductType };
    return result;
}
```

```
C:\Users\kpnigalye...
Beuty Products : Skin Care
Beuty Products : Grooming Kit
Beuty Products : Fragrances
Beuty Products : Fairness Cream
Beuty Products : Bath and Shower
Books : Kindle Book
Books : Hardcopy
Books : E-book
Electronics : TV
Electronics : Laptop
Software : Video Games
Software : Gaming Console
Software : Antivirus
```

GroupBy and ToLookUp operator

- This operator lets you group data according to the given parameter. Each group has a key based on which grouping is done.
- 'ToLookUp' is almost same as 'groupBy'. The difference between the two is query syntax does not support ToLookUp.
- Other important difference is in GroupBy execution is deferred and in ToLookUp execution is immediate.

```
/// <summary>
/// Query using GroupBy clause
/// </summary>
public static void ExecuteGroupByClause()
{
    IList<Student> studentList = GetStudentData();

    // Using Method Syntax
    //var groupedResult = studentList.GroupBy(a => a.Age);

    var groupedResult = from s in studentList
                        group s by s.Age;

    //iterate each group
    foreach (var ageGroup in groupedResult)
    {
        Console.WriteLine("\nAge Group: {0}", ageGroup.Key); //Each group has a key
        Console.WriteLine("----------------");
        foreach (Student s in ageGroup) // Each group has inner collection
            Console.WriteLine("Student Name: {0}", s.StudentName);
    }
}
```

```
C:\Users\kpnigalye...

Age Group: 18
----------------
Student Name: John
Student Name: Bill

Age Group: 21
----------------
Student Name: Steve
Student Name: Abram

Age Group: 20
----------------
Student Name: Ram
```

```
/// <summary>
/// Query using ToLookUp clause
/// </summary>
public static void ExecuteToLookUp()
{
    IList<Student> studentList = GetStudentData();

    // Using Method Syntax
    var groupedResult = studentList.ToLookup(a => a.Age);

    //iterate each group
    foreach (var ageGroup in groupedResult)
    {
        Console.WriteLine("\nAge Group: {0}", ageGroup.Key); //Each group has a key
        Console.WriteLine("----------------");
        foreach (Student s in ageGroup) // Each group has inner collection
            Console.WriteLine("Student Name: {0}", s.StudentName);
    }
}
```

```
C:\Users\kpnigalye\...

Age Group: 18
----------------
Student Name: John
Student Name: Bill

Age Group: 21
----------------
Student Name: Steve
Student Name: Abram

Age Group: 20
----------------
Student Name: Ram
```

So, GroupBy & ToLookup return a collection that has a key and an inner collection based on a key field value.

JOIN OPERATOR

- Join operators lets you perform operations on more than one collection.
- Join query syntax is as follows,

```
from... in outerSequence

join... in innerSequence

on  outerKey equals innerKey

select ...
```

```
/// <summary>
/// Inner Join
/// </summary>
public static dynamic ExecuteInnerJoin(in IEnumerable<string> productTypes, in IEnumerable<Product> products)
{
    // Using Method Syntax
    //var result = products.Join( // outer
    //    productTypes    // inner
    //    , product => product.ProductType    // inner key selector
    //    , productType => productType        // outer key selector
    //    , (product, productType) => new     // result formation
    //    {
    //        product.Name,
    //        Category = product.ProductType
    //    });

    var result = from r in productTypes
                 join p in products
                 on r equals p.ProductType
                 select new { p.Name, Category = p.ProductType };
    return result;
}
```

```
C:\Users\kpnigalye\Docu...
Inner Join
------------

Electronics : TV
Electronics : Laptop
Books : E-book
Books : Hardcopy
Books : Kindle Book
Beuty Products : Fairness Cream
Beuty Products : Grooming Kit
Beuty Products : Skin Care
Beuty Products : Fragrances
Beuty Products : Bath and Shower
Software : Gaming Console
```

GROUPJOIN OPERATOR

- GroupJoin operators in a same way as Join operator and in addition to that it lets you organize the result in groups based on a key.
- Group join allows you to group similar data.

```
/// <summary>
/// Group Join
/// Allows you to group data according to a param
/// </summary>
public static dynamic ExecuteGroupJoin(in IEnumerable<string> productTypes, in IEnumerable<Product> products)
{
    //var result = productTypes.GroupJoin(// outer
    //    products   // inner
    //    , productType => productType        // outer key selector
    //    , product => product.ProductType    // inner key selector
    //    , (productType, matchedProducts) => new     // result formation
    //    {
    //        Products = matchedProducts,
    //        Category = productType
    //    });
    var result = from r in productTypes
                 join p in products on r equals p.ProductType into pd
                 select new { Category = r, Products = pd };

    return result;
}
```

```
Console.WriteLine("Group Join");
Console.WriteLine("-------------\n");

foreach (var item in result)
{
    Console.WriteLine($"{item.Category}");
    foreach(var p in item.Products)
    {
        Console.WriteLine($"\t{p.Name}");
    }
}
```

```
Electronics
        TV
        Laptop
Books
        E-book
        Hardcopy
        Kindle Book
Beuty Products
        Fairness Cream
        Grooming Kit
        Skin Care
        Fragrances
        Bath and Shower
Fitness
Support
Software
        Gaming Console
        Antivirus
        Video Games
Furniture
```

LEFT JOIN OPERATOR

- Left Join operators in a same way as Left Join operator of SQL.
- The following example displays data using **Left Outer Join** on *ProductTypes* and *Products*. Left join allows you to get all the data from left table and matching columns in the right table. See the functions defined and output of the LINQ query.

```
public static dynamic ExecuteLeftJoin(in IEnumerable<string> productTypes, in IEnumerable<Product> products)
{
    var result = from r in productTypes
                 join p in products on r equals p.ProductType into pd
                 from p in pd.DefaultIfEmpty()
                 select new { Category = r, Name = p == null ? "No Product" : p.Name };
    return result;
}
```

OR

```
public static dynamic ExecuteRightJoin(in IEnumerable<string> productTypes, in IEnumerable<Product> products)
{
    var result = from r in productTypes
                 join p in products on r equals p.ProductType into pd
                 from p in pd.DefaultIfEmpty(new Product { Name = "No Product" })
                 select new { Category = r, p.Name };
    return result;
}
```

Output will be same in both the cases.

```
Electronics : TV
Electronics : Laptop
Books : E-book
Books : Hardcopy
Books : Kindle Book
Beuty Products : Fairness Cream
Beuty Products : Grooming Kit
Beuty Products : Skin Care
Beuty Products : Fragrances
Beuty Products : Bath and Shower
Fitness : No Product
Support : No Product
Software : Gaming Console
Software : Antivirus
Software : Video Games
Furniture : No Product
```

Select & SelectMany operator

- **Select** operator can be used to formulate the result as per our requirements. It can return the collection or anonymous types.
- A **Select** operator is used to select value from a collection and **SelectMany** operator is used for selecting values from a nested collection.
- Here is the list of students used for this demo.

```
C:\Users\k...
List of Students
-------------------

1: John
Skills:
        C++
        Java

2: Steve
Skills:
        Python
        C++
        Java

3: Bill
Skills:
        Python
        C#
        Java

4: Ram
Skills:
        Angular
        Python

5: Abram
Skills:
        C#
```

- The example shown below shows the difference between the two.
- **Select** can be used to show every record in the students Collection and **SelectMany** can be used directly to access only the list of skills of every student.

```
public static void ExecuteSelectClause()
{
    IList<Student> studentList = StudentData.GetStudentData();
    // Using Method Syntax
    //var result = studentList.Select(a => new { a.StudentName, a.Skills });

    var result = from s in studentList
                 select new { s.StudentName, s.Skills };

    Console.WriteLine("Select Clause");
    Console.WriteLine("----------------");

    foreach (var student in result)
    {
        Console.WriteLine(student.StudentName);
        foreach (var skill in student.Skills)
        {
            Console.WriteLine($"\t{skill}");
        }
    }
    Console.WriteLine();
}

/// <summary> Using 'SelectMany' clause
public static void ExecuteSelectManyClause()
{
    IList<Student> studentList = StudentData.GetStudentData();

    var skills = studentList.SelectMany(a => a.Skills);

    Console.WriteLine("SelectMany Clause");
    Console.WriteLine("----------------");

    foreach (var item in skills)
    {
        Console.WriteLine(item);
    }
}
```

```
C:\Users\kpnigalye\...
Select Clause
----------------
John
        C++
        Java
Steve
        Python
        C++
        Java
Bill
        Python
        C#
        Java
Ram
        Angular
        Python
Abram
        C#

SelectMany Clause
----------------
C++
Java
Python
C++
Java
Python
C#
Java
Angular
Python
C#
```

QUANTIFIER OPERATORS

- LINQ has three quantifier operators. They are ‘All’, ‘Any’ and ‘Contains’.
- They evaluate the elements of sequence based on some condition and returns a **Boolean** to indicate that some or all elements satisfy the condition.
- Important point to remember is, these operators are not supported by query syntax.
- As you can see, **all** the elements of the sequence satisfy the conditions so the result is true.

```
/// <summary>
/// Query using 'All' operator
/// </summary>
public static void ExecuteOperator_All()
{
    IList<Student> studentList = GetStudentData();

    // Using Method Syntax
    bool result = studentList.All(a => a.Age > 15 && a.Age < 22);

    Console.WriteLine("Result of All operator: {0}", result);
}
```

```
C:\Users\kpnigalye\D...
Result of All operator: True
```

- As you can see, there are **some** elements of the sequence that satisfy the conditions so the result is true.

```
/// <summary>
/// Query using 'Any' operator
/// </summary>
public static void ExecuteOperator_Any()
{
    IList<Student> studentList = GetStudentData();

    // Using Method Syntax
    bool result = studentList.Any(a => a.Age < 20);

    Console.WriteLine("Result of Any operator: {0}", result);
}
```

```
C:\Users\kpnigalye\Documents\_p...
Result of Any operator: True
```

- As you can see, all the elements of the sequence satisfy the conditions so the result is true.

Contains operators

- This is the one of the Quantifier operators in LINQ. It checks whether a specified element exists in the element or not.

```
public static void ExecuteOperator_ContainsUsingCollections()
{
    List<int> nums = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    Console.WriteLine($"Does list contains 0? {nums.Contains(0)}");
    Console.WriteLine($"Does list contains 5? {nums.Contains(5)}");
}
```

```
C:\Users\kpnigalye\Documents\_programming\c-sharp-advanced\co...
Does list contains 0? False
Does list contains 5? True
```

This is simple right? But this does not work in case of objects. Let us consider our list of products.

```
C:\Users\kpnigalye\Documents...
Product Types
------------
Electronics
Books
Beuty Products
Fitness
Support
Software
Furniture

Products
------------
TV
Laptop
E-book
Hardcopy
Kindle Book
Fairness Cream
Grooming Kit
Skin Care
Fragrances
Bath and Shower
Gaming Console
Antivirus
Video Games
```

```
public static void ExecuteOperator_Contains(in IEnumerable<Product> products)
{
    Product product = new Product { Name = "Laptop", ProductType = "Electronics" };

    Console.WriteLine($"Does Product List contains our product? {products.Contains(product)}");
}
```

```
C:\Users\kpnigalye\Documents\_programming\c-sharp-advanced\co...
Does Product List contains our product? False
```

This happens because **Contains** works well with Primitive types but not with custom classes. It checks only the references not the actual objects. Therefore, even if the data is same, it returns false.

To make it work, we have to define a custom class that implements '**IEqualityComparer'** interface. Why? Take a look at the second parameter of the overloaded method.

```
Console.WriteLine($"Does Product List contains our product? " +
    $"{products.Contains(product,)}");
```

▲ 2 of 2 ▼ (extension) bool IEnumerable<Product>.Contains<Product>(Product value, **IEqualityComparer<Product> comparer**)
Determines whether a sequence contains a specified element by using a specified IEqualityComparer<in T>.
comparer: *An equality comparer to compare values.*

dregion

This interface has **Equals** method to compare values of two objects.

```
class ProductComparer : IEqualityComparer<Product>
{
    public bool Equals(Product x, Product y)
    {
        if (x.Name == y.Name && x.ProductType == y.ProductType)
            return true;

        return false;
    }

    public int GetHashCode(Product obj)
    {
        return obj.GetHashCode();
    }
}
```

Now when you will execute your method again, you will see the expected output.

```
public static void ExecuteOperator_Contains(in IEnumerable<Product> products)
{
    Product product = new Product { Name = "Laptop", ProductType = "Electronics" };

    Console.WriteLine($"Does Product List contains our product? " +
        $"{products.Contains(product, new ProductComparer())}");
}
```

C:\Users\kpnigalye\Documents_programming\c-sharp-...

```
Does Product List contains our product? True
```

AGGREGATE OPERATORS

- Aggregate operators such as aggregate, sum, max, min, average are used to perform operations on numeric property of objects.
- They are not supported with query syntax in C#.

1. Take a look at the use of aggregate function. There are three different overloads.
    a. The following function concatenates list of strings into a string of comma separated values.

```
public static void ExecuteAggregateMethod()
{
    IList<string> electronics = new List<string> { "Phone", "Computer", "Laptop", "Kindle", "Smart TVs", "LED TVs" };

    Console.WriteLine("Aggregate Method");
    Console.WriteLine("------------------");

    Console.WriteLine(electronics.Aggregate((item1, item2) => item1 + ", " + item2));
}
```

```
C:\Users\kpnigalye\Documents\_programming\c-sharp-advanced\co...
Aggregate Method
------------------
Phone, Computer, Laptop, Kindle, Smart TVs, LED TVs
```

A suitable explanation of how this function works,

```
IList<String> strList = new List<String>() { "One", "Two", "Three", "Four", "Five" };

                                                    s2      s2      s2      s2
var commaSeparatedString = strList.Aggregate((s1, s2) => s1 + ", " + s2);

                                                s1 = s1 + ", " + s2;

                                               s1 = "One";
                                               s1 = "One" + ", " + "Two";
                                               s1 = "One, Two" + ", " + "Three";
                                               s1 = "One, Two, Three" + ", " +"Four";
                                               s1 = "One, Two, Three, Four" + ", " + "Five";
                                               s1 = "One, Two, Three, Four, Five";
```

b. The second overload of Aggregate function takes a seed value. It will work like this.

```
public static void ExecuteAggregateMethodWithSeed()
{
    IList<string> electronics = new List<string> { "Phone", "Computer", "Laptop", "Kindle", "Smart TVs", "LED TVs" };

    Console.WriteLine("Aggregate Method");
    Console.WriteLine("------------------");

    Console.WriteLine(electronics.Aggregate("Available Electronics: ", (seed, item2) => seed += item2 + ", "));
}
```

```
C:\Users\kpnigalye\Documents\_programming\c-sharp-advanced\code\c-sharp-linq\bin\De...
Aggregate Method
------------------
Available Electronics: Phone, Computer, Laptop, Kindle, Smart TVs, LED TVs,
```

Aggregate function can also be used for adding integer data.

```
public static void ExecuteAggregateMethodForAddition()
{
    IList<Student> students = StudentData.GetStudentData();

    Console.WriteLine("Aggregate Method");
    Console.WriteLine("------------------");

    Console.WriteLine("Total Age: {0}", students.Aggregate(0, (totalAge, student) => totalAge += student.Age));
}
```

```
C:\Users\kpnigalye\Documents\_programming\c-sharp-...
Aggregate Method
------------------
Total Age: 98
```

c. There is a third overload of Aggregate function which allows you to formulate your result.

```
public static void ExecuteAggregateMethodWithResult()
{
    IList<Student> students = StudentData.GetStudentData();

    string names = students.Aggregate("Names: ",
        (str, s) => str += s.Name + ",",
        str => str.Substring(0, str.Length - 1));

    Console.WriteLine("Aggregate Method");
    Console.WriteLine("------------------");

    Console.WriteLine(names);
}
```

```
C:\Users\kpnigalye\Documents\_programming\c...
Aggregate Method
------------------
Names: John,Steve,Bill,Ram,Abram
```

2. Average Functions are also not supported by query syntax. It is used to calculate average of the numeric items in the collections.

```
public static void ExecuteAverageMethod()
{
    IList<int> allAges = StudentData.GetStudentData().Select(a => a.Age).ToList();

    Console.WriteLine("Average Method");
    Console.WriteLine("------------------");
    Console.WriteLine("Average Age: {0}", allAges.Average());
}
```

```
C:\Users\kpnigalye\Docum...
Average Method
------------------
Average Age: 19.6
```

3. Count Method is used to count the number of elements in the collection or number of elements which satisfies the condition.

```
public static void ExecuteCountMethod()
{
    int count = StudentData.GetStudentData().Count();

    Console.WriteLine("Count Method");
    Console.WriteLine("------------------");
    Console.WriteLine("Number of Students: {0}", count);
}

public static void ExecuteCountMethodWithCondition()
{
    int count = StudentData.GetStudentData().Count(a => a.Age > 20);

    Console.WriteLine("Number of Students whose age is above 20: {0}", count);
}
```

```
C:\Users\kpnigalye\Documents\_programming\c-s...
Count Method
------------------
Number of Students: 5
Number of Students whose age is above 20: 2
```

4. Min and Max methods are used the following way.

```
public static void ExecuteMinMaxMethod()
{
    int maxAge = StudentData.GetStudentData().Max(a=>a.Age);
    int minAge = StudentData.GetStudentData().Min(a=>a.Age);

    Console.WriteLine("Use of Min-Max Methods");
    Console.WriteLine("------------------");
    Console.WriteLine("Min Age: {0}", minAge);
    Console.WriteLine("Max Age: {0}", maxAge);
}
```

```
C:\Users\kpnigalye\Documents\_prog...
Use of Min-Max Methods
------------------
Min Age: 18
Max Age: 21
```

```
// Sum of the ages of students where age is a even number
public static void ExecuteSumMethod()
{
    int sum = StudentData.GetStudentData().Sum(a =>
    {
        if (a.Age % 2 == 0)
            return a.Age;
        else
            return 0;
    });

    Console.WriteLine("Use of Sum Method with a condition");
    Console.WriteLine("------------------------------------");
    Console.WriteLine("Sum of Even Ages: {0}", sum);
}
```

```
Select C:\Users\kpnigalye\Documents\_prog...
Use of Sum Method with a condition
------------------------------------
Sum of Even Ages: 56
```

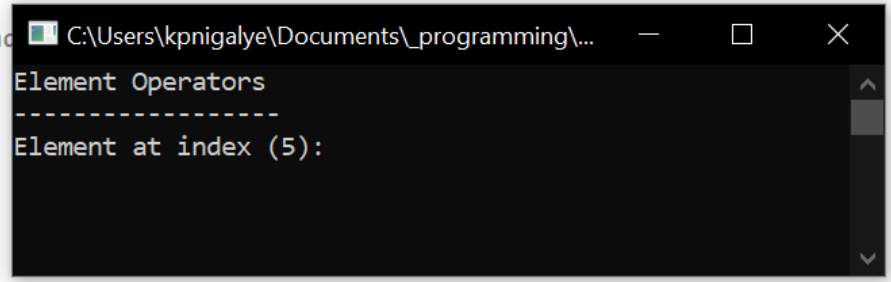
5. The following code shows use of Sum method using a condition.

ELEMENT OPERATORS

Element operators are used to get elements from a collection.

1. The ElementAt() method returns an element from the specified index from a given collection. If the specified index is out of the range of a collection then it will throw an Index out of range exception.
   The ElementAtOrDefault() method also returns an element from the specified index from a collection and if the specified index is out of range of a collection then it will **return a default value of the data type** instead of throwing an error.

```
public static void ExecuteElementAt(int index)
{
    IList<string> electronics = new List<string> { "Phone", "Computer", "Laptop", "Kindle", "Smart TVs", "LED TVs" };

    Console.WriteLine("Element Operators");
    Console.WriteLine("------------------");

    // electronics.ElementAt(index) will throw an error if index is out of the range of collection
    // electronics.ElementAtOrDefault(index) will return null if element at specified index is out of range
    Console.WriteLine("Element at index (5): {0}", electronics.ElementAtOrDefault(index));
}
#enc
```



2. The First() method returns the first element of a collection, or the first element that satisfies the specified condition. If a given or result collection is empty then it will throw an exception.
   The FirstOrDefault() method does the same thing as First() method. The only difference is that it returns default value of the data type of a collection if a collection is empty or doesn't find any element that satisfies the condition.

```
public static void ExecuteFirstOp()
{
    IList<int> numbers = new List<int> { 2, 4, 6, 8, 9, 3 };
    IList<int> emptyList = new List<int>();

    Console.WriteLine("Element Operators");
    Console.WriteLine("------------------");

    try
    {
        // First() will throw an error if collection is empty
        Console.WriteLine("First Element of emptyList: {0}", emptyList.First());
    }
    catch(Exception ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine("\nFirst Element of emptyList: {0}", emptyList.FirstOrDefault());
    }

    // FirstOrDefault() will return the default value of the type if result collection is empty
    Console.WriteLine("First Element of numbers: {0}", numbers.FirstOrDefault());
}
```

```
C:\Users\kpnigalye\Documents\_programming\c-sha...
Element Operators
------------------
Sequence contains no elements

First Element of emptyList: 0
First Element of numbers: 2
```

3. The Last() method returns the last element of a collection, or the last element that satisfies the specified condition. If a given or result collection is empty then it will throw an exception.
   The LastOrDefault() method does the same thing as Last() method. The only difference is that it returns default value of the data type of a collection if a collection is empty or doesn't find any element that satisfies the condition.

```
public static void ExecuteLastOp()
{
    IList<int> numbers = new List<int> { 2, 4, 6, 8, 9, 3 };
    IList<int> emptyList = new List<int>();

    Console.WriteLine("Element Operators");
    Console.WriteLine("------------------");

    try
    {
        // Last() will throw an error if collection is empty
        Console.WriteLine("Last Element of emptyList: {0}", emptyList.Last(i => i % 2 == 0));
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine("\nLast Element of emptyList: {0}", emptyList.LastOrDefault());
    }

    // LastOrDefault() will return the default value of the type if result collection is empty
    Console.WriteLine("Last Element of numbers: {0}", numbers.LastOrDefault(i => i % 2 == 0));
}
#en
```

```
C:\Users\kpnigalye\Documents\_programming\...
Element Operators
------------------
Sequence contains no matching element

Last Element of emptyList: 0
Last Element of numbers: 8
```

4. Single() returns the only element from a collection, or the only element that satisfies the specified condition. If a given collection includes no elements or more than one elements then Single() throws InvalidOperationException.

```
public static void ExecuteSingleOp()
{
    IList<int> numbers = new List<int> { 2, 4, 6, 8, 9, 3 };
    IList<int> emptyList = new List<int>();

    Console.WriteLine("Element Operators");
    Console.WriteLine("------------------");

    try
    {
        // Single() will throw an error if collection is empty
        // or more than one elements are present which satisfies the condition
        Console.WriteLine("Only Element of emptyList: {0}", emptyList.Single(i => i % 2 == 0));
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        Console.WriteLine("\nOnly Element of emptyList (Prints defaut value): {0}", emptyList.SingleOrDefault());
    }
}
#end
```

```
C:\Users\kpnigalye\Documents\_programming\c-sharp-advanced\code\c-sha...
Element Operators
------------------
Sequence contains no matching element

Only Element of emptyList (Prints defaut value): 0
```

The SingleOrDefault() method does the same thing as Single() method. The only difference is that it returns default value of the data type of a collection if a collection is empty or there are no elements that match the condition.

If the collection contains more than one element that satisfies the condition then it throws an exception.

```
public static void ExecuteSingleOrDefaultOp()
{
    IList<int> numbers = new List<int> { 2, 4, 6, 8, 9, 3 };
    IList<int> emptyList = new List<int>();

    Console.WriteLine("Element Operators");
    Console.WriteLine("------------------");

    try
    {
        // SingleOrDefault() will return the default value of the type if result collection contains no elements
        // that satisfies the condition
        Console.WriteLine("Only Element of numbers: {0}", numbers.SingleOrDefault(i => i > 10));

        // SingleOrDefault() will throw an exception if result collection contains more than one element
        // that satisfies the condition
        Console.WriteLine("Only Element of numbers(throws an exception): {0}", numbers.SingleOrDefault(i => i % 2 == 0));

    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

#end
```

```
C:\Users\kpnigalye\Documents\_programming\c-sharp-adv...
Element Operators
------------------
Only Element of numbers: 0
Sequence contains more than one matching element
```