LEARNING linq

#linq

Table of Contents

About

Chapter 1: Getting started with linq

Remarks

LINQ is a set of features introduced in the .NET Framework version 3.5 that bridges the gap between the world of objects and the world of data.

Traditionally, queries against data are expressed as simple strings without type checking at compile time or IntelliSense support. Furthermore, you have to learn a different query language for each type of data source: SQL databases, XML documents, various Web services, and so on. LINQ makes a query a first-class language construct in C# and Visual Basic. You write queries against strongly typed collections of objects by using language keywords and familiar operators.

Examples

Setup

LINQ requires .NET 3.5 or higher (or .NET 2.0 using LINQBridge).

Add a reference to **System.Core**, if it hasn't been added yet.

At the top of the file, import the namespace:

- C#

```
using System;
using System.Linq;
```

- VB.NET

```
Imports System.Linq
```

The different joins in LINQ

In the following examples, we'll be using the following samples:

```
List<Product> Products = new List<Product>()
{
  new Product()
  {
    ProductId = 1,
    Name = "Book nr 1",
    Price = 25
  },
  new Product()
  {
    ProductId = 2,
    Name = "Book nr 2",
    Price = 15
  },
  new Product()
  {
    ProductId = 3,
    Name = "Book nr 3",
    Price = 20
  },
};
List<Order> Orders = new List<Order>()
{
  new Order()
  {
    OrderId = 1,
    ProductId = 1,
  },
  new Order()
  {
    OrderId = 2,
    ProductId = 1,
  },
  new Order()
  {
    OrderId = 3,
    ProductId = 2,
  },
  new Order()
  {
    OrderId = 4,
    ProductId = NULL,
  },
};
```

INNER JOIN

Query Syntax

```
var joined = (from p in Products
              join o in Orders on p.ProductId equals o.ProductId
              select new
              {
                o.OrderId,
                p.ProductId,
                p.Name
              }).ToList();
```

Method Syntax

```
var joined = Products.Join(Orders, p => p.ProductId,
                                   o => o.OrderId,
                                     => new
                                    {
                                      OrderId   = o.OrderId,
                                      ProductId = p.ProductId,
                                      Name      = p.Name
                                    })
                     .ToList();
```

Result:

```
{ 1, 1, "Book nr 1" },
{ 2, 1, "Book nr 1" },
{ 3, 2, "Book nr 2" }
```

LEFT OUTER JOIN

```
var joined = (from p in Products
              join o in Orders on p.ProductId equals o.ProductId into g
              from lj in g.DefaultIfEmpty()
              select new
              {
                //For the empty records in lj, OrderId would be NULL
                OrderId = (int?)lj.OrderId,
                p.ProductId,
                p.Name
              }).ToList();
```

Result:

```
{ 1, 1, "Book nr 1" },
{ 2, 1, "Book nr 1" },
{ 3, 2, "Book nr 2" },
{ NULL, 3, "Book nr 3" }
```

CROSS JOIN

```
var joined = (from p in Products
              from o in Orders
              select new
              {
                o.OrderId,
                p.ProductId,
                p.Name
              }).ToList();
```

Result:

```
{ 1, 1, "Book nr 1" },
{ 2, 1, "Book nr 1" },
{ 3, 2, "Book nr 2" },
{ NULL, 3, "Book nr 3" },
{ 4, NULL, NULL }
```

GROUP JOIN

```
var joined = (from p in Products
              join o in Orders on p.ProductId equals o.ProductId
                into t
              select new
              {
                p.ProductId,
                p.Name,
                Orders = t
              }).ToList();
```

The Propertie `Orders` now contains an `IEnumerable<Order>` with all linked Orders.

Result:

```
{ 1, "Book nr 1", Orders = { 1, 2 } },
{ 2, "Book nr 2", Orders = { 3 } },
{ 3, "Book nr 3", Orders = { } },
```

How to join on multiple conditions

When joining on a single condition, you can use:

```
join o in Orders
  on p.ProductId equals o.ProductId
```

When joining on multiple, use:

```
join o in Orders
  on new { p.ProductId, p.CategoryId } equals new { o.ProductId, o.CategoryId }
```

Make sure that both anonymous objects have the same properties, and in VB.NET, they must be marked `Key`, although VB.NET allows multiple `Equals` clauses separated by `And`:

```
Join o In Orders
  On p.ProductId Equals o.ProductId And p.CategoryId Equals o.CategoryId
```

Query Syntax and Method Syntax

Query syntax and method syntax are semantically identical, but many people find query syntax simpler and easier to read. Let’s say we need to retrieve all even items ordered in ascending order from a collection of numbers.

C#:

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

// Query syntax:
IEnumerable<int> numQuery1 =
            from num in numbers
            where num % 2 == 0
            orderby num
            select num;

// Method syntax:
IEnumerable<int> numQuery2 = numbers.Where(num => num % 2 == 0).OrderBy(n => n);
```

VB.NET:

```
Dim numbers() As Integer = { 0, 1, 2, 3, 4, 5, 6 }

' Query syntax: '
Dim numQuery1 = From num In numbers
                 Where num Mod 2 = 0
                 Select num
                 Order By num

' Method syntax: '
Dim numQuery2 = numbers.where(Function(num) num Mod 2 = 0).OrderBy(Function(num) num)
```

Remember that some queries **must** be expressed as method calls. For example, you must use a method call to express a query that retrieves the number of elements that match a specified condition. You also must use a method call for a query that retrieves the element that has the maximum value in a source sequence. So that might be an advantage of using method syntax to make the code more consistent. However, of course you can always apply the method after a query syntax call:

C#:

```
int maxNum =
    (from num in numbers
     where num % 2 == 0
     select num).Max();
```

VB.NET:

```
Dim maxNum =
    (From num In numbers
     Where num Mod 2 = 0
     Select num).Max();
```

LINQ methods, and IEnumerable vs IQueryable

LINQ extension methods on `IEnumerable<T>` take actual methods[1], whether anonymous methods:

```
//C#
Func<int,bool> fn = x => x > 3;
var list = new List<int>() {1,2,3,4,5,6};
var query = list.Where(fn);

'VB.NET
Dim fn = Function(x As Integer) x > 3
Dim list = New List From {1,2,3,4,5,6};
Dim query = list.Where(fn);
```

or named methods (methods explicitly defined as part of a class):

```
//C#
class Program {
    bool LessThan4(int x) {
        return x < 4;
    }
    void Main() {
        var list = new List<int>() {1,2,3,4,5,6};
        var query = list.Where(LessThan4);
    }
}

'VB.NET
Class Program
    Function LessThan4(x As Integer) As Boolean
        Return x < 4
    End Function
    Sub Main
        Dim list = New List From {1,2,3,4,5,6};
        Dim query = list.Where(AddressOf LessThan4)
    End Sub
End Class
```

In theory, it is possible to parse the method's IL, figure out what the method is trying to do, and apply that method's logic to any underlying data source, not just objects in memory. But parsing IL is not for the faint of heart.

Fortunately, .NET provides the `IQueryable<T>` interface, and the extension methods at `System.Linq.Queryable`, for this scenario. These extension methods take an expression tree — a data structure representing code — instead of an actual method, which the LINQ provider can then parse[2] and convert to a more appropriate form for querying the underlying data source. For example:

```
//C#
IQueryable<Person> qry = PersonsSet();

// Since we're using a variable of type Expression<Func<Person,bool>>, the compiler
// generates an expression tree representing this code
Expression<Func<Person,bool>> expr = x => x.LastName.StartsWith("A");
// The same thing happens when we write the lambda expression directly in the call to
// Queryable.Where

qry = qry.Where(expr);


'VB.NET
Dim qry As IQueryable(Of Person) = PersonSet()

' Since we're using a variable of type Expression(Of Func(Of Person,Boolean)), the compiler
' generates an expression tree representing this code
Dim expr As Expression(Of Func(Of Person, Boolean)) = Function(x) x.LastName.StartsWith("A")
' The same thing happens when we write the lambda expression directly in the call to
' Queryable.Where

qry = qry.Where(expr)
```

If (for example) this query is against a SQL database, the provider could convert this expression to the following SQL statement:

```
SELECT *
```

```
FROM Persons
WHERE LastName LIKE N'A%'
```

and execute it against the data source.

On the other hand, if the query is against a REST API, the provider could convert the same expression to an API call:

```
http://www.example.com/person?filtervalue=A&filtertype=startswith&fieldname=lastname
```

There are two primary benefits in tailoring a data request based on an expression (as opposed to loading the entire collection into memory and querying locally):

- The underlying data source can often query more efficiently. For example, there may very well be an index on `LastName`. Loading the objects into local memory and querying in-memory loses that efficiency.
- The data can be shaped and reduced before it is transferred. In this case, the database / web service only needs to return the matching data, as opposed to the entire set of Persons available from the data source.

Notes

1. Technically, they don't actually take methods, but rather delegate instances which point to methods. However, this distinction is irrelevant here.
2. This is the reason for errors like "*LINQ to Entities does not recognize the method 'System.String ToString()' method, and this method cannot be translated into a store expression.*". The LINQ provider (in this case the Entity Framework provider) doesn't know how to parse and translate a call to `ToString` to equivalent SQL.

Read Getting started with linq online: https://riptutorial.com/linq/topic/842/getting-started-with-linq

Chapter 2: Linq Using Take while And Skip While

Introduction

Take, Skip, TakeWhile and SkipWhile are all called Partitioning Operators since they obtain a section of an input sequence as determined by a given condition. Let us discuss these operators

Examples

Take method

The Take Method Takes elements up to a specified position starting from the first element in a sequence. **Signature of Take:**

```
Public static IEnumerable<TSource> Take<TSource>(this IEnumerable<TSource> source,int count);
```

Example:

```
int[] numbers = { 1, 5, 8, 4, 9, 3, 6, 7, 2, 0 };
var TakeFirstFiveElement = numbers.Take(5);
```

Output:

The Result is 1,5,8,4 and 9 To Get Five Element.

Skip Method

Skips elements up to a specified position starting from the first element in a sequence.

Signature of Skip:

```
Public static IEnumerable Skip(this IEnumerable source,int count);
```

Example

```
int[] numbers = { 1, 5, 8, 4, 9, 3, 6, 7, 2, 0 };
var SkipFirstFiveElement = numbers.Take(5);
```

Output: The Result is 3,6,7,2 and 0 To Get The Element.

TakeWhile():

Returns elements from the given collection until the specified condition is true. If the first element itself doesn't satisfy the condition then returns an empty collection.

Signature of TakeWhile():

```
Public static IEnumerable <TSource> TakeWhile<TSource>(this IEnumerable <TSource>
source,Func<TSource,bool>,predicate);
```

Another Over Load Signature:

```
Public static IEnumerable <TSource> TakeWhile<TSource>(this IEnumerable <TSource>
source,Func<TSource,int,bool>,predicate);
```

Example I:

```
int[] numbers = { 1, 5, 8, 4, 9, 3, 6, 7, 2, 0 };
var SkipFirstFiveElement = numbers.TakeWhile(n => n < 9);
```

Output:

It Will return Of eleament 1,5,8 and 4

Example II :

```
int[] numbers = { 1, 2, 3, 4, 9, 3, 6, 7, 2, 0 };
var SkipFirstFiveElement = numbers.TakeWhile((n,Index) => n < index);
```

Output:

It Will return Of element 1,2,3 and 4

SkipWhile()

Skips elements based on a condition until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition, it then skips 0 elements and returns all the elements in the sequence.

Signature of SkipWhile():

```
Public static IEnumerable <TSource> SkipWhile<TSource>(this IEnumerable <TSource>
source,Func<TSource,bool>,predicate);
```

Another Over Load Signature:

```
Public static IEnumerable <TSource> SkipWhile<TSource>(this IEnumerable <TSource>
source,Func<TSource,int,bool>,predicate);
```

Example I:

```
int[] numbers = { 1, 5, 8, 4, 9, 3, 6, 7, 2, 0 };
```

```
var SkipFirstFiveElement = numbers.SkipWhile(n => n < 9);
```

Output:

It Will return Of element 9,3,6,7,2 and 0.

Example II:

```
int[] numbers = { 4, 5, 8, 1, 9, 3, 6, 7, 2, 0 };
var indexed = numbers.SkipWhile((n, index) => n > index);
```

Output:

It Will return Of element 1,9,3,6,7,2 and 0.

Read Linq Using Take while And Skip While online: https://riptutorial.com/linq/topic/10810/linq-using-take-while-and--skip-while

Chapter 3: Method execution modes - immediate, deferred streaming, deferred non-streaming

Examples

Deferred execution vs immediate execution

Some LINQ methods return a query object. This object does not hold the results of the query; instead, it has all the information needed to generate those results:

```
var list = new List<int>() {1, 2, 3, 4, 5};
var query = list.Select(x => {
    Console.Write($"{x} ");
    return x;
});
```

The query contains a call to `Console.Write`, but nothing has been output to the console. This is because the query hasn't been executed yet, and thus the function passed to `Select` has never been evaluated. This is known as **deferred execution** -- the query's execution is delayed until some later point.

Other LINQ methods force an **immediate execution** of the query; these methods execute the query and generate its values:

```
var newList = query.ToList();
```

At this point, the function passed into `Select` will be evaluated for each value in the original list, and the following will be output to the console:

```
1 2 3 4 5
```

Generally, LINQ methods which return a single value (such as `Max` or `Count`), or which return an object that actually holds the values (such as `ToList` or `ToDictionary`) execute immediately.

Methods which return an `IEnumerable<T>` or `IQueryable<T>` are returning the query object, and allow deferring the execution until a later point.

Whether a particular LINQ method forces a query to execute immediately or not, can be found at MSDN -- C#, or VB.NET.

Streaming mode (lazy evaluation) vs non-streaming mode (eager evaluation)

Of the LINQ methods which use deferred execution, some require a single value to be evaluated at a time. The following code:

```
var lst = new List<int>() {3, 5, 1, 2};
var streamingQuery = lst.Select(x => {
    Console.WriteLine(x);
    return x;
});
foreach (var i in streamingQuery) {
    Console.WriteLine($"foreach iteration value: {i}");
}
```

will output:

```
3
foreach iteration value: 3
5
foreach iteration value: 5
1
foreach iteration value: 1
2
foreach iteration value: 2
```

because the function passed to `Select` is evaluated at each iteration of the `foreach`. This is known as **streaming mode** or **lazy evaluation**.

Other LINQ methods -- sorting and grouping operators -- require *all* the values to be evaluated, before they can return *any* value:

```
var nonStreamingQuery = lst.OrderBy(x => {
    Console.WriteLine(x);
    return x;
});
foreach (var i in nonStreamingQuery) {
    Console.WriteLine($"foreach iteration value: {i}");
}
```

will output:

```
3
5
1
2
foreach iteration value: 1
foreach iteration value: 2
foreach iteration value: 3
foreach iteration value: 5
```

In this case, because the values must be generated to the `foreach` in ascending order, all the elements must first be evaluated, in order to determine which is the smallest, and which is the next smallest, and so on. This is known as **non-streaming mode** or **eager evaluation**.

Whether a particular LINQ method uses streaming or non-streaming mode, can be found at MSDN -- C#, or VB.NET.

Benefits of deferred execution - building queries

Deferred execution enables combining different operations to build the final query, before evaluating the values:

```
var list = new List<int>() {1,1,2,3,5,8};
var query = list.Select(x => x + 1);
```

If we execute the query at this point:

```
foreach (var x in query) {
    Console.Write($"{x} ");
}
```

we would get the following output:

```
2 2 3 4 6 9
```

But we can modify the query by adding more operators:

```
Console.WriteLine();
query = query.Where(x => x % 2 == 0);
query = query.Select(x => x * 10);

foreach (var x in query) {
    Console.Write($"{x} ");
}
```

Output:

```
20 20 40 60
```

Benefits of deferred execution - querying current data

With deferred execution, if the data to be queried is changed, the query object uses the data at the time of execution, not at the time of definition.

```
var data = new List<int>() {2, 4, 6, 8};
var query = data.Select(x => x * x);
```

If we execute the query at this point with an immediate method or `foreach`, the query will operate on the list of even numbers.

However, if we change the values in the list:

```
data.Clear();
data.AddRange(new [] {1, 3, 5, 7, 9});
```

or even if we assign a a new list to `data`:

```
data = new List<int>() {1, 3, 5, 7, 9};
```

and then execute the query, the query will operate on the new value of `data`:

```
foreach (var x in query) {
    Console.Write($"{x} ");
}
```

and will output the following:

```
1 9 25 49 81
```

Read Method execution modes - immediate, deferred streaming, deferred non-streaming online: https://riptutorial.com/linq/topic/7102/method-execution-modes---immediate--deferred-streaming--deferred-non-streaming

Chapter 4: Standard Query Operators

Remarks

Linq queries are written using the Standard Query Operators (which are a set of extension methods that operates mainly on objects of type `IEnumerable<T>` and `IQueryable<T>`) or using Query Expressions (which at compile time, are converted to Standard Query Operator method calls).

Query operators provide query capabilities including filtering, projection, aggregation, sorting and more.

Examples

Concatenation Operations

Concatenation refers to the operation of appending one sequence to another.

Concat

Concatenates two sequences to form one sequence.

Method Syntax

```
// Concat

var numbers1 = new int[] { 1, 2, 3 };
var numbers2 = new int[] { 4, 5, 6 };

var numbers = numbers1.Concat(numbers2);

// numbers = { 1, 2, 3, 4, 5, 6 }
```

Query Syntax

```
// Not applicable.
```

Filtering Operations

Filtering refers to the operations of restricting the result set to contain only those elements that satisfy a specified condition.

Where

Selects values that are based on a predicate function.

Method Syntax

```
// Where

var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };

var evens = numbers.Where(n => n % 2 == 0);

// evens = { 2, 4, 6, 8 }
```

Query Syntax

```
// where

var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8 };

var odds = from n in numbers
           where n % 2 != 0
           select n;

// odds = { 1, 3, 5, 7 }
```

OfType

> Selects values, depending on their ability to be cast to a specified type.

Method Syntax

```
// OfType

var numbers = new object[] { 1, "one", 2, "two", 3, "three" };

var strings = numbers.OfType<string>();

// strings = { "one", "two", "three" }
```

Query Syntax

```
// Not applicable.
```

Join Operations

A join of two data sources is the association of objects in one data source with objects that share a common attribute in another data source.

Join

> Joins two sequences based on key selector functions and extracts pairs of values.

Method Syntax

```
// Join

class Customer
```

```
{
    public int Id { get; set; }
    public string Name { get; set; }
}

class Order
{
    public string Description { get; set; }
    public int CustomerId { get; set; }
}
...

var customers = new Customer[]
{
    new Customer { Id = 1, Name = "C1" },
    new Customer { Id = 2, Name = "C2" },
    new Customer { Id = 3, Name = "C3" }
};

var orders = new Order[]
{
    new Order { Description = "O1", CustomerId = 1 },
    new Order { Description = "O2", CustomerId = 1 },
    new Order { Description = "O3", CustomerId = 2 },
    new Order { Description = "O4", CustomerId = 3 },
};

var join = customers.Join(orders, c => c.Id, o => o.CustomerId, (c, o) => c.Name + "-" +
o.Description);

// join = { "C1-O1", "C1-O2", "C2-O3", "C3-O4" }
```

Query Syntax

```
// join … in … on … equals …

var join = from c in customers
           join o in orders
           on c.Id equals o.CustomerId
           select o.Description + "-" + c.Name;

// join = { "O1-C1", "O2-C1", "O3-C2", "O4-C3" }
```

GroupJoin

> Joins two sequences based on key selector functions and groups the resulting matches for each element.

Method Syntax

```
// GroupJoin

var groupJoin = customers.GroupJoin(orders,
                                    c => c.Id,
                                    o => o.CustomerId,
                                    (c, ors) => c.Name + "-" + string.Join(",", ors.Select(o
=> o.Description)));
```

```
// groupJoin = { "C1-O1,O2", "C2-O3", "C3-O4" }
```

Query Syntax

```
// join … in … on … equals … into …

var groupJoin = from c in customers
                join o in orders
                on c.Id equals o.CustomerId
                into customerOrders
                select string.Join(",", customerOrders.Select(o => o.Description)) + "-" +
c.Name;

// groupJoin = { "O1,O2-C1", "O3-C2", "O4-C3" }
```

Zip

> Applies a specified function to the corresponding elements of two sequences, producing a sequence of the results.

```
var numbers = new [] { 1, 2, 3, 4, 5, 6 };
var words = new [] { "one", "two", "three" };

var numbersWithWords =
    numbers
    .Zip(
        words,
        (number, word) => new { number, word });

// Results

//| number | word   |
//| ------ | ------ |
//| 1      |  one   |
//| 2      |  two   |
//| 3      |  three |
```

Projection Operations

Projection refers to the operations of transforming an object into a new form.

Select

> Projects values that are based on a transform function.

Method Syntax

```
// Select

var numbers = new int[] { 1, 2, 3, 4, 5 };

var strings = numbers.Select(n => n.ToString());

// strings = { "1", "2", "3", "4", "5" }
```

Query Syntax

```
// select

var numbers = new int[] { 1, 2, 3, 4, 5 };

var strings = from n in numbers
              select n.ToString();

// strings = { "1", "2", "3", "4", "5" }
```

SelectMany

> Projects sequences of values that are based on a transform function and then flattens them into one sequence.

Method Syntax

```
// SelectMany

class Customer
{
    public Order[] Orders { get; set; }
}

class Order
{
    public Order(string desc) { Description = desc; }
    public string Description { get; set; }
}
...

var customers = new Customer[]
{
    new Customer { Orders = new Order[] { new Order("O1"), new Order("O2") } },
    new Customer { Orders = new Order[] { new Order("O3") } },
    new Customer { Orders = new Order[] { new Order("O4") } },
};

var orders = customers.SelectMany(c => c.Orders);

// orders = { Order("O1"), Order("O3"), Order("O3"), Order("O4") }
```

Query Syntax

```
// multiples from

var orders = from c in customers
             from o in c.Orders
             select o;

// orders = { Order("O1"), Order("O3"), Order("O3"), Order("O4") }
```

Sorting Operations

A sorting operation orders the elements of a sequence based on one or more attributes.

OrderBy

Sorts values in ascending order.

Method Syntax

```
// OrderBy

var numbers = new int[] { 5, 4, 8, 2, 7, 1, 9, 3, 6 };

var ordered = numbers.OrderBy(n => n);

// ordered = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

Query Syntax

```
// orderby

var numbers = new int[] { 5, 4, 8, 2, 7, 1, 9, 3, 6 };

var ordered = from n in numbers
              orderby n
              select n;

// ordered = { 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

OrderByDescending

Sorts values in descending order.

Method Syntax

```
// OrderByDescending

var numbers = new int[] { 5, 4, 8, 2, 7, 1, 9, 3, 6 };

var ordered = numbers.OrderByDescending(n => n);

// ordered = { 9, 8, 7, 6, 5, 4, 3, 2, 1 }
```

Query Syntax

```
// orderby

var numbers = new int[] { 5, 4, 8, 2, 7, 1, 9, 3, 6 };

var ordered = from n in numbers
              orderby n descending
              select n;

// ordered = { 9, 8, 7, 6, 5, 4, 3, 2, 1 }
```

ThenBy

Performs a secondary sort in ascending order.

Method Syntax

```
// ThenBy

string[] words = { "the", "quick", "brown", "fox", "jumps" };

var ordered = words.OrderBy(w => w.Length).ThenBy(w => w[0]);

// ordered = { "fox", "the", "brown", "jumps", "quick" }
```

Query Syntax

```
// orderby …, …

string[] words = { "the", "quick", "brown", "fox", "jumps" };

var ordered = from w in words
              orderby w.Length, w[0]
              select w;

// ordered = { "fox", "the", "brown", "jumps", "quick" }
```

ThenByDescending

Performs a secondary sort in descending order.

Method Syntax

```
// ThenByDescending

string[] words = { "the", "quick", "brown", "fox", "jumps" };

var ordered = words.OrderBy(w => w[0]).ThenByDescending(w => w.Length);

// ordered = { "brown", "fox", "jumps", "quick", "the" }
```

Query Syntax

```
// orderby …, … descending

string[] words = { "the", "quick", "brown", "fox", "jumps" };

var ordered = from w in words
              orderby w.Length, w[0] descending
              select w;

// ordered = { "the", "fox", "quick", "jumps", "brown" }
```

Reverse

Reverses the order of the elements in a collection.

Method Syntax

```
// Reverse

var numbers = new int[] { 1, 2, 3, 4, 5 };

var reversed = numbers.Reverse();

// reversed = { 5, 4, 3, 2, 1 }
```

Query Syntax

```
// Not applicable.
```

Conversion Operations

Conversion operations change the type of input objects.

AsEnumerable

Returns the input typed as IEnumerable.

Method Syntax

```
// AsEnumerable

int[] numbers = { 1, 2, 3, 4, 5 };

var nums = numbers.AsEnumerable();

// nums: static type is IEnumerable<int>
```

Query Syntax

```
// Not applicable.
```

AsQueryable

Converts a IEnumerable to a IQueryable.

Method Syntax

```
// AsQueryable

int[] numbers = { 1, 2, 3, 4, 5 };

var nums = numbers.AsQueryable();

// nums: static type is IQueryable<int>
```

Query Syntax

```
// Not applicable.
```

Cast

Casts the elements of a collection to a specified type.

Method Syntax

```
// Cast

var numbers = new object[] { 1, 2, 3, 4, 5 };

var nums = numbers.Cast<int>();

// nums: static type is IEnumerable<int>
```

Query Syntax

```
// Use an explicitly typed range variable.

var numbers = new object[] { 1, 2, 3, 4, 5 };

var nums = from int n in numbers select n;

// nums: static type is IEnumerable<int>
```

OfType

Filters values, depending on their ability to be cast to a specified type.

Method Syntax

```
// OfType

var objects = new object[] { 1, "one", 2, "two", 3, "three" };

var numbers = objects.OfType<int>();

// nums = { 1, 2, 3 }
```

Query Syntax

```
// Not applicable.
```

ToArray

Converts a collection to an array.

Method Syntax

```
// ToArray

var numbers = Enumerable.Range(1, 5);

int[] array = numbers.ToArray();

// array = { 1, 2, 3, 4, 5 }
```

Query Syntax

```
// Not applicable.
```

ToList

Converts a collection to a list.

Method Syntax

```
// ToList

var numbers = Enumerable.Range(1, 5);

List<int> list = numbers.ToList();

// list = { 1, 2, 3, 4, 5 }
```

Query Syntax

```
// Not applicable.
```

ToDictionary

Puts elements into a dictionary based on a key selector function.

Method Syntax

```
// ToDictionary

var numbers = new int[] { 1, 2, 3 };

var dict = numbers.ToDictionary(n => n.ToString());

// dict = { "1" => 1, "2" => 2, "3" => 3 }
```

Query Syntax

```
// Not applicable.
```

Aggregation Operations

Aggregation operations computes a single value from a collection of values.

Aggregate

> Performs a custom aggregation operation on the values of a collection.

Method Syntax

```
// Aggregate

var numbers = new int[] { 1, 2, 3, 4, 5 };

var product = numbers.Aggregate(1, (acc, n) => acc * n);

// product = 120
```

Query Syntax

```
// Not applicable.
```

Average

> Calculates the average value of a collection of values.

Method Syntax

```
// Average

var numbers = new int[] { 1, 2, 3, 4, 5 };

var average = numbers.Average();

// average = 3
```

Query Syntax

```
// Not applicable.
```

Count

> Counts the elements in a collection, optionally only those elements that satisfy a predicate function.

Method Syntax

```
// Count

var numbers = new int[] { 1, 2, 3, 4, 5 };
```

```
int count = numbers.Count(n => n % 2 == 0);

// count = 2
```

Query Syntax

```
// Not applicable.
```

LongCount

> Counts the elements in a large collection, optionally only those elements that satisfy a predicate function.

Method Syntax

```
// LongCount

var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

long count = numbers.LongCount();

// count = 10
```

Query Syntax

```
// Not applicable.
```

Max

> Determines the maximum value in a collection. Throws exception if collection is empty.

Method Syntax

```
// Max

var numbers = new int[] { 1, 2, 3, 4, 5 };

var max = numbers.Max();

// max = 5
```

Query Syntax

```
// Not applicable.
```

Min

> Determines the minimum value in a collection. Throws exception if collection is empty.

Method Syntax

```
// Min

var numbers = new int[] { 1, 2, 3, 4, 5 };

var min = numbers.Min();

// min = 1
```

Query Syntax

```
// Not applicable.
```

Min-/MaxOrDefault

> Unlike other LinQ extensions `Min()` and `Max()` do not have an overload without exceptions. Therefor the `IEnumerable` must be checked for `Any()` before calling `Min()` or `Max()`

```
// Max

var numbers = new int[] { };

var max = numbers.Any() ? numbers.Max() : 0;

// max = 0
```

Sum

> Calculates the sum of the values in a collection.

Method Syntax

```
// Sum

var numbers = new int[] { 1, 2, 3, 4, 5 };

var sum = numbers.Sum();

// sum = 15
```

Query Syntax

```
// Not applicable.
```

Quantifier Operations

Quantifier operations return a Boolean value that indicates whether some or all of the elements in a sequence satisfy a condition.

All

Determines whether all the elements in a sequence satisfy a condition.

Method Syntax

```
// All

var numbers = new int[] { 1, 2, 3, 4, 5 };

bool areLessThan10 = numbers.All(n => n < 10);

// areLessThan10 = true
```

Query Syntax

```
// Not applicable.
```

Any

Determines whether any elements in a sequence satisfy a condition.

Method Syntax

```
// Any

var numbers = new int[] { 1, 2, 3, 4, 5 };

bool anyOneIsEven = numbers.Any(n => n % 2 == 0);

// anyOneIsEven = true
```

Query Syntax

```
// Not applicable.
```

Contains

Determines whether a sequence contains a specified element.

Method Syntax

```
// Contains

var numbers = new int[] { 1, 2, 3, 4, 5 };

bool appears = numbers.Contains(10);

// appears = false
```

Query Syntax

```
// Not applicable.
```

Grouping Operations

Grouping refers to the operations of putting data into groups so that the elements in each group share a common attribute.

GroupBy

> Groups elements that share a common attribute.

Method Syntax

```
// GroupBy

class Order
{
    public string Customer { get; set; }
    public string Description { get; set; }
}
...

var orders = new Order[]
{
    new Order { Customer = "C1", Description = "O1" },
    new Order { Customer = "C2", Description = "O2" },
    new Order { Customer = "C3", Description = "O3" },
    new Order { Customer = "C1", Description = "O4" },
    new Order { Customer = "C1", Description = "O5" },
    new Order { Customer = "C3", Description = "O6" },
};

var groups = orders.GroupBy(o => o.Customer);

// groups: { (Key="C1", Values="O1","O4","O5"), (Key="C2", Values="O2"), (Key="C3",
Values="O3","O6") }
```

Query Syntax

```
// group … by

var groups = from o in orders
             group o by o.Customer;

// groups: { (Key="C1", Values="O1","O4","O5"), (Key="C2", Values="O2"), (Key="C3",
Values="O3","O6") }
```

ToLookup

> Inserts elements into a one-to-many dictionary based on a key selector function.

Method Syntax

```
// ToLookUp
```

```
var ordersByCustomer = orders.ToLookup(o => o.Customer);

// ordersByCustomer = ILookUp<string, Order>
// {
//     "C1" => { Order("01"), Order("04"), Order("05") },
//     "C2" => { Order("02") },
//     "C3" => { Order("03"), Order("06") }
// }
```

Query Syntax

```
// Not applicable.
```

Partition Operations

Partitioning refers to the operations of dividing an input sequence into two sections, without rearranging the elements, and then returning one of the sections.

Skip

Skips elements up to a specified position in a sequence.

Method Syntax

```
// Skip

var numbers = new int[] { 1, 2, 3, 4, 5 };

var skipped = numbers.Skip(3);

// skipped = { 4, 5 }
```

Query Syntax

```
// Not applicable.
```

SkipWhile

Skips elements based on a predicate function until an element does not satisfy the condition.

Method Syntax

```
// Skip

var numbers = new int[] { 1, 3, 5, 2, 1, 3, 5 };

var skipLeadingOdds = numbers.SkipWhile(n => n % 2 != 0);

// skipLeadingOdds = { 2, 1, 3, 5 }
```

Query Syntax

```
// Not applicable.
```

Take

Takes elements up to a specified position in a sequence.

Method Syntax

```
// Take

var numbers = new int[] { 1, 2, 3, 4, 5 };

var taken = numbers.Take(3);

// taken = { 1, 2, 3 }
```

Query Syntax

```
// Not applicable.
```

TakeWhile

Takes elements based on a predicate function until an element does not satisfy the condition.

Method Syntax

```
// TakeWhile

var numbers = new int[] { 1, 3, 5, 2, 1, 3, 5 };

var takeLeadingOdds = numbers.TakeWhile(n => n % 2 != 0);

// takeLeadingOdds = { 1, 3, 5 }
```

Query Syntax

```
// Not applicable.
```

Generation Operations

Generation refers to creating a new sequence of values.

DefaultIfEmpty

Replaces an empty collection with a default valued singleton collection.

Method Syntax

```
// DefaultIfEmpty

var nums = new int[0];

var numbers = nums.DefaultIfEmpty();

// numbers = { 0 }
```

Query Syntax

```
// Not applicable.
```

Empty

Returns an empty collection.

Method Syntax

```
// Empty

var empty = Enumerable.Empty<string>();

// empty = IEnumerable<string> { }
```

Query Syntax

```
// Not applicable.
```

Range

Generates a collection that contains a sequence of numbers.

Method Syntax

```
// Range

var range = Enumerable.Range(1, 5);

// range = { 1, 2, 3, 4, 5 }
```

Query Syntax

```
// Not applicable.
```

Repeat

Generates a collection that contains one repeated value.

Method Syntax

```
// Repeat

var repeats = Enumerable.Repeat("s", 3);

// repeats = { "s", "s", "s" }
```

Query Syntax

```
// Not applicable.
```

Set Operations

Set operations refer to query operations that produce a result set that is based on the presence or absence of equivalent elements within the same or separate collections (or sets).

Distinct

> Removes duplicate values from a collection.

Method Syntax

```
// Distinct

var numbers = new int[] { 1, 2, 3, 1, 2, 3 };

var distinct = numbers.Distinct();

// distinct = { 1, 2, 3 }
```

Query Syntax

```
// Not applicable.
```

Except

> Returns the set difference, which means the elements of one collection that do not appear in a second collection.

Method Syntax

```
// Except

var numbers1 = new int[] { 1, 2, 3, 4, 5 };
var numbers2 = new int[] { 4, 5, 6, 7, 8 };

var except = numbers1.Except(numbers2);

// except = { 1, 2, 3 }
```

Query Syntax

```
// Not applicable.
```

Intersect

> Returns the set intersection, which means elements that appear in each of two collections.

Method Syntax

```
// Intersect

var numbers1 = new int[] { 1, 2, 3, 4, 5 };
var numbers2 = new int[] { 4, 5, 6, 7, 8 };

var intersect = numbers1.Intersect(numbers2);

// intersect = { 4, 5 }
```

Query Syntax

```
// Not applicable.
```

Union

> Returns the set union, which means unique elements that appear in either of two collections.

Method Syntax

```
// Union

var numbers1 = new int[] { 1, 2, 3, 4, 5 };
var numbers2 = new int[] { 4, 5, 6, 7, 8 };

var union = numbers1.Union(numbers2);

// union = { 1, 2, 3, 4, 5, 6, 7, 8 }
```

Query Syntax

```
// Not applicable.
```

Equality Operations

Two sequences whose corresponding elements are equal and which have the same number of elements are considered equal.

SequenceEqual

> Determines whether two sequences are equal by comparing elements in a pair-wise manner.

Method Syntax

```
// SequenceEqual

var numbers1 = new int[] { 1, 2, 3, 4, 5 };
var numbers2 = new int[] { 1, 2, 3, 4, 5 };

var equals = numbers1.SequenceEqual(numbers2);

// equals = true
```

Query Syntax

```
// Not Applicable.
```

Element Operations

Element operations return a single, specific element from a sequence.

ElementAt

> Returns the element at a specified index in a collection.

Method Syntax

```
// ElementAt

var strings = new string[] { "zero", "one", "two", "three" };

var str = strings.ElementAt(2);

// str = "two"
```

Query Syntax

```
// Not Applicable.
```

ElementAtOrDefault

> Returns the element at a specified index in a collection or a default value if the index is out of range.

Method Syntax

```
// ElementAtOrDefault
```

```
var strings = new string[] { "zero", "one", "two", "three" };

var str = strings.ElementAtOrDefault(10);

// str = null
```

Query Syntax

```
// Not Applicable.
```

First

> Returns the first element of a collection, or the first element that satisfies a condition.

Method Syntax

```
// First

var numbers = new int[] { 1, 2, 3, 4, 5 };

var first = strings.First();

// first = 1
```

Query Syntax

```
// Not Applicable.
```

FirstOrDefault

> Returns the first element of a collection, or the first element that satisfies a condition. Returns a default value if no such element exists.

Method Syntax

```
// FirstOrDefault

var numbers = new int[] { 1, 2, 3, 4, 5 };

var firstGreaterThanTen = strings.FirstOrDefault(n => n > 10);

// firstGreaterThanTen = 0
```

Query Syntax

```
// Not Applicable.
```

Last

Returns the last element of a collection, or the last element that satisfies a condition.

Method Syntax

```
// Last

var numbers = new int[] { 1, 2, 3, 4, 5 };

var last = strings.Last();

// last = 5
```

Query Syntax

```
// Not Applicable.
```

LastOrDefault

Returns the last element of a collection, or the last element that satisfies a condition. Returns a default value if no such element exists.

Method Syntax

```
// LastOrDefault

var numbers = new int[] { 1, 2, 3, 4, 5 };

var lastGreaterThanTen = strings.LastOrDefault(n => n > 10);

// lastGreaterThanTen = 0
```

Query Syntax

```
// Not Applicable.
```

Single

Returns the only element of a collection, or the only element that satisfies a condition.

Method Syntax

```
// Single

var numbers = new int[] { 1 };

var single = strings.Single();

// single = 1
```

Query Syntax

```
// Not Applicable.
```

SingleOrDefault

Returns the only element of a collection, or the only element that satisfies a condition. Returns a default value if no such element exists or the collection does not contain exactly one element.

Method Syntax

```
// SingleOrDefault

var numbers = new int[] { 1, 2, 3, 4, 5 };

var singleGreaterThanFour = strings.SingleOrDefault(n => n > 4);

// singleGreaterThanFour = 5
```

Query Syntax

```
// Not Applicable.
```

Read Standard Query Operators online: https://riptutorial.com/linq/topic/2535/standard-query-operators

Credits

S. No	Chapters	Contributors
1	Getting started with linq	AlexFoxGill, Arturo Menchaca, Colin Young, Community, David B, flindeberg, Ivan Yurchenko, Joshua Poling, Kobi, Mark Hurd, Matthew Haugen, meJustAndrew, mmushtaq, Peter Mortensen, Richard Everett, Ryan Abbott, Tom Wuyts, Travis J, Wyck, Zev Spitz
2	Linq Using Take while And Skip While	kari kalan
3	Method execution modes - immediate, deferred streaming, deferred non-streaming	Colin Young, mmushtaq, Travis J, Zev Spitz
4	Standard Query Operators	Arturo Menchaca, James Cockayne, Mark Hurd, Toxantron