**Architecture of LINQ**

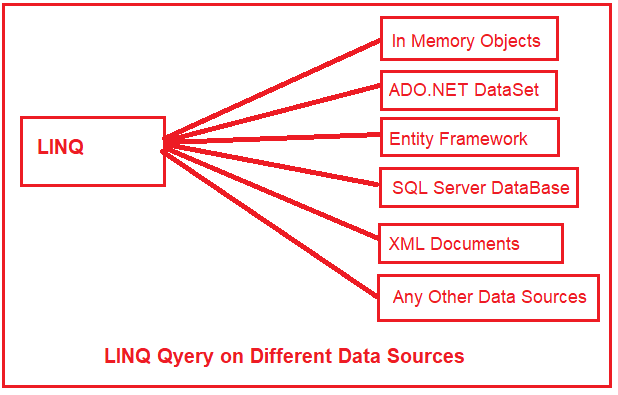
In this article, I am going to discuss the **Architecture of LINQ**. The term **LINQ** stands for **Language Integrated Query** and it is pronounced as **LINK**. Nowadays the use of use LINQ increasing rapidly. So, as a developer, you should understand the Linq and its architecture. At the end of this article, you will have a very good understanding of the following pointers.

1. **What is LINQ?**
2. **Why should we learn LINQ?**
3. **How LINQ works?**
4. **What are LINQ Providers?**
5. **Advantages of using LINQ.**
6. **Disadvantages of using LINQ.**

**What is LINQ?**

The LINQ (Language Integrated Query) is a part of a language but not a complete language. It was introduced by Microsoft with .NET Framework 3.5 and C# 3.0 and is available in **System.Linq** namespace.

LINQ provides us common query syntax which allows us to query the data from various data sources. That means using a single query we can get or set 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.

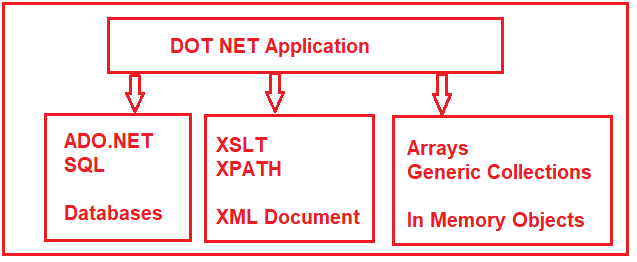


**Why should we learn LINQ?**

Let us understand why we should learn LINQ with an example.

Suppose we are developing a .NET Application and that application requires data from different data sources. For example

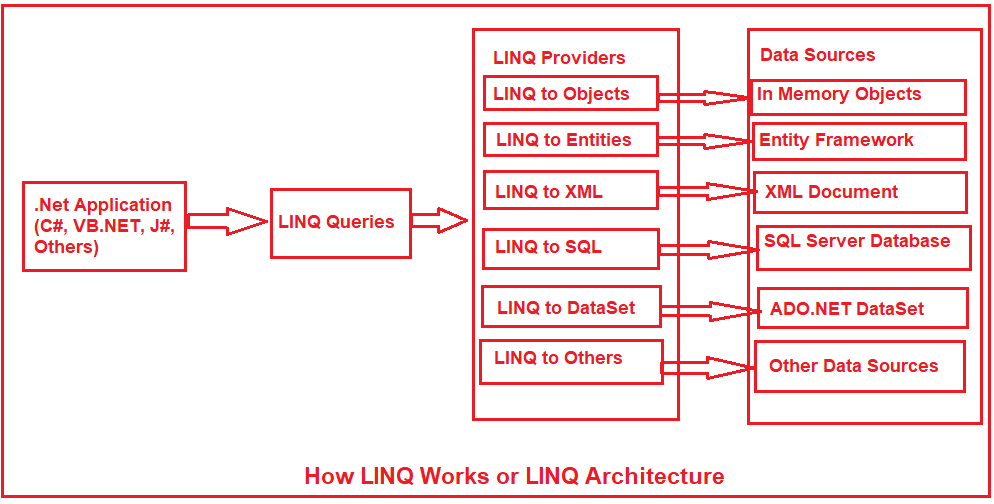
1. The application needs data from SQL Server Database. So as a developer in order to access the data from SQL Server Database, we need to understand ADO.NET and SQL Server-specific syntaxes. If the database is Oracle then we need to learn SQL Syntax specific to Oracle Database.
2. The application also needs data from an XML Document. So as a developer in order to work with XML Document, we need to understand **XPath and** **XSLT**queries.
3. The application also needs to manipulate the data (objects) in memory such as **List<Products>, List<Orders>**, etc. So as a developer we should also have to understand how to work with in-memory objects.



LINQ provides a **uniform programming model**(i.e. common query syntax) which allows us to work with different data sources but using a standard or you can say unified coding style. As a result, we don’t require to learn different syntaxes to query different data sources.

**Note:** If you are either a C# or a VB.NET Developer (Web, Windows, Mobile, Console, etc.) then you should learn LINQ.

**How LINQ works?**



As shown in the above diagram, you can write the LINQ queries using any .NET supported programming languages such as C#, VB.NET, J#, etc.

The LINQ provider is a software component that lies between the LINQ queries and the actual data source. The Linq provider will convert the LINQ queries into a format that can be understood by the underlying data source. For example, **LINQ to SQL** provider will convert the LINQ queries to SQL statements which can be understood by the SQL Server database. Similarly, the **LINQ to XML** provider will convert the queries into a format that can be understood by the XML document.

**What are LINQ Providers?**

A LINQ provider is software that implements the **IQueryProvider** and **IQueryable** interface for a particular data source. In other words, it allows us to write LINQ queries against that data source. If you want to create your custom LINQ provider then it must implement **IQueryProvider** and **IQueryable** interface. Without LINQ Provider we cannot execute our LINQ Queries.

Let us discuss some of the LINQ Providers and how they work internally.

**LINQ to Objects:**

The 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 SQL (DLINQ):**

The 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. These .NET classes are going to be created automatically by the wizard based on the database table.

**LINQ to Datasets:**

The LINQ to Datasets provider provides us the flexibility to query data cached in a Dataset in an easy and faster way. It also allows us to do further data manipulation operations such as searching, filtering, sorting, etc. on the Dataset using the LINQ Syntax.

**LINQ to Entities:**

The LINQ to Entities provider 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. The point that you need to remember is, it is used to query any database such as SQL Server, Oracle, MySQL, DB2, etc. Now, it is called ADO.NET Entity Framework.

**LINQ to XML (XLINQ):**

The 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. The **System.Xml.Linq** namespace contains the required classes for LINQ to XML.

**Parallel LINQ (PLINQ):**

The Parallel LINQ or PLINQ was introduced with .NET Framework 4.0. This provider provides the flexibility of parallel implementation of LINQ to Objects. The PLINQ was designed in such a way that it uses the power of parallel programming which targets the Task Parallel Library. So if you want to execute your LINQ query simultaneously or parallel on different processors then you need to write the query using PLINQ.

**Advantages of using LINQ?**

1. We don’t need to learn new query language syntaxes for different data sources as it provides common query syntax to query different data sources.
2. Less code as compared to the traditional approach. That means using LINQ we can minimize our code.
3. It provides Compile time error checking as well as intelligence support in Visual Studio. This powerful feature helps us to avoid run-time errors.
4. LINQ provides a lot of inbuilt methods that we can use to perform different operations such as filtering, ordering, grouping, etc. which makes our work easy.
5. Its query can be reused.

**Disadvantages of using LINQ?**

The disadvantages of using LINQ are as follows:

1. Using LINQ it’s very difficult to write complex queries like SQL.
2. LINQ doesn’t take the full advantage of SQL features like cached execution plan for the stored procedure.
3. We will get the worst performance if we don’t write the queries properly.
4. If we do some changes to our queries, then we need to recompile the application and need to redeploy the dll to the server.

# Different Ways to write LINQ Query

## ****Different Ways to Write LINQ Query with Examples****

In this article, I am going to discuss the **Different Ways to write LINQ Query** i.e. Linq Query Syntax and Linq Method Syntax with examples. Please read our previous article where we discussed the [**Architecture of LINQ**](https://dotnettutorials.net/lesson/introduction-to-linq/) i.e. how LINQ works. In this article, we are going to discuss the following pointers.

1. **Different things required to write a LINQ Query?**
2. **What is a Query?**
3. **What are the different ways to write a LINQ query?**
4. **Example of using both Method and Query Syntax.**

##### ****What are the different things required to write a LINQ Query?****

In order to write a LINQ query, we need the following three things

1. Data Source (in-memory objects, SQL, XML)
2. Query
3. Execution of the query

##### ****What is a Query?****

A query is nothing but a set of instructions which is applied on a data source (i.e. in-memory objects, SQL, XML, etc.) to perform certain operations (i.e. CRUD operations) and then tells the shape of the output from that query. That means the query is not responsible for what will be the output rather it is responsible for the shape of the output. Means what is going to return from that query whether it is going to return a particular value, or a particular list or an object.

Each query is a combination of three things. They are as follows:

1. Initialization (to work with a particular data source)
2. Condition (where, filter, sorting condition)
3. Selection (single selection, group selection or joining)

##### ****What are the different ways to write a LINQ query?****

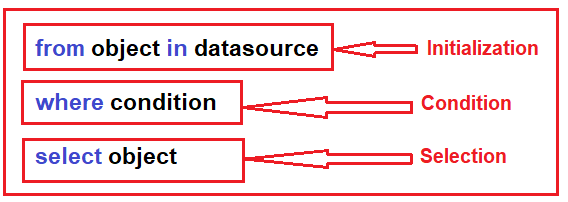
We can write the LINQ query in three different ways. They are as follows

1. Query Syntax
2. Method Syntax
3. Mixed Syntax (Query + Method)

**Note:** From the performance point of view there is no difference between the above three approaches. So, which you need to use, it totally depends on your personal preference. But the point that you need to keep in mind is, behind the scene, the LINQ queries written using query syntax are translated into their lambda expressions before they are compiled.

##### ****LINQ Query Syntax:****

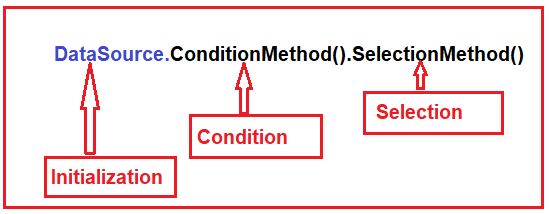
This is one of the easy ways to write complex LINQ queries in an easy and readable format. The syntax for this type of query is very much similar to SQL Query. If you are familiar with SQL queries then it is going to be easy for you to write LINQ queries using this query syntax. The syntax is given below.



##### ****LINQ Method Syntax:****

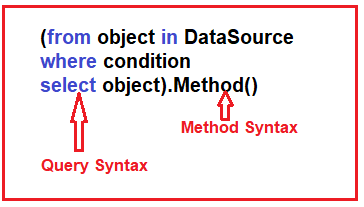
Method syntax becomes most popular now a day to write LINQ queries. It uses a lambda expression to define the condition for the query. Method syntaxes are easy to write simple queries to perform read-write operations on a particular data source. But for complex queries Method syntaxes are a little hard to write as compared to query syntax.

In this approach, the LINQ query is written by using multiple methods by combining them with a dot (.). The Syntax is given below:



##### ****LINQ Mixed Syntax:****

This is the combination of both Query and Method syntax. The syntax is given below.



**Let us understand how to use Query Syntax, Method Syntax, and Mixed Syntax with examples.**

**Example:** We have an integer list and we need to write a LINQ query which will return all the integers which are greater than 5. We are going to create a console application.

###### ****Example Using Query Syntax:****

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Data Source

List**<int>** integerList = new List**<int>()**

**{**

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

**}**;

//LINQ Query using Query Syntax

var QuerySyntax = **from** obj in integerList

**where** obj **>** 5

**select** obj;

//Execution

**foreach(**var item in QuerySyntax**)**

**{**

Console.Write**(**item + " "**)**;

**}**

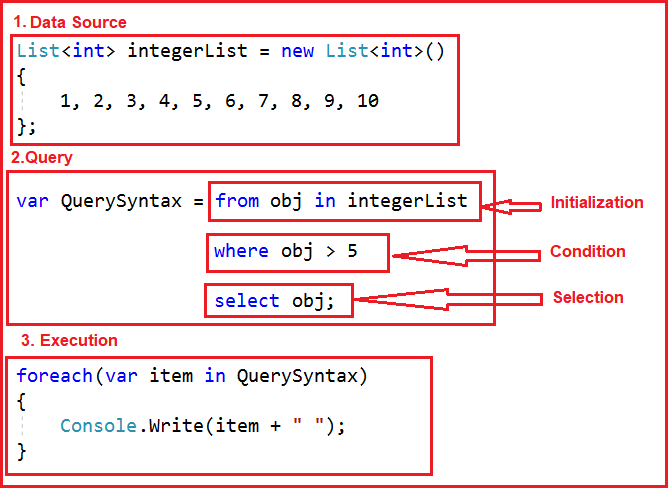
Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it will display the values 6 7 8 9 10 as expected in the console window. Let us understand what we did in the above code.



###### ****Example Using Method Syntax:****

Let us rewrite the same example using LINQ Method Syntax.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Data Source

List**<int>** integerList = new List**<int>()**

**{**

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

**}**;

//LINQ Query using Method Syntax

var MethodSyntax = integerList.Where**(**obj =**>** obj **>** 5**)**.ToList**()**;

//Execution

**foreach(**var item in MethodSyntax**)**

**{**

Console.Write**(**item + " "**)**;

**}**

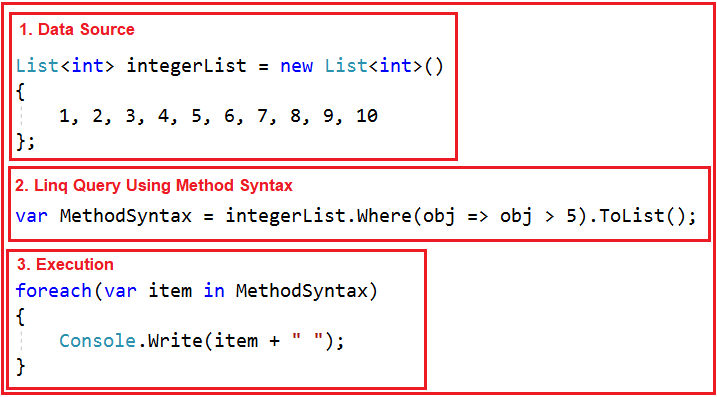
Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and you will get the output as expected. Let us have a look at the following diagram to understand Method Syntax.



###### ****Example Using Mixed Syntax:****

Let us change our requirements. First, we need to filter the list where the value is greater than 5 and then we need to calculate the sum.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Data Source

List**<int>** integerList = new List**<int>()**

**{**

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

**}**;

//LINQ Query using Mixed Syntax

var MethodSyntax = **(from** obj in integerList

**where** obj **>** 5

**select** obj**)**.Sum**()**;

//Execution

Console.Write**(**"Sum Is : " + MethodSyntax**)**;

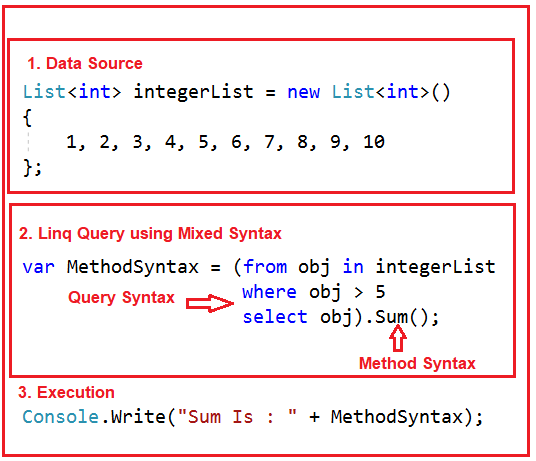
Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and you will see the output as expected. Let us understand what we did in the above code by looking at the following image.



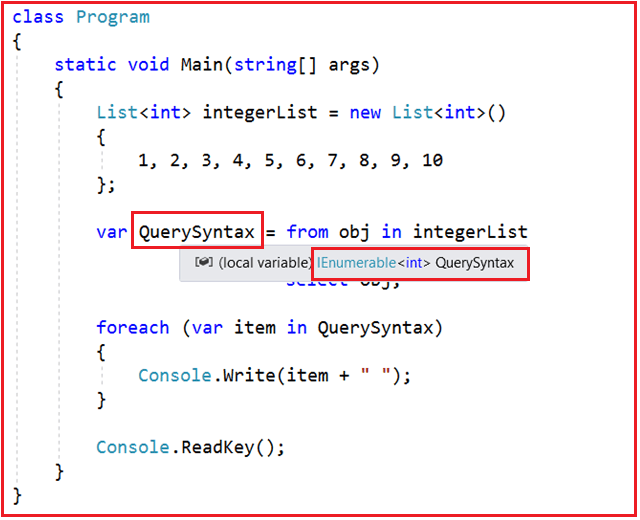
# IEnumerable and IQueryable in C#

## ****IEnumerable and IQueryable in C# with Examples****

In this article, I am going to discuss **IEnumerable and IQueryable in C#** with examples. Please read our previous article where we discussed the [**different ways to write LINQ queries**](https://dotnettutorials.net/lesson/different-ways-to-write-linq-query/) with examples. In this article, we are going to discuss the following concepts in detail.

1. **What is IEnumerable in C#?**
2. **What is IQueryable in C#?**
3. **Examples of IEnumerable and IQueryable**

In our previous article, we write the following program using LINQ Query Syntax.



In the above example, we use the **var** keyword to create the variable and store the result of the LINQ query. So let’s check what is the type of the variable? In order to check this, just mouseover the pointer on to the QuerySynntax variable and you will see that the type is **IEnumerable<int>** which is a generic type. So it is important to understand what is IEnumerable?

So in the above example, instead of writing the **var** keyword, you can also write **IEnumerable<int>** and it should work as expected as shown below.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** integerList = new List**<int>()**

**{**

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

**}**;

IEnumerable**<int>** QuerySyntax = **from** obj in integerList

**where** obj **>** 5

**select** obj;

**foreach** **(**var item in QuerySyntax**)**

**{**

Console.Write**(**item + " "**)**;

**}**

Console.ReadKey**()**;

**}**

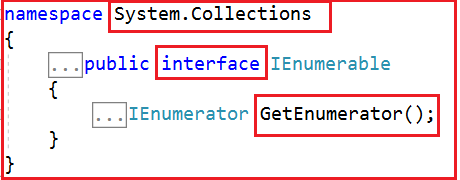
**}**

**}**

With this keep in mind, let us understand what is IEnumerable?

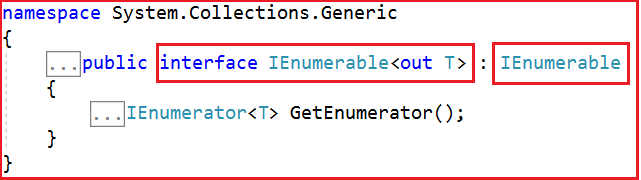
##### ****What is IEnumerable in C#?****

Let us first see the definition of the IEnumerable which is given below.

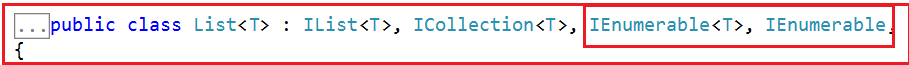


As we see from the above image, **IEnumerable** is an interface that is available in **System.Collection** namespace. The IEnumerable interface is a type of iteration design pattern. It means we can iterate on the collection of the type IEnumerable. As you can see in the above definition, the IEnumerable interface has one method called **GetEnumerator** which will return an **IEnumerator** that iterates through a collection.

The IEnumerable interface has also a child generic interface i.e. **IEnumerable<T>**. Let see the definition of **IEnumerable<T>** interface.



The most important point that you need to remember is, in C#, all the collection classes (both generic and non-generic) implements the IEnumerable interface. Let us prove this by visiting the definition of List<T> generic collection class as shown in the below image.



Now let us see the definition of ArrayList collection which is a non-generic collection class.

ArrayList Collection class Definition

**Note:** The **IEnumerable or IEnumerable<T>** interface should be used only for in-memory data objects. Why that we will discuss in our next article.

##### ****Let us see an example to understand C# IEnumerable with complex type.****

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** studentList = new List**<**Student**>()**

**{**

new Student**(){**ID = 1, Name = "James", Gender = "Male"**}**,

new Student**(){**ID = 2, Name = "Sara", Gender = "Female"**}**,

new Student**(){**ID = 3, Name = "Steve", Gender = "Male"**}**,

new Student**(){**ID = 4, Name = "Pam", Gender = "Female"**}**

**}**;

//Linq Query to Fetch all students with Gender Male

IEnumerable**<**Student**>** QuerySyntax = **from** std in studentList

**where** std.Gender == "Male"

**select** std;

//Iterate through the collection

**foreach** **(**var student in QuerySyntax**)**

**{**

Console.WriteLine**(** $"ID : {student.ID} Name : {student.Name}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**}**

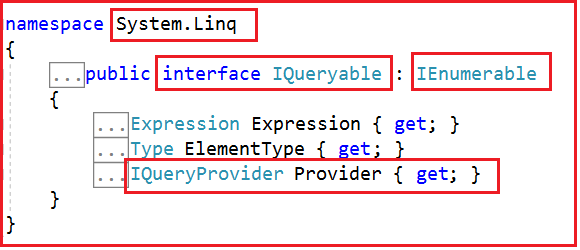
**}**

When we execute the program it will display the result as expected as shown in the below image.

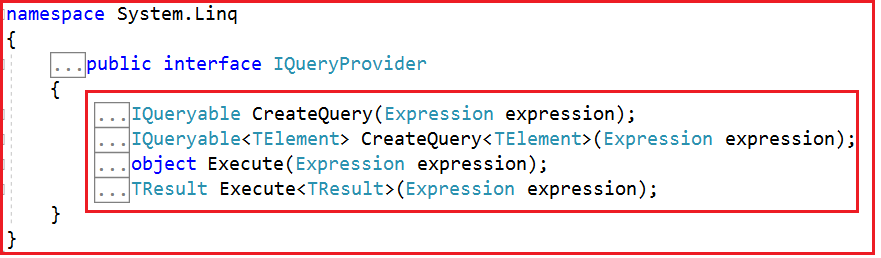
C# IEnumerable examples

##### ****What is IQueryable in C#?****

Let us first see the definition of IQueryable as shown below.



As you can see in the above image, the **IQueryable** is an interface and it is available in **System.Linq** namespace. The IQuerable interface is a child of the IEnumerable interface. So we can store IQuerable in a variable of type IEnumerable. The IQuerable interface has a property called **Provider** which is of type **IQueryProvider** interface. Let us see the definition of IQueryProvider.



The methods provided by the **IQueryProvider** are used to create all Linq Providers. So, this is the best choice for other data providers such as **Linq to SQL, Linq to Entities**, etc. Why that we will discuss in our next article.

##### ****Let us see an example to understand C# IQueryable interface.****

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** studentList = new List**<**Student**>()**

**{**

new Student**(){**ID = 1, Name = "James", Gender = "Male"**}**,

new Student**(){**ID = 2, Name = "Sara", Gender = "Female"**}**,

new Student**(){**ID = 3, Name = "Steve", Gender = "Male"**}**,

new Student**(){**ID = 4, Name = "Pam", Gender = "Female"**}**

**}**;

//Linq Query to Fetch all students with Gender Male

IQueryable**<**Student**>** MethodSyntax = studentList.AsQueryable**()**

.Where**(**std =**>** std.Gender == "Male"**)**;

//Iterate through the collection

**foreach** **(**var student in MethodSyntax**)**

**{**

Console.WriteLine**(** $"ID : {student.ID} Name : {student.Name}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**}**

**}**

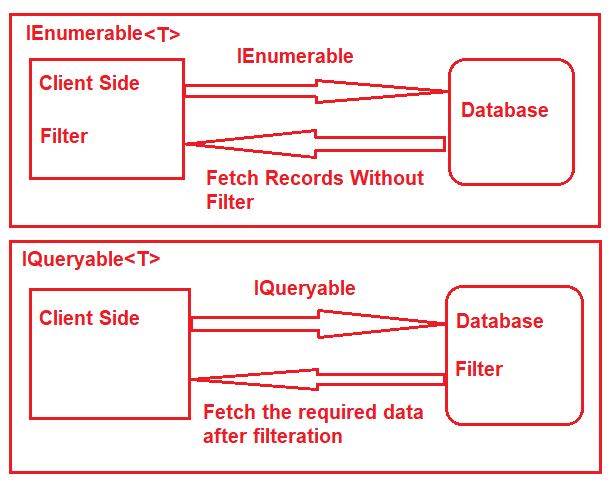
Now run the application and you will see the data as expected. The point that you need to remember is in order to return a collection of type IQueryable, first, you need to call the **AsQueryable()** method on the collection as we did in the above example.

# Differences between IEnumerable and IQueryable

## ****Differences between IEnumerable and IQueryable in C#****

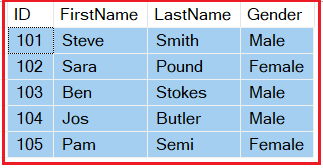
In this article, I am going to discuss the **Differences between IEnumerable and IQueryable in C#** with an example. Please read our previous article before proceeding to this article where we discussed the basics of [**IEnumerable and IQueryable in C#**](https://dotnettutorials.net/lesson/ienumerable-iqueryable-csharp/) with some examples.

The IEnumerable and IQueryable are used to hold a collection of data and also used to perform data manipulation operations such as filtering, Ordering, Grouping, etc. based on the business requirements. Here in this article, we will see the difference between IEnumerable and IQueryable with examples.



##### ****Example:****

Here in this demo, we will create a console application that will retrieve the data from the SQL Server database using Entity Framework database first approach. We are going to fetch the following Student information from the Student table.



Please use the below SQL Script to create and populate the Student table with the required test data.

-- Create the required Student table

**CREATE** **TABLE** Student

(

**ID** **INT** **PRIMARY** **KEY**,

FirstName **VARCHAR**(50),

LastName **VARCHAR**(50),

Gender **VARCHAR**(50)

)

**GO**

-- Insert the required test data

**INSERT** **INTO** Student **VALUES** (101, 'Steve', 'Smith', 'Male')

**INSERT** **INTO** Student **VALUES** (102, 'Sara', 'Pound', 'Female')

**INSERT** **INTO** Student **VALUES** (103, 'Ben', 'Stokes', 'Male')

**INSERT** **INTO** Student **VALUES** (104, 'Jos', 'Butler', 'Male')

**INSERT** **INTO** Student **VALUES** (105, 'Pam', 'Semi', 'Female')

**GO**

Once you created the Student table with the required test data then create a new Console application. Once you create the Console application then add the ADO.NET Entity Data Model Database approach.

##### ****Example: Using IEnumerable****

Let us modify the Program class as shown below.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

StudentDBContext dBContext = new StudentDBContext**()**;

IEnumerable**<**Student**>** listStudents = dBContext.Students.Where**(**x =**>** x.Gender == "Male"**)**;

listStudents = listStudents.Take**(**2**)**;

**foreach(**var std in listStudents**)**

**{**

Console.WriteLine**(**std.FirstName + " " + std.LastName**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Here we create the LINQ Query using IEnumerable. Please use SQL Profiler to log the SQL Script. Now run the application and you will see the following SQL Script is generated and executed.

**SELECT**

[Extent1].[**ID**] **AS** [**ID**],

[Extent1].[FirstName] **AS** [FirstName],

[Extent1].[LastName] **AS** [LastName],

[Extent1].[Gender] **AS** [Gender]

**FROM** [dbo].[Student] **AS** [Extent1]

**WHERE** 'Male' = [Extent1].[Gender]

As shown in the above SQL Script, it will not use the TOP clause. So here it will fetch the data from SQL Server to in-memory and then it will filter the data.

##### ****Example: Using IQueryable****

Let us modify the Program class as shown below to use IQueryable.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

StudentDBContext dBContext = new StudentDBContext**()**;

IQueryable**<**Student**>** listStudents = dBContext.Students

.AsQueryable**()**

.Where**(**x =**>** x.Gender == "Male"**)**;

listStudents = listStudents.Take**(**2**)**;

**foreach(**var std in listStudents**)**

**{**

Console.WriteLine**(**std.FirstName + " " + std.LastName**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Once you run the application, it will create the following SQL Script.

**SELECT** **TOP** (2)

[Extent1].[**ID**] **AS** [**ID**],

[Extent1].[FirstName] **AS** [FirstName],

[Extent1].[LastName] **AS** [LastName],

[Extent1].[Gender] **AS** [Gender]

**FROM** [dbo].[Student] **AS** [Extent1]

**WHERE** 'Male' = [Extent1].[Gender]

As you can see it includes the TOP clause in the SQL Script and then fetches the data from the database. With this keep in mind let us discuss the differences between IEnumerable and IQueryable.

**Differences between IEnumerable and IQueryable in C#:**

##### ****IEnumerable:****

1. IEnumerable is an interface that is available in the System.Collections namespace.
2. While querying the data from the database, the IEnumerable executes the “select statement” on the server-side (i.e. on the database), loads data into memory on the client-side, and then only applied the filters on the retrieved data.
3. So you need to use the IEnumerable when you need to query the data from in-memory collections like List, Array, and so on.
4. The IEnumerable is mostly used for LINQ to Object and LINQ to XML queries.
5. The IEnumerable collection is of type forward only. That means it can only move in forward, it can’t move backward and between the items.
6. IEnumerable supports deferred execution.
7. It doesn’t support custom queries.
8. The IEnumerable doesn’t support lazy loading. Hence, it is not suitable for paging like scenarios.

##### ****IQueryable:****

1. The IQueryable is an interface that exists in the System.Linq Namespace.
2. While querying the data from a database, the IQueryable executes the “select query” with the applied filter on the server-side i.e. on the database, and then retrieves data.
3. So you need to use the IQueryable when you want to query the data from out-memory such as remote database, service, etc.
4. IQueryable is mostly used for LINQ to SQL and LINQ to Entities queries.
5. The collection of type IQueryable can move only forward, it can’t move backward and between the items.
6. IQueryable supports deferred execution.
7. It also supports custom queries using CreateQuery and Executes methods.
8. IQueryable supports lazy loading and hence it is suitable for paging like scenarios.

If you go to the definition of where method, then you will see that it is implemented as an extension method of IQueryable interface as shown below.

https://dotnettutorials.net/wp-content/uploads/2019/04/word-image-2.png

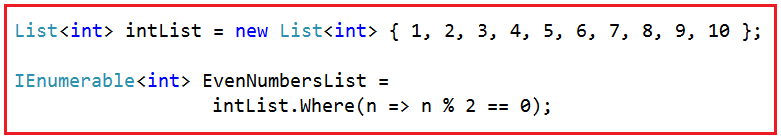
# LINQ Extension Methods

## ****LINQ Extension Methods in C# with Examples****

In this article, I am going to discuss the **LINQ Extension Methods in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Differences between IEnumerable and IQueryable in C#**](https://dotnettutorials.net/lesson/differences-between-ienumerable-and-iqueryable/). At the end of this article, you will understand the following three concepts in C#.

1. **What are Extension Methods in C#?**
2. **When to use Extension Methods in C#?**
3. **How to implement extension methods in C#?**
4. **Understanding the LINQ Extension Method.**

The LINQ’s standard queryoperators such as **select, where,** etc. are implemented in **Enumerable**class. These methods are implemented as extension methods of the type **IEnumerable<T>** interface. Let us understand this with an example. We have the following code in our Main method.



The above **Where()**method is not belonging to **List<T>**class, but still, we are able to call it as it belongs to **List<T>**class. Why it is possible to call it using **List<T>** object, let’s find out. If you go to the definition of where method, then you will find the following definition.

Where Extension Method

As you can see in the signature, the where **Where()**method is implemented as an extension method on **IEnumerable<T>**interface and we know **List<T>**implements **IEnumerable<T>**interface. This is the reason why we are able to call the **Where()** method using the **List<T>** object.

With this keep in mind, let us understand what extension methods are and how they are implemented in C#.

##### ****What are Extension Methods?****

According to MSDN, Extension methods allow us to add methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.

In simple words, we can say that the Extension methods can be used as an approach of extending the functionality of a class by adding new methods in the future if the source code of the class is not available or if we don’t have any permission in making changes to the class.

The most important point that you need to keep in mind is, extension methods are the special kind of static methods of a static class, but they are going to be called as if they were instance methods on the extended type.

##### ****When to use Extension Methods in C#?****

You need to use an extension method if any of the following conditions are true:

1. You need a method on an existing type and you are not the owner of the source code of that type.
2. You need a method on an existing type, you do own the source code of that type but that type is an interface.
3. You need a method on an existing type, you do own the source code and that type is not an interface but adding the method creates undesired coupling.

Otherwise, you should go with the normal method of the actual type itself.

##### ****How to implement extension methods in C#?****

Let us understand this with an example. Our requirement is, we want to add a method in the built-in **string** class, let’s call this method as **GetWordCount()** which will count the word present in a string separated by a space.

For example, if the string is “**Welcome to Dotnet Tutorials**”, then it should return the word count as 4. The most important point is, we need to call this method on the String object as shown below.

**int wordCount = sentence.GetWordCount();**

**Note:** We cannot define the **GetWordCount()** method directly in the string class as we are not the owner of the string class. The string class belongs to **System** namespace which is own by the .NET framework. So, the alternative solution to achieve this is to write a wrapper class as shown below.

**public** **class** ExtensionHelper

**{**

**public** **static** **int** GetWordCount**(**string str**)**

**{**

**if** **(**!String.IsNullOrEmpty**(**str**))**

**return** str.Split**(**' '**)**.Length;

**return** 0;

**}**

**}**

The above **ExtensionHelper** Wrapper class works fine, but the problem is, here we cannot call the **GetWordCount()** method using the string object as shown below.

**int wordCount = sentence.GetWordCount();**

Instead, we need to call the **GetWordCount()** method as shown below.

**int wordCount = ExtensionHelper.GetWordCount(sentence);**

##### ****How to convert the above GetWordCount() method to an Extension Method of string class?****

Now let’s convert the **GetWordCount()** method to an extension method on the String class. So that we can able to call the **GetWordCount()** method using the following syntax.

**int wordCount = sentence.GetWordCount();**

In order to make the above **GetWordCount()** method as an extension method, we need to make two changes which are as follows.

1. First, we need to make the **ExtensionHelper** class as a **static** class.
2. Second, the type the method extends (i.e. **string**) should be passed as the **first parameter** preceding with the “**this**” keyword to the **GetWordCount()** method.

With the above two changes in place, now the **GetWordCount()** method becomes an extension method and hence we can call the **GetWordCount()**method in the same way as we call an instance method of a class.

##### ****The complete code is given below.****

**using** *System;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string sentence = "Welcome to Dotnet Tutorials";

**int** wordCount = sentence.GetWordCount**()**;

Console.WriteLine**(**$"Count : {wordCount}"**)**;

Console.ReadKey**()**;

**}**

**}**

**public** **static** **class** ExtensionHelper

**{**

**public** **static** **int** GetWordCount**(**this string str**)**

**{**

**if** **(**!String.IsNullOrEmpty**(**str**))**

**return** str.Split**(**' '**)**.Length;

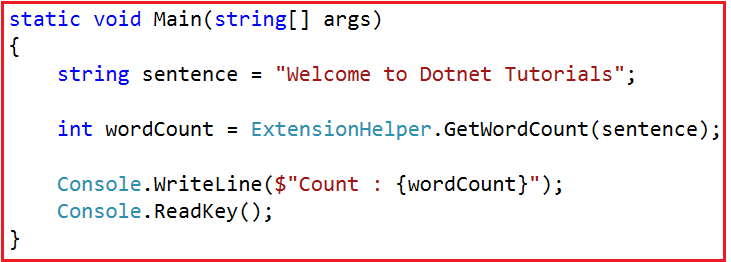
**return** 0;

**}**

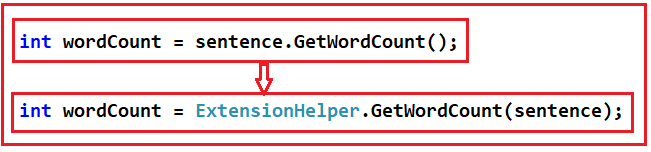
**}**

**}**

Now run the application and you will see the word count as expected in the console window. Here we are still able to call the **GetWordCount()** extension method using the wrapper class style syntax and also get the output as expected as shown below.



So, the point that I need to keep focus is, behind the scene this is how the extension methods are called internally.



That means, it is also possible to call the LINQ extension methods such as **select, where,** etc. using the wrapper class style syntax. As all the **LINQ extension methods** are implemented in the **Enumerable** class, So, the syntax to call those methods should looks as shown below.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** intList = new List**<int>** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

IEnumerable**<int>** EvenNumbers = Enumerable.Where**(**intList, n =**>** n % 2 == 0**)**;

Console.ReadKey**()**;

**}**

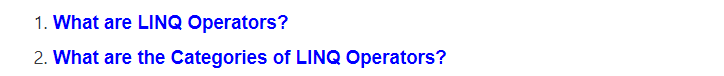
**}**

**}**

# LINQ Operators

## ****LINQ Operators in C#****

In this article, I am going to discuss the **LINQ operators** in C#. Please read our previous article where we discussed the [**LINQ Extension methods**](https://dotnettutorials.net/lesson/linq-extension-methods/) and how they are implemented. As part of this article, we are going to discuss the following two points.



##### ****What are LINQ Operators?****

The LINQ Operators are nothing but a set of extension methods that are used to write LINQ Query. These LINQ extension methods provide lots of very useful features which we can apply to the data source. Some of the features are filtering the data, sorting the data, grouping the data, etc.

##### ****What are the Categories of LINQ Operators?****

In LINQ, the operators are divided into the following categories:

1. Projection Operators
2. Ordering Operators
3. Filtering Operators
4. Set Operators
5. Quantifier Operators
6. Grouping Operators
7. Partitioning Operators
8. Equality Operators
9. Element Operators
10. Conversion Operators
11. Concatenation Operators
12. Aggregation Operators
13. Generation Operators
14. Join Operators
15. Custom Sequence Operators
16. Miscellaneous Operators

# LINQ Select Operator in C#

## ****LINQ Select Projection Operator in C# with Examples****

In this article, I am going to discuss the **LINQ Select Projection Operator in C#** with examples. Please read our previous article where we discussed what are [**LINQ Operators**](https://dotnettutorials.net/lesson/linq-operators/) and the different categories of LINQ Operators in C#. At the end of this article, you will understand the following pointers related to Linq Select Projection Operator in C#.

1. **What is Projection?**
2. **What are Projection Operators and Methods in LINQ?**
3. **How to use the Select Method?**
4. **Basic Selection of Data**
5. **How to Select Data to other classes and to anonymous type?**
6. **Performing Calculations on the selected data using LINQ Select Operator.**
7. **How to select data with index value?**

**Note:** We will discuss each example using both Query and Method syntax.

##### ****What is Projection?****

Projection is nothing but the mechanism which is used to select the data from a data source. You can select the data in the same form (i.e. the original data in its original state). It is also possible to create a new form of data by performing some operations on it.

##### ****What are Projection Methods or Operators available in LINQ?****

There are two methods available in projection. They are as follows

1. Select
2. SelectMany

In this article, we will discuss the Select Method and in the next article, we will discuss the SelectMany Method.

##### ****Select Operator:****

As we know the Select clausein SQL allows us to specify what columns we want to retrieve, whether you want to retrieve all the columns or some of the columns that you need to specify the select clause.

In the same way, the LINQ Select operator also allows us to specify what properties we want to retrieve, whether you want to retrieve all the properties or some of the properties that you need to specify in the select operator. The standard LINQ Select Operator also allows us to perform some calculations.

##### ****Example:****

Let us understand the select projection operator with some examples. Here we are going to use a console application. So first create a console application with the name **LINQDemo** (you can give any meaningful name). Then add a new class file with the name Employee.cs. Once you add the Employee.cs class file, then copy and paste the following in it.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string FirstName **{** **get**; **set**; **}**

**public** string LastName **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** **static** List**<**Employee**>** GetEmployees**()**

**{**

List**<**Employee**>** employees = new List**<**Employee**>**

**{**

new Employee **{**ID = 101, FirstName = "Preety", LastName = "Tiwary", Salary = 60000 **}**,

new Employee **{**ID = 102, FirstName = "Priyanka", LastName = "Dewangan", Salary = 70000 **}**,

new Employee **{**ID = 103, FirstName = "Hina", LastName = "Sharma", Salary = 80000 **}**,

new Employee **{**ID = 104, FirstName = "Anurag", LastName = "Mohanty", Salary = 90000 **}**,

new Employee **{**ID = 105, FirstName = "Sambit", LastName = "Satapathy", Salary = 100000 **}**,

new Employee **{**ID = 106, FirstName = "Sushanta", LastName = "Jena", Salary = 160000 **}**

**}**;

**return** employees;

**}**

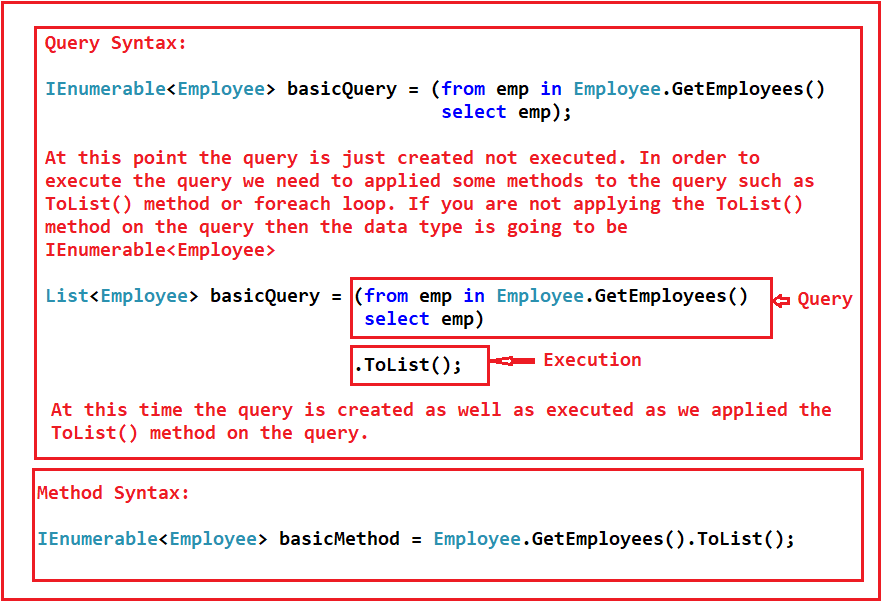
**}**

**}**

As you can see we have created the Employee class with the following four properties such as ID, FirstName, LastName, and Salary. We also created one static method which will return the list of employees which will act as our data source. Let us discuss some examples to understand the LINQ Select Operator.

##### ****Example1:****

**Select all the data from the data source using both Method and Query Syntax.**



##### ****The complete code is given below.****

Modify the Program class as shown below.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Query Syntax

List**<**Employee**>** basicQuery = **(from** emp in Employee.GetEmployees**()**

**select** emp**)**.ToList**()**;

**foreach** **(**Employee emp in basicQuery**)**

**{**

Console.WriteLine**(**$"ID : {emp.ID} Name : {emp.FirstName} {emp.LastName}"**)**;

**}**

//Using Method Syntax

IEnumerable**<**Employee**>** basicMethod = Employee.GetEmployees**()**.ToList**()**;

**foreach** **(**Employee emp in basicMethod**)**

**{**

Console.WriteLine**(**$"ID : {emp.ID} Name : {emp.FirstName} {emp.LastName}"**)**;

**}**

Console.ReadKey**()**;

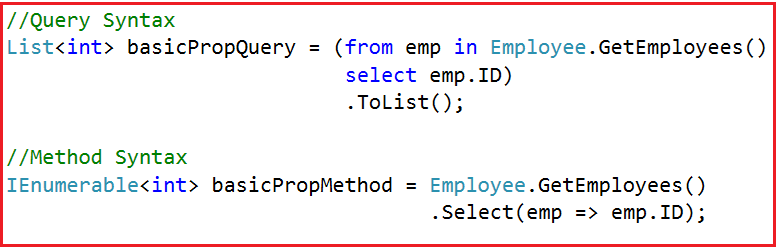
**}**

**}**

**}**

###### ****How to select a single property using Select Operator?****

Select all the Employee IDs only using both method and query syntax.



**Note:**In the Query Syntax, the data type of the basicPropQuery is List<int>, this is because of the ToList() method that we applied on the query syntax. And because of this ToList() method, the query is executed at that point only.

But in the case of Method syntax, we have not applied the ToList() method and this is the reason why the data type of the basicPropMethod variable is of IEnumerable<int> type. And more importantly, at that point, the query is just generated but not executed. When we use the basicPropMethod within the foreach loop, then at that time the query is executed.

##### ****The complete example is given below.****

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Query Syntax

List**<int>** basicPropQuery = **(from** emp in Employee.GetEmployees**()**

**select** emp.ID**)**

.ToList**()**;

**foreach** **(**var id in basicPropQuery**)**

**{**

Console.WriteLine**(**$"ID : {id}"**)**;

**}**

//Using Method Syntax

IEnumerable**<int>** basicPropMethod = Employee.GetEmployees**()**

.Select**(**emp =**>** emp.ID**)**;

**foreach** **(**var id in basicPropMethod**)**

**{**

Console.WriteLine**(**$"ID : {id}"**)**;

**}**

Console.ReadKey**()**;

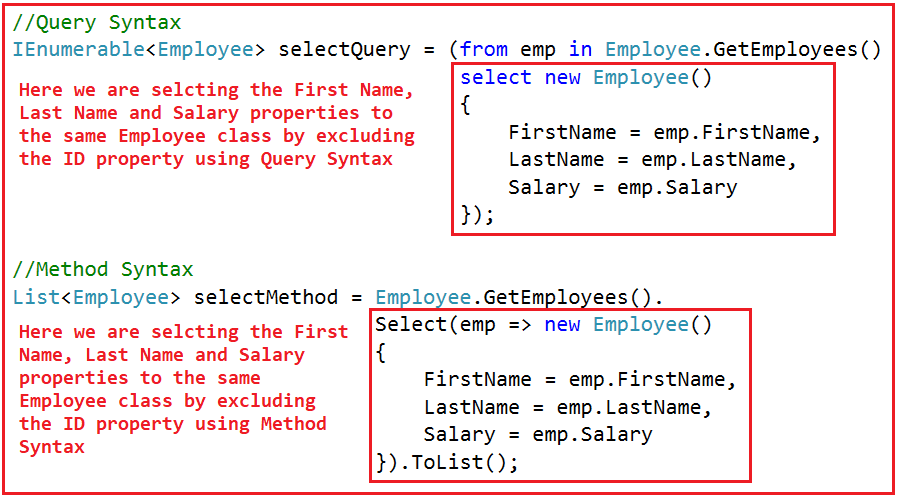
**}**

**}**

**}**

###### ****Example3:****

Our requirement is to select only the Employee First Name, Last Name, and Salary properties. We don’t require to select the ID property.



##### ****The Complete code is given below.****

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Query Syntax

IEnumerable**<**Employee**>** selectQuery = **(from** emp in Employee.GetEmployees**()**

**select** new Employee**()**

**{**

FirstName = emp.FirstName,

LastName = emp.LastName,

Salary = emp.Salary

**})**;

**foreach** **(**var emp in selectQuery**)**

**{**

Console.WriteLine**(**$" Name : {emp.FirstName} {emp.LastName} Salary : {emp.Salary} "**)**;

**}**

//Method Syntax

List**<**Employee**>** selectMethod = Employee.GetEmployees**()**.

Select**(**emp =**>** new Employee**()**

**{**

FirstName = emp.FirstName,

LastName = emp.LastName,

Salary = emp.Salary

**})**.ToList**()**;

**foreach** **(**var emp in selectMethod**)**

**{**

Console.WriteLine**(**$" Name : {emp.FirstName} {emp.LastName} Salary : {emp.Salary} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

##### ****How to Select Data to another class using LINQ Projection Operator?****

It is also possible to select the data to another class using the LINQ Select operator. In our previous example, we have selected the First Name, Last Name, and Salary properties to the same Employee class. Let us create a new class with the above three properties. So, add a new class file with the name EmployeeBasicInfo.cs and then copy and paste the following code.

**namespace** *LINQDemo*

**{**

**public** **class** EmployeeBasicInfo

**{**

**public** string FirstName **{** **get**; **set**; **}**

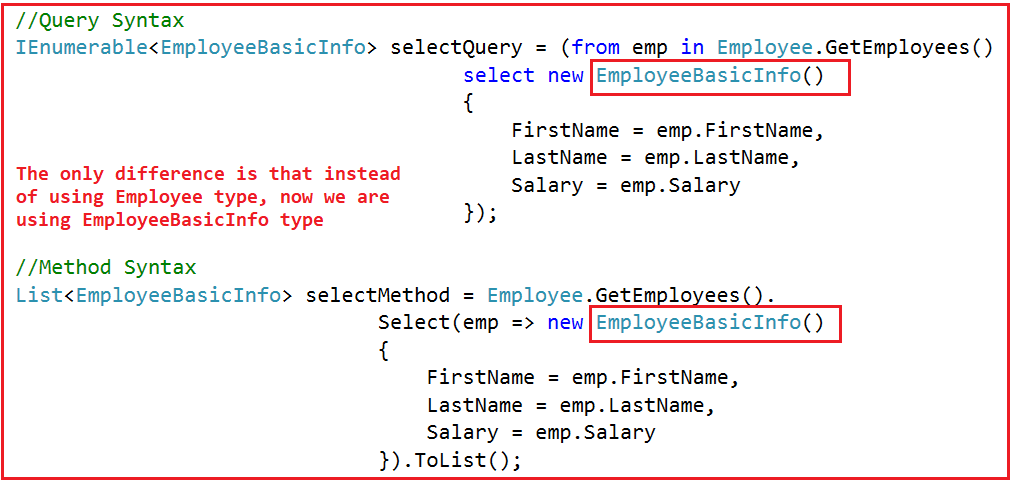
**public** string LastName **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**}**

Now what we need to do here is, we need to project the First Name, Last Name, and Salary properties to the above newly created **EmployeeBasicInfo** class.



##### ****The complete example is given below.****

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Query Syntax

IEnumerable**<**EmployeeBasicInfo**>** selectQuery = **(from** emp in Employee.GetEmployees**()**

**select** new EmployeeBasicInfo**()**

**{**

FirstName = emp.FirstName,

LastName = emp.LastName,

Salary = emp.Salary

**})**;

**foreach** **(**var emp in selectQuery**)**

**{**

Console.WriteLine**(**$" Name : {emp.FirstName} {emp.LastName} Salary : {emp.Salary} "**)**;

**}**

//Method Syntax

List**<**EmployeeBasicInfo**>** selectMethod = Employee.GetEmployees**()**.

Select**(**emp =**>** new EmployeeBasicInfo**()**

**{**

FirstName = emp.FirstName,

LastName = emp.LastName,

Salary = emp.Salary

**})**.ToList**()**;

**foreach** **(**var emp in selectMethod**)**

**{**

Console.WriteLine**(**$" Name : {emp.FirstName} {emp.LastName} Salary : {emp.Salary} "**)**;

**}**

Console.ReadKey**()**;

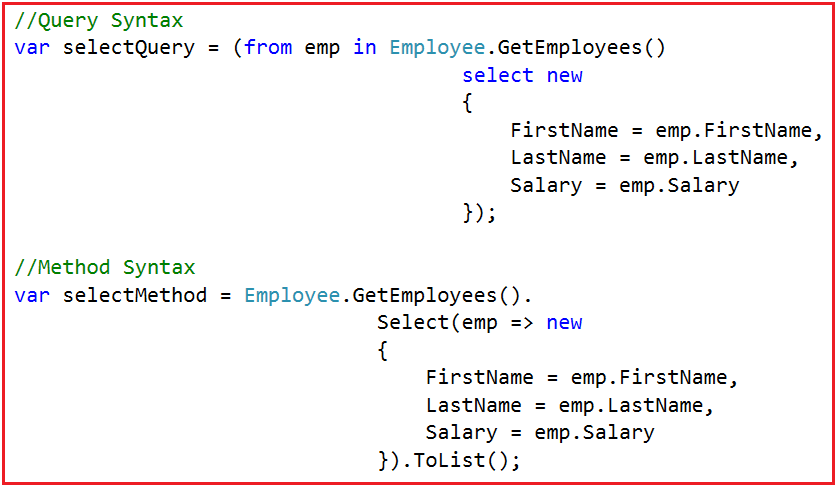
**}**

**}**

**}**

##### ****How to Select Data to Anonymous Type using LINQ Select Operator?****

Instead of projecting the data to a particular type like Employee or EmployeeBasicInfo, we can also project the data to an anonymous type in LINQ.



##### ****The complete code is given below.****

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Query Syntax

var selectQuery = **(from** emp in Employee.GetEmployees**()**

**select** new

**{**

FirstName = emp.FirstName,

LastName = emp.LastName,

Salary = emp.Salary

**})**;

**foreach** **(**var emp in selectQuery**)**

**{**

Console.WriteLine**(**$" Name : {emp.FirstName} {emp.LastName} Salary : {emp.Salary} "**)**;

**}**

//Method Syntax

var selectMethod = Employee.GetEmployees**()**.

Select**(**emp =**>** new

**{**

FirstName = emp.FirstName,

LastName = emp.LastName,

Salary = emp.Salary

**})**.ToList**()**;

**foreach** **(**var emp in selectMethod**)**

**{**

Console.WriteLine**(**$" Name : {emp.FirstName} {emp.LastName} Salary : {emp.Salary} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

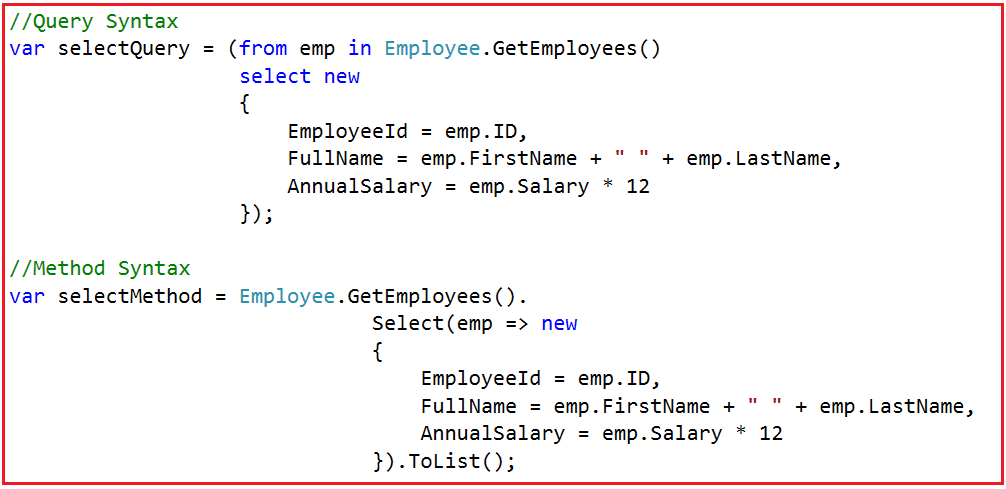
**}**

##### ****How to perform calculations on data selected using the LINQ Select Operator?****

Let me first explain what we want to achieve. We want to perform the following calculations

1. **AnnualSalary = Salary\*12**
2. **FullName = FirstName + ” ” + LastName**

Then we need to select the ID, AnnualSalary, and FullName to an anonymous type.



##### ****The Complete example is given below.****

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Query Syntax

var selectQuery = **(from** emp in Employee.GetEmployees**()**

**select** new

**{**

EmployeeId = emp.ID,

FullName = emp.FirstName + " " + emp.LastName,

AnnualSalary = emp.Salary \* 12

**})**;

**foreach** **(**var emp in selectQuery**)**

**{**

Console.WriteLine**(**$" ID {emp.EmployeeId} Name : {emp.FullName} Annual Salary : {emp.AnnualSalary} "**)**;

**}**

//Method Syntax

var selectMethod = Employee.GetEmployees**()**.

Select**(**emp =**>** new

**{**

EmployeeId = emp.ID,

FullName = emp.FirstName + " " + emp.LastName,

AnnualSalary = emp.Salary \* 12

**})**.ToList**()**;

**foreach** **(**var emp in selectMethod**)**

**{**

Console.WriteLine**(**$" ID {emp.EmployeeId} Name : {emp.FullName} Annual Salary : {emp.AnnualSalary} "**)**;

**}**

Console.ReadKey**()**;

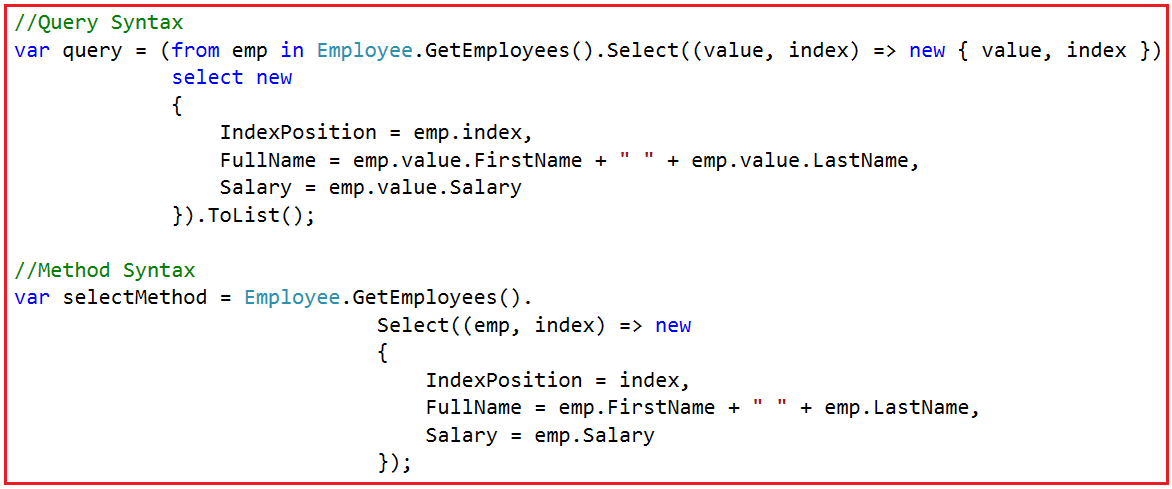
**}**

**}**

**}**

##### ****How to select data with index value?****

It is also possible to select values using an integral index. The index is 0 based.



##### ****The Complete code is given below.****

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Query Syntax

var query = **(from** emp in Employee.GetEmployees**()**.Select**((value**, index**)** =**>** new **{** **value**, index **})**

**select** new

**{**

IndexPosition = emp.index,

FullName = emp.value.FirstName + " " + emp.value.LastName,

Salary = emp.value.Salary

**})**.ToList**()**;

**foreach** **(**var emp in query**)**

**{**

Console.WriteLine**(**$" Position {emp.IndexPosition} Name : {emp.FullName} Salary : {emp.Salary} "**)**;

**}**

//Method Syntax

var selectMethod = Employee.GetEmployees**()**.

Select**((**emp, index**)** =**>** new

**{**

IndexPosition = index,

FullName = emp.FirstName + " " + emp.LastName,

Salary = emp.Salary

**})**;

**foreach** **(**var emp in selectMethod**)**

**{**

Console.WriteLine**(**$" Position {emp.IndexPosition} Name : {emp.FullName} Salary : {emp.Salary} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

# LINQ SelectMany in C#

## ****LINQ SelectMany in C# with Examples****

In this article, I am going to discuss the **LINQ SelectMany in C#** with examples. Please read our previous article where we discussed the [**Select Operator in C#**](https://dotnettutorials.net/lesson/linq-select-operator/) with some examples. The Linq SelectMany Method belongs to the projection category operator. As part of this article, we are going to discuss the following pointers in details.

1. **What is Linq SelectMany?**
2. **Examples using both query and method syntax in C#.**

##### ****What is Linq SelectMany?****

The SelectMany in LINQ is used to project each element of a sequence to an **IEnumerable<T>** and then flatten the resulting sequences into one sequence. That means the SelectMany operator combines the records from a sequence of results and then converts it into one result. If this is not clear at the moment, then don’t worry we will see it in practice.

##### ****Example1:****

Let us first write a program and see the output then we will discuss how it will work.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** nameList =new List**<**string**>(){**"Pranaya", "Kumar" **}**;

IEnumerable**<char>** methodSyntax = nameList.SelectMany**(**x =**>** x**)**;

**foreach(char** c in methodSyntax**)**

**{**

Console.Write**(**c + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

##### ****Let us understand the above code:****

In the above code, we see that the SelectMany method returns an **IEnumerable<char>**. This is because the SelectMany method returns all the elements from the sequence. Here the List is the sequence. And the list contains the strings. Here the list contains two strings. So, the SelectMany method fetches all the characters from the above two strings and then converts it into one sequence i.e. **IEnumerable<char>**.

So, when we execute the above program, it will give us the following output.

SelectMany in LINQ

##### ****Linq SelectMany Using Query Syntax in C#:****

The most important point is that there is no such SelectMany operator available in LINQ to write query syntax. But we can achieve this by writing multiple “from clause” in the query as shown in the below example.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** nameList =new List**<**string**>(){**"Pranaya", "Kumar" **}**;

IEnumerable**<char>** querySyntax = **from** str in nameList

**from** ch in str

**select** ch;

**foreach** **(char** c in querySyntax**)**

**{**

Console.Write**(**c + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now, execute the program and you will see the same output as method syntax as expected.

##### ****Example2: LINQ SelectMany with Complex Type in C#****

Let us create a class file with the name Student.cs and then copy and paste the following code.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Email **{** **get**; **set**; **}**

**public** List**<**string**>** Programming **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetStudents**()**

**{**

**return** new List**<**Student**>()**

**{**

new Student**(){**ID = 1, Name = "James", Email = "James@j.com", Programming = new List**<**string**>()** **{** "C#", "Jave", "C++"**}** **}**,

new Student**(){**ID = 2, Name = "Sam", Email = "Sara@j.com", Programming = new List**<**string**>()** **{** "WCF", "SQL Server", "C#" **}}**,

new Student**(){**ID = 3, Name = "Patrik", Email = "Patrik@j.com", Programming = new List**<**string**>()** **{** "MVC", "Jave", "LINQ"**}** **}**,

new Student**(){**ID = 4, Name = "Sara", Email = "Sara@j.com", Programming = new List**<**string**>()** **{** "ADO.NET", "C#", "LINQ" **}** **}**

**}**;

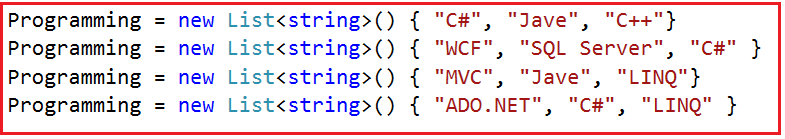
**}**

**}**

**}**

As you can see we have created the Student class with four properties. Please remember the Programming property returns list<string>. Here we also created one method which will return the List of students which will be going to act our data source.

We need to Projects all programming strings of all the students to a single **IEnumerable<string>**. As you can see, we have 4 students, so there will be 4 IEnumerable<string> sequences, which then we need to flattened to form a single sequence i.e. a single IEnumerable<string> sequence.



**The code is given below.**

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

List**<**string**>** MethodSyntax = Student.GetStudents**()**.SelectMany**(**std =**>** std.Programming**)**.ToList**()**;

//Using Query Syntax

IEnumerable**<**string**>** QuerySyntax = **from** std in Student.GetStudents**()**

**from** program in std.Programming

**select** program;

//Printing the values

**foreach** **(**string program in MethodSyntax**)**

**{**

Console.WriteLine**(**program**)**;

**}**

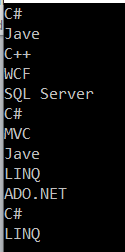
Console.ReadKey**()**;

**}**

**}**

**}**

When we execute the program, it gives us the following output.



##### ****Example3:****

In our previous example, we get the output as expected but with the duplicate program names. If you want only the distinct program names then you need to apply the distinct method on the result set as shown in the below examples.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

List**<**string**>** MethodSyntax = Student.GetStudents**()**

.SelectMany**(**std =**>** std.Programming**)**

.Distinct**()**

.ToList**()**;

//Using Query Syntax

IEnumerable**<**string**>** QuerySyntax = **(from** std in Student.GetStudents**()**

**from** program in std.Programming

**select** program**)**.Distinct**()**.ToList**()**;

//Printing the values

**foreach** **(**string program in QuerySyntax**)**

**{**

Console.WriteLine**(**program**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now execute the program and you will see that it will remove the duplicate program names from the sequence.

##### ****Example4:****

Now we need to retrieve the student name along with the program language name.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MethodSyntax = Student.GetStudents**()**

.SelectMany**(**std =**>** std.Programming,

**(**student, program**)** =**>** new

**{**

StudentName = student.Name,

ProgramName = program

**}**

**)**

.ToList**()**;

//Using Query Syntax

var QuerySyntax = **(from** std in Student.GetStudents**()**

**from** program in std.Programming

**select** new **{**

StudentName = std.Name,

ProgramName = program

**})**.ToList**()**;

//Printing the values

**foreach** **(**var item in QuerySyntax**)**

**{**

Console.WriteLine**(**item.StudentName + " => " + item.ProgramName**)**;

**}**

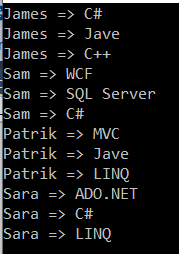
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



# Where Filtering Operators in LINQ

## ****Where Filtering Operators in LINQ with C# Examples****

In this article, I am going to discuss **Where** **Filtering Operators in LINQ** with examples. Please read our previous article before proceeding to this article where we discussed the [**SelectMany Projection operator in C#**](https://dotnettutorials.net/lesson/selectmany-in-linq/) with some examples. At the end of this article, you will understand the following concepts.

1. **What is Filtering**
2. **What are the Filtering methods available in LINQ?**
3. **The “Where” operator and its examples using both method and query syntax**
4. **What is a Predicate?**

##### ****What is Filtering?****

Filtering is nothing but the process to get only those elements from a data source that satisfied the given condition. It is also possible to fetch the data from a data source with more than one condition as per our business requirement.

**For example:**

1. Employees having a salary greater than 50000.
2. Students Having Marks greater than 80% from a particular batch.
3. Employees having experience more than 6 Years and the department is IT, etc.

##### ****What are the Filtering methods available in LINQ?****

There are two methods provided by LINQ in C# which are used for filtering. They are as follows

1. Where
2. OfType

In this article, I am going to discuss the “**Where”** operator in detail. In the next article, I  will discuss the **OfType** operator with some examples.

##### ****Where Filtering Operators in LINQ:****

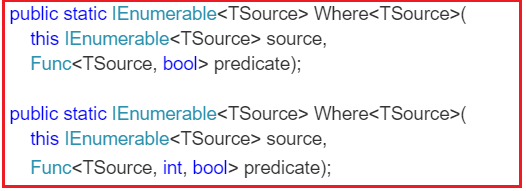
The standard query operator “**where**” comes under the **Filtering Operators** category in LINQ.

We need to use the where standard query operator in LINQ when we need to filter the data from a data source based on some condition(s) just like as we did in SQL using the where clause. So in simple words, we can say that it is used to filter the data from a data source based on some condition(s).

The “where” always expects at least one condition and we can specify the condition(s) using predicates. The conditions can be written using the following symbols

**==, >=, <=, &&, ||, >, <, etc.**

There are two overloaded versions of the “where” operator is available. They are as follows



As you can see in the above signatures, the methods are implemented as extension methods on IEnumerable<T> interface. The methods predicate as a parameter. So let us first discuss what a predicate is?

##### ****What is a Predicate?****

A predicate is nothing but a function that is used to test each and every element for a given condition. Let us understand this with an example.

In the below example, the Lambda expression (**num => num > 5**) runs for each and every element present in the “**intList**” collection. Then it will check, whether the number is greater than 5 or not. If the number value is greater than 5, then a boolean value true is returned otherwise false.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** intList = new List**<int>** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

//Method Syntax

IEnumerable**<int>** filteredData = intList.Where**(**num =**>** num **>** 5**)**;

//Query Syntax

IEnumerable**<int>** filteredResult = **from** num in intList

**where** num **>** 5

**select** num;

**foreach** **(int** number in filteredData**)**

**{**

Console.WriteLine**(**number**)**;

**}**

Console.ReadKey**()**;

**}**

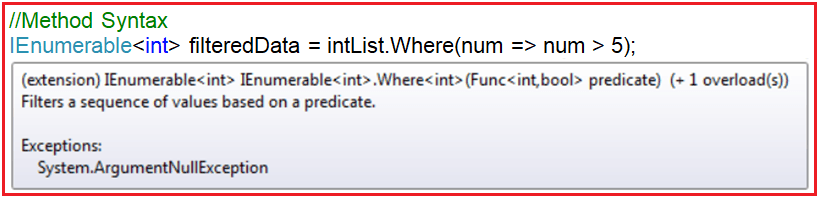
**}**

**}**

When we execute the above program, it will give us the following output as expected.

C# Where Filtering Operators in LINQ

When you hover the mouse over the **WHERE**extension method in the above example, then visual studio intelligence shows the following.



As you can see in the above image, the predicate (Func<int, bool> predicate) expects one input parameter of type integer and it returns a boolean value. The Func is a generic delegate that takes one or more input parameters and returns one out parameter. The last parameter is considered as the return value. The return type is mandatory but the input parameter is not.

If you are new to Generic Delegate, then I strongly recommended you to read the following article where we discussed the Generic Delegates in C# with examples.

[**https://dotnettutorials.net/lesson/generic-delegates-csharp/**](https://dotnettutorials.net/lesson/generic-delegates-csharp/)

The lambda expression that we passed to the where extension method in the above example operates on integer data type and should return a boolean value else we will get a compile-time error.

So the following line of code from the above example

**IEnumerable<int> filteredData = intList.Where(num => num > 5);**

Can be rewritten as shown below

**Func<int, bool> predicate = i => i > 5;**

**IEnumerable<int> filteredData = intList.Where(predicate);**

It should give the same output as expected. You can also create a separate function as shown below which also work as expected.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** intList = new List**<int>** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

//Method Syntax

IEnumerable**<int>** filteredData = intList.Where**(**num =**>** CheckNumber**(**num**))**;

**foreach** **(int** number in filteredData**)**

**{**

Console.WriteLine**(**number**)**;

**}**

Console.ReadKey**()**;

**}**

**public** **static** **bool** CheckNumber**(int** number**)**

**{**

**if** **(**number **>** 5**)**

**{**

**return** **true**;

**}**

**else**

**{**

**return** **false**;

**}**

**}**

**}**

**}**

##### ****Example2:****

In the second overloaded version of the “where” extension method, the int parameter of the predicate function represents the index position of the source element.

**public** **static** IEnumerable**<**TSource**>** Where**<**TSource**>(**

this IEnumerable**<**TSource**>** source,

Func**<**TSource, **int**, **bool>** predicate**)**;

##### ****Let us see an example to understand this.****

Here we need to filter only the odd numbers i.e. the numbers which are not divisible by 2. Along with the numbers we also need to fetch the index position of the number. The index is 0 based.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** intList = new List**<int>** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

//Method Syntax

var OddNumbersWithIndexPosition = intList.Select**((**num, index**)** =**>** new

**{**

Numbers = num,

IndexPosition = index

**})**.Where**(**x =**>** x.Numbers % 2 != 0**)**

.Select**(**data =**>** new

**{**

Number = data.Numbers,

IndexPosition = data.IndexPosition

**})**;

**foreach** **(**var item in OddNumbersWithIndexPosition**)**

**{**

Console.WriteLine**(**$"IndexPosition :{item.IndexPosition} , Value : {item.Number}"**)**;

**}**

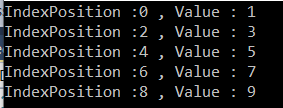
Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and you will see the odd numbers along with their index position as shown below.



##### ****Let’s rewrite the same example using query syntax.****

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** intList = new List**<int>** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

//Query Syntax

var OddNumbersWithIndexPosition = **from** number in intList.Select**((**num, index**)** =**>** new **{**Numbers = num, IndexPosition = index **})**

**where** number.Numbers % 2 != 0

**select** new

**{**

Number = number.Numbers,

IndexPosition = number.IndexPosition

**}**;

**foreach** **(**var item in OddNumbersWithIndexPosition**)**

**{**

Console.WriteLine**(**$"IndexPosition :{item.IndexPosition} , Value : {item.Number}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it will also give the same output as method syntax output.

##### ****Complex Examples:****

Let us see how to use where condition with complex type. We are going to use the following Employee class. So, create a class file with the name **Employee.cs** and then copy and paste the following code.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** List**<**string**>** Technology **{** **get**; **set**; **}**

**public** **static** List**<**Employee**>** GetEmployees**()**

**{**

List**<**Employee**>** employees = new List**<**Employee**>()**

**{**

new Employee **{**ID = 101, Name = "Preety", Gender = "Female", Salary = 60000,

Technology = new List**<**string**>()** **{**"C#", "Jave", "C++"**}** **}**,

new Employee **{**ID = 102, Name = "Priyanka", Gender = "Female", Salary = 50000,

Technology =new List**<**string**>()** **{** "WCF", "SQL Server", "C#" **}** **}**,

new Employee **{**ID = 103, Name = "Hina", Gender = "Female", Salary = 40000,

Technology =new List**<**string**>()** **{** "MVC", "Jave", "LINQ"**}}**,

new Employee **{**ID = 104, Name = "Anurag", Gender = "Male", Salary = 450000**}**,

new Employee **{**ID = 105, Name = "Sambit", Gender = "Male", Salary = 550000**}**,

new Employee **{**ID = 106, Name = "Sushanta", Gender = "Male", Salary = 700000,

Technology =new List**<**string**>()** **{** "ADO.NET", "C#", "LINQ" **}}**

**}**;

**return** employees;

**}**

**}**

**}**

As we can see we created the Employee class with five properties i.e. ID, Name, Gender, Salary, and Technology. As you can see we have also created one method which will return the list of all employees which will be going to our data source.

##### ****Example3:****

We need to fetch all the employees whose salary is greater than 50000.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Query Syntax

var QuerySyntax = **from** employee in Employee.GetEmployees**()**

**where** employee.Salary **>** 50000

**select** employee;

//Method Syntax

var MethodSyntax = Employee.GetEmployees**()**

.Where**(**emp =**>** emp.Salary **>** 50000**)**;

**foreach** **(**var emp in QuerySyntax**)**

**{**

Console.WriteLine**(**$"Name : {emp.Name}, Salary : {emp.Salary}, Gender : {emp.Gender}"**)**;

**if(**emp.Technology != **null** && emp.Technology.Count**()** **>** 0**)**

**{**

Console.Write**(**" Technology : "**)**;

**foreach** **(**var tech in emp.Technology**)**

**{**

Console.Write**(**tech + " "**)**;

**}**

Console.WriteLine**()**;

**}**

**else**

**{**

Console.WriteLine**(**" Technology Not Available "**)**;

**}**

**}**

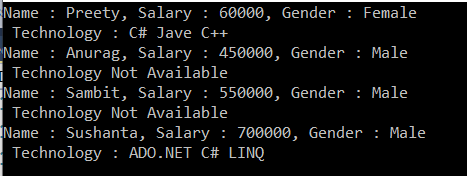
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Example4: Using multiple conditions****

We need to fetch all the employee whose gender is Male and Salary is greater than 500000.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Query Syntax

var QuerySyntax = **from** employee in Employee.GetEmployees**()**

**where** employee.Salary **>** 500000 && employee.Gender == "Male"

**select** employee;

//Method Syntax

var MethodSyntax = Employee.GetEmployees**()**

.Where**(**emp =**>** emp.Salary **>** 500000 && emp.Gender == "Male"**)**

.ToList**()**;

**foreach** **(**var emp in MethodSyntax**)**

**{**

Console.WriteLine**(**$"Name : {emp.Name}, Salary : {emp.Salary}, Gender : {emp.Gender}"**)**;

**if(**emp.Technology != **null** && emp.Technology.Count**()** **>** 0**)**

**{**

Console.Write**(**" Technology : "**)**;

**foreach** **(**var tech in emp.Technology**)**

**{**

Console.Write**(**tech + " "**)**;

**}**

Console.WriteLine**()**;

**}**

**else**

**{**

Console.WriteLine**(**" Technology Not Available "**)**;

**}**

**}**

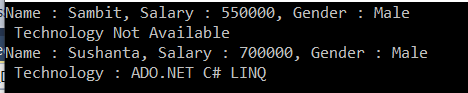
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Example5:****

Multiple conditions with the custom operation and projecting the data to an anonymous type:

**Requirement:**

All the employees whose salary is greater than or equal to 50000 and technology should not be null.

We need to fetch the following information to an anonymous type.

1. Name as it is
2. Gender as it is
3. MonthlySalary = Salary / 12

**The code is given below.**

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Query Syntax

var QuerySyntax = **(from** employee in Employee.GetEmployees**()**

**where** employee.Salary **>**= 50000 && employee.Technology != **null**

**select** new **{**

EmployeeName = employee.Name,

Gender = employee.Gender,

MonthlySalary = employee.Salary / 12

**})**.ToList**()**;

//Method Syntax

var MethodSyntax = Employee.GetEmployees**()**

.Where**(**emp =**>** emp.Salary **>**= 50000 && emp.Technology != **null)**

.Select**(**emp =**>** new **{**

EmployeeName = emp.Name,

Gender = emp.Gender,

MonthlySalary = emp.Salary / 12

**})**

.ToList**()**;

**foreach** **(**var emp in QuerySyntax**)**

**{**

Console.WriteLine**(**$"Name : {emp.EmployeeName}, Gender : {emp.Gender}, Monthly Salary : {emp.MonthlySalary}"**)**;

**}**

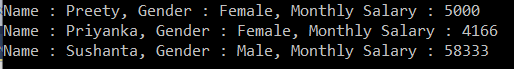
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Example6: Fetching elements along with the Index position****

Here we need to fetch all the employees whose Gender is Male and Salary is greater than 500000 along with their index position to an anonymous type.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Query Syntax

var QuerySyntax = **(from** data in Employee.GetEmployees**()**.Select**((**Data, index**)** =**>** new **{** employee = Data, Index = index **})**

**where** data.employee.Salary **>**= 500000 && data.employee.Gender == "Male"

**select** new

**{**

EmployeeName = data.employee.Name,

Gender = data.employee.Gender,

Salary = data.employee.Salary,

IndexPosition = data.Index

**})**.ToList**()**;

//Method Syntax

var MethodSyntax = Employee.GetEmployees**()**.Select**((**Data, index**)** =**>** new **{** employee = Data, Index = index **})**

.Where**(**emp =**>** emp.employee.Salary **>**= 500000 && emp.employee.Gender == "Male"**)**

.Select**(**emp =**>** new

**{**

EmployeeName = emp.employee.Name,

Gender = emp.employee.Gender,

Salary = emp.employee.Salary,

IndexPosition = emp.Index

**})**

.ToList**()**;

**foreach** **(**var emp in QuerySyntax**)**

**{**

Console.WriteLine**(**$"Position : {emp.IndexPosition} Name : {emp.EmployeeName}, Gender : {emp.Gender}, Salary : {emp.Salary}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

where with multiple condition along with index in C# linq

# OfType Operator in LINQ

## ****OfType Operator in LINQ with Examples****

In this article, I am going to discuss the **OfType Operator in LINQ** with examples. Please read our previous article before proceeding to this article where we discussed the [**where filtering operator in C#**](https://dotnettutorials.net/lesson/where-filtering-operators-in-linq/) with some examples. The **OfType Operator in C#** belongs to the filtering category of LINQ operators. As part of this article, we are going to discuss the following pointers in detail.

1. **What is OfType Operator in LINQ?**
2. **Examples using both Method and Query syntax.**
3. **Difference between OfType and is Operator in C#.NET.**

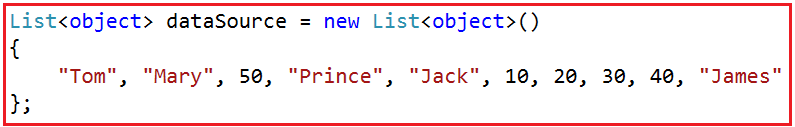
##### ****What is OfType Operator in LINQ?****

The OfType Operator in LINQ is used to filter specific type data from a data source based on the data type we passed to this operator. For example, if we have a collection that stores both integer and string values and if we need to fetch either only the integer values or only the string values from that collection then we need to use the **OfType** operator.

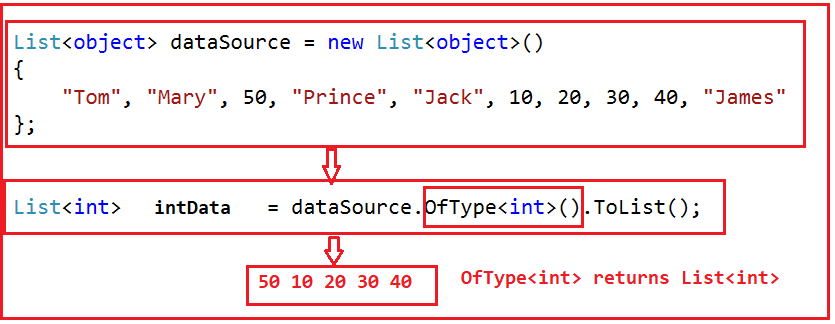
Following is the signature of the **OfType** method. It is implemented as a generic type, so there are no overloaded versions available for this method. This method can take any type of data type and then fetch the specified data type values from the collection.

**public static IEnumerable<TResult> OfType<TResult>(this IEnumerable source);**

Let say, we have a collection of type object. As we know the object class is the superclass of all data types so we can store any type of values in it like below.



Now our requirement is to fetch all the integer values from the collection by ignoring the string values. We can achieve this very easily by using the OfType operator in C# like below.



**The complete code is given below.**

**OfType Operator Using Method syntax in C#.NET:**

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<object>** dataSource = new List**<object>()**

**{**

"Tom", "Mary", 50, "Prince", "Jack", 10, 20, 30, 40, "James"

**}**;

List**<int>** intData = dataSource.OfType**<int>()**.ToList**()**;

**foreach** **(int** number in intData**)**

**{**

Console.Write**(**number + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

OfType Operator in C#

##### ****OfType Operator Using Query Syntax in C#.NET:****

There is no such OfType operator available in query syntax. Instead “is” operator is available which we can use to filter data based on a type from a data source. In the following example, the collection contains both string and integer values and we need to fetch only the string values.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<object>** dataSource = new List**<object>()**

**{**

"Tom", "Mary", 50, "Prince", "Jack", 10, 20, 30, 40, "James"

**}**;

var stringData = **(from** name in dataSource

**where** name **is** string

**select** name**)**.ToList**()**;

**foreach** **(**string name in stringData**)**

**{**

Console.Write**(**name + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

C# OfType Operator using LINQ

##### ****OfType and is Operator with a condition in C#.NET:****

Let say, we want to retrieve all the names whose length is greater than 3 and all the integer number which is greater than 30. Here we will use OfType operator to retrieve all the names and “is” operator to retrieve all the integers along with the required conditions.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<object>** dataSource = new List**<object>()**

**{**

"Tom", "Mary", 50, "Prince", "Jack", 10, 20, 30, 40, "James"

**}**;

//Using Method Syntax

var intData = dataSource.OfType**<int>()**.Where**(**num =**>** num **>** 30**)**.ToList**()**;

**foreach** **(int** number in intData**)**

**{**

Console.Write**(**number + " "**)**;

**}**

Console.WriteLine**()**;

//Using Qyery Syntax

var stringData = **(from** name in dataSource

**where** name **is** string && name.ToString**()**.Length **>** 3

**select** name**)**.ToList**()**;

**foreach** **(**string name in stringData**)**

**{**

Console.Write**(**name + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

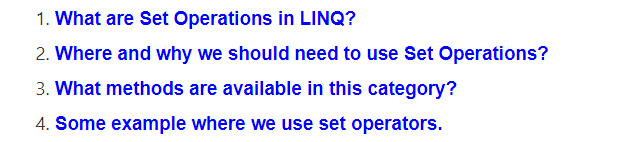
**Output:**

OfType Operator

# Set Operators in LINQ

## ****Set Operators in LINQ using C#****

In this article, I am going to discuss **Set Operators in LINQ**using C#. Please read our previous article where we discussed [**OfType operators in LINQ**](https://dotnettutorials.net/lesson/oftype-operator-in-linq/) with some examples. As part of this article, we are going to discuss the following concepts.



**Note:** The most important thing that we always need to remember is, before learning something first we need to understand where and why we need to use that thing. With this keep in mind let’s proceed to this article.

##### ****Set Operators in LINQ:****

The Set Operators in LINQ are used to produce the result set based on the presence and absence of elements within the same or different data sources. That means these operations are performed either on a single data source or on multiple data sources and in the output some of the data are present and some of the data are absent. If this is not clear at the moment then don’t worry we will discuss each and everything with examples.

##### ****Examples of Set Operations:****

Let us discuss some of the examples where we need to use the set operations.

1. If we need to select the distinct records from a data source (No Duplicate Records) then we need to use Set Operators.
2. Suppose we need to select all the Employees of a company except a particular department then you need to use Set Operations.
3. Another example maybe if you have multiple classes and you want only to select all the toppers from all the classes then also you need to use Set Operations.
4. Suppose we have different data sources with similar structure and if we want to combine all the data sources into a single data source then we need to use Set Operations.

##### ****LINQ Set Operation Methods in C#:****

The following LINQ Extension Methods are provided to perform set operations in C#.

**Distinct:** We need to use the Distinct() method when we want to remove the duplicate data or records from a data source. This method operates on a single data source.

**Except:** We need to use the Except() LINQ Extension method when we want to return all the elements from the first data source which do not exists in the second data source. This method operates on two data sources.

**Intersect:** This method is used to return the common elements from both the data sources i.e. the elements which exist in both the data set are going to returns as output.

**Union:** This method is used to return all the elements which are present in either of the data sources. That means it combines the data from both the data sources and produce a single result set.

# LINQ Distinct Method in C#

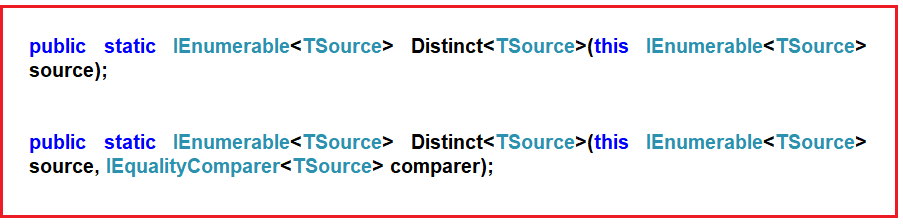
## ****LINQ Distinct Method in C# with Examples****

In this article, I am going to discuss the **LINQ Distinct Method in C#** using examples. Please read our previous article where we discussed the basics of [**LINQ Set Operators**](https://dotnettutorials.net/lesson/set-operation-linq/). At the end of this article, you will understand the following the Linq Distinct Query in details with the following pointers.

1. **What is LINQ Distinct Method in C#?**
2. **Examples of LINQ Distinct Method using both Method and Query Syntax**
3. **How to implement IEqualityComparer?**

##### ****What us LINQ Distinct Method in C#?****

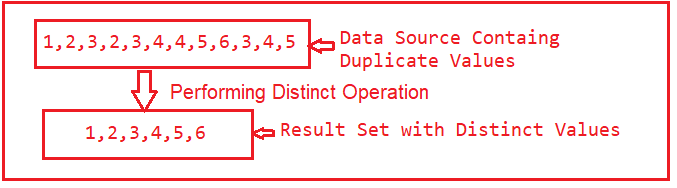
The **LINQ Distinct Method in C#** is used to return the distinct elements from a single data source. There are two overloaded versions available for the Distinct Method as shown below.



The one and the only difference between these two methods is the second overloaded version takes an **IEqualityComparer** as input that means the Distinct Operator can also be used with Comparer also. If this is not clear at the moment, don’t worry we will cover the use of the Comparer in this article also.

##### ****Example1: LINQ Distinct Method on Value Type****

Here we have an integer collection that contains duplicate integer values. Our requirement is to fetch remove the duplicate values and return only the distinct values as shown below.



The following program shows how to get the distinct integer values using both Method and Query syntax:

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** intCollection = new List**<int>()**

**{**

1,2,3,2,3,4,4,5,6,3,4,5

**}**;

//Using Method Syntax

var MS = intCollection.Distinct**()**;

//Using Query Syntax

var QS = **(from** num in intCollection

**select** num**)**.Distinct**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

https://dotnettutorials.net/wp-content/uploads/2019/05/word-image-12.png

Note: In query syntax, there is no such operator call distinct, so here we need to use both query and method syntax to achieve the same.

##### ****Example2:****

Here we have a string array of names and we need to return the distinct names from the collection.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** namesArray = **{** "Priyanka", "HINA", "hina", "Anurag", "Anurag", "ABC", "abc" **}**;

var distinctNames = namesArray.Distinct**()**;

**foreach** **(**var name in distinctNames**)**

**{**

Console.WriteLine**(**name**)**;

**}**

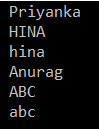
Console.ReadKey**()**;

**}**

**}**

**}**

When we execute the above program, it gives us the below output.



As you can see the name Hina and Abc have appeared twice. This is because the default comparer, which is used to filter the duplicate values is case-sensitive.

If you want to make the comparison to be case-insensitive then you need to use the other overloaded version which takes **IEqualityComparer** as an argument. So here we need to pass a class which must implement the **IEqualityComparer** interface.

So let’s modify the program as shown below. As you can see here we are passing **StringComparer** as an argument to the Distinct method.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** namesArray = **{** "Priyanka", "HINA", "hina", "Anurag", "Anurag", "ABC", "abc" **}**;

var distinctNames = namesArray.Distinct**(**StringComparer.OrdinalIgnoreCase**)**;

**foreach** **(**var name in distinctNames**)**

**{**

Console.WriteLine**(**name**)**;

**}**

Console.ReadKey**()**;

**}**

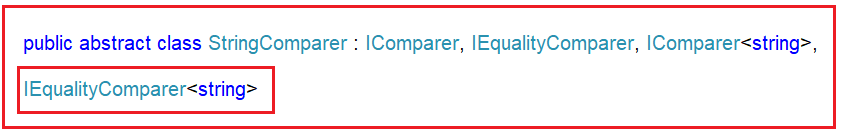
**}**

**}**

Now run the application and it should display the distinct names as shown below.

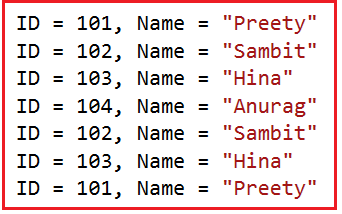


If we go to the definition of **StringComparer** class then we can observe that this class implements the **IEqualityComparer** interface as shown below.



##### ****LINQ Distinct Operation with Complex Type:****

The LINQ Distinct Method works in a different manner with complex types like Employee, Product, Student, etc. Let us understand with an example. We are going to work with the following Student data.



Create a class file with the name **Student.cs** and then copy and paste the following code.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetStudents**()**

**{**

List**<**Student**>** students = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 103, Name = "Hina"**}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 102, Name = "Sambit"**}**,

new Student **{**ID = 103, Name = "Hina"**}**,

new Student **{**ID = 101, Name = "Preety" **}**,

**}**;

**return** students;

**}**

**}**

**}**

Here we created the student class with the required properties. Along the same way, we have GetStudents() method which will return all the students.

##### ****Example3:****

Here we need to fetch all the distinct names from the student’s collection. The following programs show how to achieve this using both Method and Query syntax.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MS = Student.GetStudents**()**

.Select**(**std =**>** std.Name**)**

.Distinct**()**.ToList**()**;

//Using Query Syntax

var QS = **(from** std in Student.GetStudents**()**

**select** std.Name**)**

.Distinct**()**.ToList**()**;

**foreach(**var item in MS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Example4:****

Now we need to select distinct students from the collection. As you can see in our collection three students are identical and in our result set, they should appear only once. Let us modify the program class as shown below.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MS = Student.GetStudents**()**

.Distinct**()**.ToList**()**;

//Using Query Syntax

var QS = **(from** std in Student.GetStudents**()**

**select** std**)**

.Distinct**()**.ToList**()**;

**foreach** **(**var item in QS**)**

**{**

Console.WriteLine**(**$"ID : {item.ID} , Name : {item.Name} "**)**;

**}**

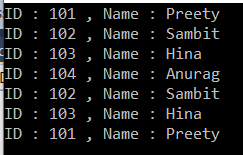
Console.ReadKey**()**;

**}**

**}**

**}**

Now execute the query and see the output.



As you can see, it will not select distinct students rather it select all the students. This is because the default comparer which is used for comparison is only checked whether two object references are equal and not the individual property values of the complex object.

##### ****How to solve the above problem?****

We can solve the above problem in four different ways. They are as follows

1. We need to use the other overloaded version of **Distinct()** method which takes **IEqualityComparer** interface as an argument. So here we will create a class that implements **IEqualityComparer** interface and then we need to pass that compare instance to the **Distinct()** method.
2. In the second approach, we need to override the **Equals()**and **GetHashCode()**methods within the **Student**class itself.
3. In the third approach, we need to project the required properties into a **new anonymous type**, which already overrides the **Equals()**and **GetHashCode()**methods
4. By Implementing **IEquatable<T>** interface.

##### ****Approach1: Implementing IEqualityComparer interface****

Create a class file with the name **StudentComparer.cs** and then implement the **IEqualityComparer** interface as shown below.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** StudentComparer : IEqualityComparer**<**Student**>**

**{**

**public** **bool** Equals**(**Student x, Student y**)**

**{**

//First check if both object reference are equal then return true

**if(object**.ReferenceEquals**(**x, y**))**

**{**

**return** **true**;

**}**

//If either one of the object refernce is null, return false

**if** **(object**.ReferenceEquals**(**x,**null)** || **object**.ReferenceEquals**(**y, **null))**

**{**

**return** **false**;

**}**

//Comparing all the properties one by one

**return** x.ID == y.ID && x.Name == y.Name;

**}**

**public** **int** GetHashCode**(**Student obj**)**

**{**

//If obj is null then return 0

**if** **(**obj == **null)**

**{**

**return** 0;

**}**

//Get the ID hash code value

**int** IDHashCode = obj.ID.GetHashCode**()**;

//Get the string HashCode Value

//Check for null refernece exception

**int** NameHashCode = obj.Name == **null** ? 0 : obj.Name.GetHashCode**()**;

**return** IDHashCode ^ NameHashCode;

**}**

**}**

**}**

As you can here we implements the **IEqualityComparer<T>** interface and then implements the **Equals()**and **GetHashCode()**methods. Now we need to create an instance of **StudentComparer** class and then we need to pass that instance to the Distinct method as shown in the below program.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Creating an instance of StudentComparer

StudentComparer studentComparer = new StudentComparer**()**;

//Using Method Syntax

var MS = Student.GetStudents**()**

.Distinct**(**studentComparer**)**.ToList**()**;

//Using Query Syntax

var QS = **(from** std in Student.GetStudents**()**

**select** std**)**

.Distinct**(**studentComparer**)**.ToList**()**;

**foreach** **(**var item in QS**)**

**{**

Console.WriteLine**(**$"ID : {item.ID} , Name : {item.Name} "**)**;

**}**

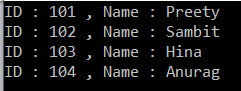
Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it will display the distinct students as shown below.



##### ****Approach2: Overriding the Equals() and GetHashCode() methods****

Now, we need to override the **Equals()**and **GetHashCode()**methods within the Student class. So, modify the Student class as shown below.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetStudents**()**

**{**

List**<**Student**>** students = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 103, Name = "Hina"**}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 102, Name = "Sambit"**}**,

new Student **{**ID = 103, Name = "Hina"**}**,

new Student **{**ID = 101, Name = "Preety" **}**,

**}**;

**return** students;

**}**

**public** **override** **bool** Equals**(object** obj**)**

**{**

//As the obj parameter type id object, so we need to

//cast it to Student Type

**return** this.ID == **((**Student**)**obj**)**.ID && this.Name == **((**Student**)**obj**)**.Name;

**}**

**public** **override** **int** GetHashCode**()**

**{**

//Get the ID hash code value

**int** IDHashCode = this.ID.GetHashCode**()**;

//Get the string HashCode Value

//Check for null refernece exception

**int** NameHashCode = this.Name == **null** ? 0 : this.Name.GetHashCode**()**;

**return** IDHashCode ^ NameHashCode;

**}**

**}**

**}**

With the above changes in Student class, modify the Main method as shown below.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MS = Student.GetStudents**()**

.Distinct**()**.ToList**()**;

//Using Query Syntax

var QS = **(from** std in Student.GetStudents**()**

**select** std**)**

.Distinct**()**.ToList**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**$"ID : {item.ID} , Name : {item.Name} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now execute the program and it will display the distinct records as expected.

##### ****Approach3:**** Using Anonymous Type

Here we need to project the properties of Student class into a **new anonymous type**, which already overrides the **Equals()**and **GetHashCode()**methods.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MS = Student.GetStudents**()**

.Select**(**std =**>** new **{** std.ID, std.Name**})**

.Distinct**()**.ToList**()**;

//Using Query Syntax

var QS = **(from** std in Student.GetStudents**()**

**select** std**)**

.Select**(**std =**>** new **{** std.ID, std.Name **})**

.Distinct**()**.ToList**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**$"ID : {item.ID} , Name : {item.Name} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

In the above example, we project the ID and Name properties to IEnumeable<’a> means to anonymous type which already overrides the Equals and GetHashCode method. Now run the application and you will see the output as expected.

##### ****Approach4: Implementing the IEquatble<T> interface in Student Class.****

Modify the Student class as shown below.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student : IEquatable**<**Student**>**

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetStudents**()**

**{**

List**<**Student**>** students = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 103, Name = "Hina"**}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 102, Name = "Sambit"**}**,

new Student **{**ID = 103, Name = "Hina"**}**,

new Student **{**ID = 101, Name = "Preety" **}**,

**}**;

**return** students;

**}**

**public** **bool** Equals**(**Student other**)**

**{**

**if** **(object**.ReferenceEquals**(**other, **null))**

**{**

**return** **false**;

**}**

**if** **(object**.ReferenceEquals**(**this, other**))**

**{**

**return** **true**;

**}**

**return** this.ID.Equals**(**other.ID**)** && this.Name.Equals**(**other.Name**)**;

**}**

**public** **override** **int** GetHashCode**()**

**{**

**int** IDHashCode = this.ID.GetHashCode**()**;

**int** NameHashCode = this.Name == **null** ? 0 : this.Name.GetHashCode**()**;

**return** IDHashCode ^ NameHashCode;

**}**

**}**

**}**

As you can see, here we have done two things. First, we implement the Equals method of the IEquatable interface and then override the GetHashCode method.

**Now change the Program class as shown below.**

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MS = Student.GetStudents**()**

.Distinct**()**.ToList**()**;

//Using Query Syntax

var QS = **(from** std in Student.GetStudents**()**

**select** std**)**

.Distinct**()**.ToList**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**$"ID : {item.ID} , Name : {item.Name} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Run the application and see the output as expected.

##### ****Difference between IEqualityComparer<T> and IEquatable<T>:****

The **IEqualityComparer<T>** is an interface for an object that performs the comparison on two objects of the type T whereas the **IEquatable<T>** is also an interface for an object of type T so that it can compare itself to another.

# LINQ Except in C#

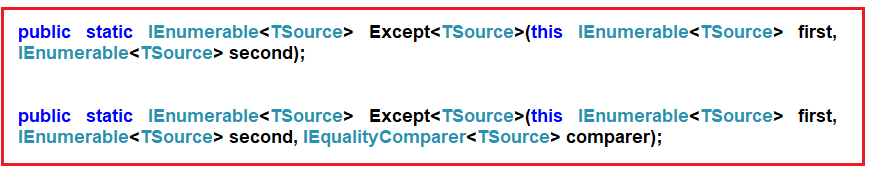
## ****LINQ Except in C# with Examples****

In this article, I am going to discuss the **LINQ Except in C#** with examples. Please read our previous article where we discussed the [**LINQ Distinct Method**](https://dotnettutorials.net/lesson/linq-distinct-method/) with examples. As part of this article, we are going to discuss the following pointers

1. **What is LINQ Except in C#?**
2. **Examples of C# LINQ Except Method using both Method and Query Syntax**
3. **How to implement IEqualityComparer?**
4. **Example using Anonymous Type**

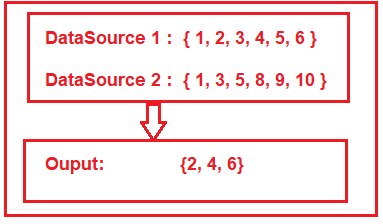
##### ****What is LINQ Except in C#?****

The **LINQ Except Method in C#** is used to return the elements which are present in the first data source but not in the second data source. There are two overloaded versions available for the LINQ Except Method as shown below.



The one and the only difference between the above two methods is the second overloaded version takes **IEqualityComparer** as an argument. That means the **Except** Method can also be used with Comparer also.

**Let us understand this with an example:**



As you can see in the above image, we have two data sources i.e. DataSource 1 and Data Source 2. The DataSource 1 contains elements such as 1, 2, 3, 4, 5, 6 and the DataSource 2 contains elements such as 1, 3, 5, 8, 9, and 10. If we want to retrieve the elements such as 2, 4, and 6 from the first data source which does not exist in the second data source then we need to apply the distinct operation.

Let us see how to do this with both Query and Method syntax.

##### ****Example1:****

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** dataSource1 = new List**<int>()** **{** 1, 2, 3, 4, 5, 6 **}**;

List**<int>** dataSource2 = new List**<int>()** **{** 1, 3, 5, 8, 9, 10 **}**;

//Method Syntax

var MS = dataSource1.Except**(**dataSource2**)**.ToList**()**;

//Query Syntax

var QS = **(from** num in dataSource1

**select** num**)**

.Except**(**dataSource2**)**.ToList**()**;

**foreach** **(**var item in QS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Run the application and you will see the output as expected.

**Note:** In query syntax, there is no such operator call Except, so here we need to use both query and method syntax to achieve the same.

##### ****Example2:****

Here we have a string array of countries and we need to return the countries from the first collection, those are not present in the second collection.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** dataSource1 = **{** "India", "USA", "UK", "Canada", "Srilanka"**}**;

string**[]** dataSource2 = **{** "India", "uk", "Canada", "France", "Japan" **}**;

//Method Syntax

var MS = dataSource1.Except**(**dataSource2**)**.ToList**()**;

//Query Syntax

var QS = **(from** country in dataSource1

**select** country**)**

.Except**(**dataSource2**)**.ToList**()**;

**foreach** **(**var item in QS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

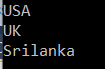
Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and see the output.



In spite of having the country UK in the second collection, it still shows in the output. This is because the default comparer that is being used to filter the data is case-insensitive.

So if you want to ignore the case-sensitive then you need to use the other overloaded version of the Except() method which takes **IEqualityComparer** as an argument.

So, modify the program as shown where we pass **StringComparer** as an argument and this **StringComparer** class already implements the **IEqualityComparer** interface.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** dataSource1 = **{** "India", "USA", "UK", "Canada", "Srilanka"**}**;

string**[]** dataSource2 = **{** "India", "uk", "Canada", "France", "Japan" **}**;

//Method Syntax

var MS = dataSource1.Except**(**dataSource2, StringComparer.OrdinalIgnoreCase**)**.ToList**()**;

//Query Syntax

var QS = **(from** country in dataSource1

**select** country**)**

.Except**(**dataSource2, StringComparer.OrdinalIgnoreCase**)**.ToList**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it will display the data as expected.

##### ****LINQ Except() Method with Complex Type:****

The **LINQ Except() Method in C#** works slightly different manner when working with complex types such as Employee, Product, Student, etc. Let us understand this with an example.

Create a class file with the name **Student.cs** and then copy and paste the following code in it.

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

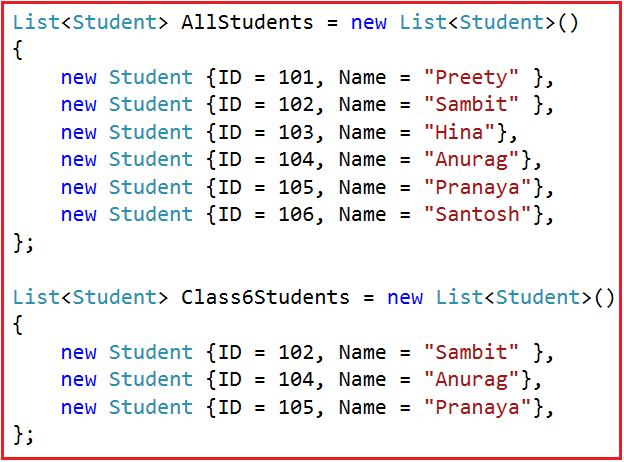
**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**}**

**}**

This is a very simple student class with just two properties. Let say, we have the following two data sources.



As you can see in the above image, we have two data sources. The first data source i.e. AllStudents hold the information of all the students while the second data source i.e. Class6Students hold the data of only the 6th class students.

##### ****Example3:****

Our requirement is only to fetch all the student names from except the 6th class students. That means we need to fetch the student names from the AllStudents data source which are not present in the second data source i.e. Class6Students.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** AllStudents = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 103, Name = "Hina"**}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 105, Name = "Pranaya"**}**,

new Student **{**ID = 106, Name = "Santosh"**}**,

**}**;

List**<**Student**>** Class6Students = new List**<**Student**>()**

**{**

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 105, Name = "Pranaya"**}**,

**}**;

//Method Syntax

var MS = AllStudents.Select**(**x =**>** x.Name**)**.Except**(**Class6Students.Select**(**y =**>** y.Name**))**.ToList**()**;

//Query Syntax

var QS = **(from** std in AllStudents

**select** std.Name**)**.Except**(**Class6Students.Select**(**y =**>** y.Name**))**.ToList**()**;

**foreach** **(**var name in MS**)**

**{**

Console.WriteLine**(**name**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Example4:****

Now we need to select all the information of all the students from the first data source which are not present in the second data source. Let us modify the program class as shown below.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** AllStudents = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 103, Name = "Hina"**}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 105, Name = "Pranaya"**}**,

new Student **{**ID = 106, Name = "Santosh"**}**,

**}**;

List**<**Student**>** Class6Students = new List**<**Student**>()**

**{**

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 105, Name = "Pranaya"**}**,

**}**;

//Method Syntax

var MS = AllStudents.Except**(**Class6Students**)**.ToList**()**;

//Query Syntax

var QS = **(from** std in AllStudents

**select** std**)**.Except**(**Class6Students**)**.ToList**()**;

**foreach** **(**var student in MS**)**

**{**

Console.WriteLine**(**$" ID : {student.ID} Name : {student.Name}"**)**;

**}**

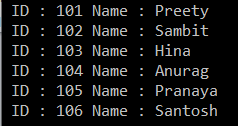
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



As you can see, it displays all the student’s data from the first data source. This is because the default comparer which is used for comparison is only checking whether two object references are equal and not the individual property values of the complex object.

In our previous article, we already discussed there many ways to solve the above problem. Here let us see how to use an anonymous type to solve the above problem.

##### ****Using Anonymous Type:****

In this approach, we need to select all the individual properties to an anonymous type. The following program does exactly the same things.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** AllStudents = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 103, Name = "Hina"**}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 105, Name = "Pranaya"**}**,

new Student **{**ID = 106, Name = "Santosh"**}**,

**}**;

List**<**Student**>** Class6Students = new List**<**Student**>()**

**{**

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 105, Name = "Pranaya"**}**,

**}**;

//Method Syntax

var MS = AllStudents.Select**(**x =**>** new **{**x.ID, x.Name **})**

.Except**(**Class6Students.Select**(**x =**>** new **{** x.ID, x.Name **}))**.ToList**()**;

//Query Syntax

var QS = **(from** std in AllStudents

**select** new **{** std.ID, std.Name**})**

.Except**(**Class6Students.Select**(**x =**>** new **{** x.ID, x.Name **}))**.ToList**()**;

**foreach** **(**var student in QS**)**

**{**

Console.WriteLine**(**$" ID : {student.ID} Name : {student.Name}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it should display the output as expected. Let us see how to achieve the same thing using Comparer.

##### ****Using Comparer:****

In this approach, we need to create a class and then we need to implement the **IEqualityComparer** interface. So, create a class file with the name **StudentComparer.cs** and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** StudentComparer : IEqualityComparer**<**Student**>**

**{**

**public** **bool** Equals**(**Student x, Student y**)**

**{**

//First check if both object reference are equal then return true

**if(object**.ReferenceEquals**(**x, y**))**

**{**

**return** **true**;

**}**

//If either one of the object refernce is null, return false

**if** **(object**.ReferenceEquals**(**x,**null)** || **object**.ReferenceEquals**(**y, **null))**

**{**

**return** **false**;

**}**

//Comparing all the properties one by one

**return** x.ID == y.ID && x.Name == y.Name;

**}**

**public** **int** GetHashCode**(**Student obj**)**

**{**

//If obj is null then return 0

**if** **(**obj == **null)**

**{**

**return** 0;

**}**

//Get the ID hash code value

**int** IDHashCode = obj.ID.GetHashCode**()**;

//Get the Name HashCode Value

**int** NameHashCode = obj.Name == **null** ? 0 : obj.Name.GetHashCode**()**;

**return** IDHashCode ^ NameHashCode;

**}**

**}**

**}**

Then create an instance of **StudentComparer** class and pass that instance to the Except method as shown in the below program.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** AllStudents = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 103, Name = "Hina"**}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 105, Name = "Pranaya"**}**,

new Student **{**ID = 106, Name = "Santosh"**}**,

**}**;

List**<**Student**>** Class6Students = new List**<**Student**>()**

**{**

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 104, Name = "Anurag"**}**,

new Student **{**ID = 105, Name = "Pranaya"**}**,

**}**;

//Create an instance of StudentComparer

StudentComparer studentComparer = new StudentComparer**()**;

//Method Syntax

var MS = AllStudents

.Except**(**Class6Students, studentComparer**)**.ToList**()**;

//Query Syntax

var QS = **(from** std in AllStudents

**select** std**)**

.Except**(**Class6Students, studentComparer**)**.ToList**()**;

**foreach** **(**var student in MS**)**

**{**

Console.WriteLine**(**$" ID : {student.ID} Name : {student.Name}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it should display the output as expected.

# LINQ Intersect Method in C#

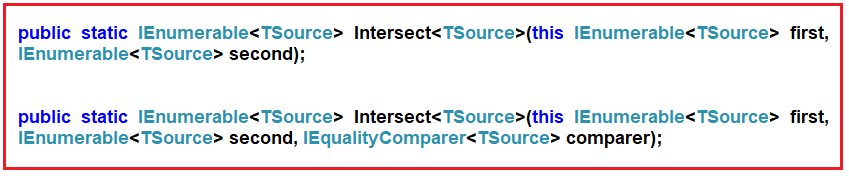
## ****LINQ Intersect Method in C# with Examples****

In this article, I am going to discuss the **LINQ Intersect Method in C#** using examples. Please read our previous article where we discussed the [**LINQ Except() Method**](https://dotnettutorials.net/lesson/linq-except-method/) with examples. As part of this article, I am going to discuss the following pointers

1. **What is LINQ Intersect Method?**
2. **Examples of LINQ Intersect Method using both Method and Query Syntax**
3. **How to implement IEqualityComparer?**

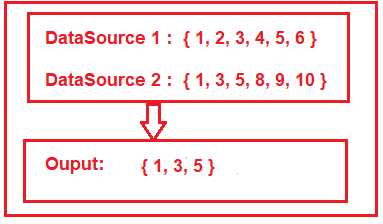
##### ****LINQ Intersect Method in C#:****

The **LINQ Intersect Method in C#** is used to return the common elements from both the collections. The elements that are present in both the data sources. There are two overloaded versions available for the LINQ Intersect Method as shown below.



The one and the only difference between the above two LINQ Intersect methods is that the second overloaded version takes **IEqualityComparer** as an argument. That means the Intersect Method is also used for Comparer.

Let us understand this with an example:



As shown in the above image, here we have two integer data sources i.e. DataSource 1 and Data Source 2. The DataSource 1 contains elements such as 1, 2, 3, 4, 5, 6 and the DataSource 2 contains elements such as 1, 3, 5, 8, 9, and 10. If we want to retrieve the elements such as 1, 3, and 5 which are exist in both the data sources then we need to use the LINQ Intersect method.

##### ****Example1:****

The following example shows the use of the LINQ Intersect() Method using both Method and Query Syntax to fetch the common elements exists in both the collections.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** dataSource1 = new List**<int>()** **{** 1, 2, 3, 4, 5, 6 **}**;

List**<int>** dataSource2 = new List**<int>()** **{** 1, 3, 5, 8, 9, 10 **}**;

//Method Syntax

var MS = dataSource1.Intersect**(**dataSource2**)**.ToList**()**;

//Query Syntax

var QS = **(from** num in dataSource1

**select** num**)**

.Intersect**(**dataSource2**)**.ToList**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Run the application and you will see the output as expected.

**Note:** In query syntax, there is no such operator call Intersect, so here we need to use the mixed syntax i.e. both the query and method syntax to achieve the same.

##### ****Example2:****

Here we have two arrays of countries and we need to return the common countries from both the collections.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** dataSource1 = **{** "India", "USA", "UK", "Canada", "Srilanka" **}**;

string**[]** dataSource2 = **{** "India", "uk", "Canada", "France", "Japan" **}**;

//Method Syntax

var MS = dataSource1.Intersect**(**dataSource2**)**.ToList**()**;

//Query Syntax

var QS = **(from** country in dataSource1

**select** country**)**

.Intersect**(**dataSource2**)**.ToList**()**;

**foreach** **(**var item in QS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and have a look at the output as shown below.

https://dotnettutorials.net/wp-content/uploads/2019/05/word-image-21.png

As you can see it displays only India and Canada. If you look at our collections, then you can see the country “UK” is present in both the collections but the Intersect method did not fetch that country. This is because the default comparer that is being used by the Intersect method is case-insensitive.

So if you want to ignore the case-sensitive then you need to use the other overloaded version of the Intersect() method which takes IEqualityComparer as an argument. So, modify the program as shown below where we pass StringComparer as an argument.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** dataSource1 = **{** "India", "USA", "UK", "Canada", "Srilanka" **}**;

string**[]** dataSource2 = **{** "India", "uk", "Canada", "France", "Japan" **}**;

//Method Syntax

var MS = dataSource1.Intersect**(**dataSource2, StringComparer.OrdinalIgnoreCase**)**.ToList**()**;

//Query Syntax

var QS = **(from** country in dataSource1

**select** country**)**

.Intersect**(**dataSource2, StringComparer.OrdinalIgnoreCase**)**.ToList**()**;

**foreach** **(**var item in QS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it will display the data as expected as shown below

 .

##### ****LINQ Intersect() Method with Complex Type:****

The LINQ Intersect() Method like other Set Operators (such as Distinct, Expect) also works in a different manner when working with complex types such as Product, Employee, Student, etc. Let us understand this with an example.

Create a class file with the name **Student.cs** and then copy and paste the following code in it.

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

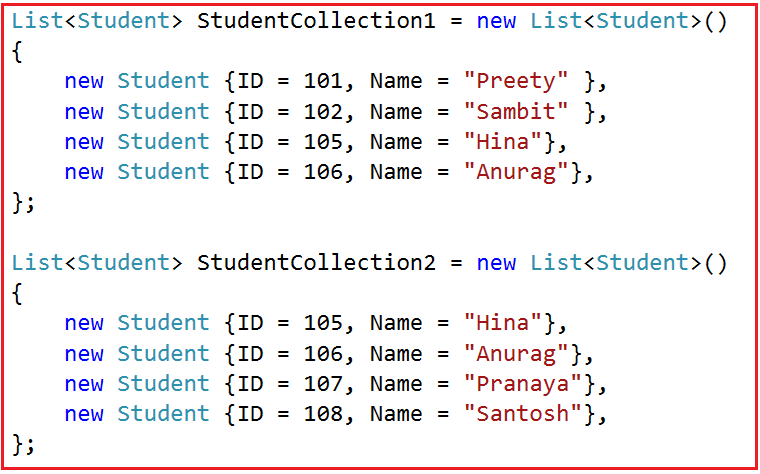
**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**}**

**}**

This is a very simple student class with just two properties. Let say, we have the following two data sources.



As you can see in the above image, we have two collections of student data. And if you notice we have two students which are appeared in both the collections.

##### ****Example3:****

Our requirement is to fetch all the student names which are present in both the collections. That is the common student’s names from both the collections.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentCollection1 = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

**}**;

List**<**Student**>** StudentCollection2 = new List**<**Student**>()**

**{**

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

new Student **{**ID = 107, Name = "Pranaya"**}**,

new Student **{**ID = 108, Name = "Santosh"**}**,

**}**;

//Method Syntax

var MS = StudentCollection1.Select**(**x =**>** x.Name**)**

. Intersect**(**StudentCollection2.Select**(**y =**>** y.Name**))**.ToList**()**;

//Query Syntax

var QS = **(from** std in StudentCollection1

**select** std.Name**)**

. Intersect**(**StudentCollection2.Select**(**y =**>** y.Name**))**.ToList**()**;

**foreach** **(**var name in MS**)**

**{**

Console.WriteLine**(**name**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

https://dotnettutorials.net/wp-content/uploads/2019/05/word-image-23.png

##### ****Example4:****

Now we need to select all the information of all the students those are present in both the collections. In order to do this, let us modify the program class as shown below.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentCollection1 = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

**}**;

List**<**Student**>** StudentCollection2 = new List**<**Student**>()**

**{**

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

new Student **{**ID = 107, Name = "Pranaya"**}**,

new Student **{**ID = 108, Name = "Santosh"**}**,

**}**;

//Method Syntax

var MS = StudentCollection1.Intersect**(**StudentCollection2**)**.ToList**()**;

//Query Syntax

var QS = **(from** std in StudentCollection1

**select** std**)**.Intersect**(**StudentCollection2**)**.ToList**()**;

**foreach** **(**var student in MS**)**

**{**

Console.WriteLine**(**$" ID : {student.ID} Name : {student.Name}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Once you run the application, then it will not display any data. This is because the default comparer which is used for comparison is only checked whether two object references are equal and not the individual property values of the complex object.

Let us see how to use an anonymous type to solve the above problem.

##### ****Using Anonymous Type for Intersect Operation in C#:****

In this approach, we need to select all the individual properties to an anonymous type. The following program does exactly the same things.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentCollection1 = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

**}**;

List**<**Student**>** StudentCollection2 = new List**<**Student**>()**

**{**

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

new Student **{**ID = 107, Name = "Pranaya"**}**,

new Student **{**ID = 108, Name = "Santosh"**}**,

**}**;

//Method Syntax

var MS = StudentCollection1.Select**(**x =**>** new **{** x.ID, x.Name **})**

.Intersect**(**StudentCollection2.Select**(**x =**>** new **{** x.ID, x.Name **}))**.ToList**()**;

//Query Syntax

var QS = **(from** std in StudentCollection1

**select** new **{**std.ID, std.Name **})**

.Intersect**(**StudentCollection2.Select**(**x =**>** new **{** x.ID, x.Name **}))**.ToList**()**;

**foreach** **(**var student in MS**)**

**{**

Console.WriteLine**(**$" ID : {student.ID} Name : {student.Name}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it should display the output as expected as shown below.

https://dotnettutorials.net/wp-content/uploads/2019/05/word-image-24.png

Let us see how to achieve the same thing using Comparer.

##### ****Using Comparer:****

In this approach, we need to create a class and then we need to implement the **IEqualityComparer** interface. So, create a class file with the name **StudentComparer.cs** and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** StudentComparer : IEqualityComparer**<**Student**>**

**{**

**public** **bool** Equals**(**Student x, Student y**)**

**{**

**return** x.ID == y.ID && x.Name == y.Name;

**}**

**public** **int** GetHashCode**(**Student obj**)**

**{**

**return** obj.ID.GetHashCode**()** ^ obj.Name.GetHashCode**()**;

**}**

**}**

**}**

Now, we need to create an instance of StudentComparer class and then we need to pass that instance to the Intersect method as shown in the below program.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentCollection1 = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

**}**;

List**<**Student**>** StudentCollection2 = new List**<**Student**>()**

**{**

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

new Student **{**ID = 107, Name = "Pranaya"**}**,

new Student **{**ID = 108, Name = "Santosh"**}**,

**}**;

StudentComparer studentComparer = new StudentComparer**()**;

//Method Syntax

var MS = StudentCollection1

.Intersect**(**StudentCollection2, studentComparer**)**.ToList**()**;

//Query Syntax

var QS = **(from** std in StudentCollection1

**select** std**)**

.Intersect**(**StudentCollection2, studentComparer**)**.ToList**()**;

**foreach** **(**var student in QS**)**

**{**

Console.WriteLine**(**$" ID : {student.ID} Name : {student.Name}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

# LINQ Union in C#

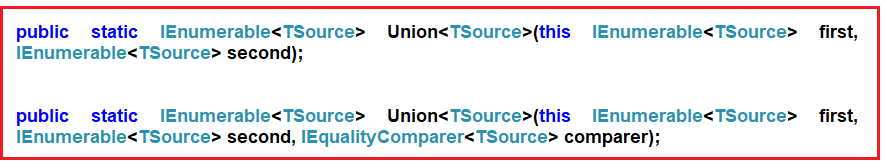
## ****LINQ Union in C# with Examples****

In this article, I am going to discuss the **LINQ Union in C#** with examples. Please read our previous article where we discussed the [**LINQ Intersect() Method**](https://dotnettutorials.net/lesson/linq-intersect-method/) with examples. As part of this article, we are going to discuss the following pointers related to the LINQ Union method in C#.

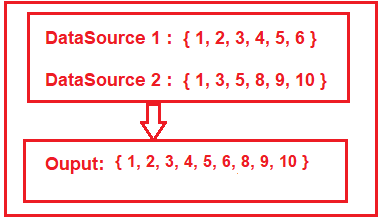
1. **What is LINQ Union in C#?**
2. **Examples of C# LINQ Union Method using both Method and Query Syntax**
3. **Using C# Linq Union Method with Complex Type.**
4. **How to implement IEqualityComparer?**
5. **Example using Anonymous Type**

##### ****What us Linq Union in C#:****

The **LINQ Union Method in C#** is used to combine the multiple data sources into one data source by removing the duplicate elements. There are two overloaded versions available for the LINQ Union Method as shown below.



Let us understand this with an example. Please have a look at the following image.



As shown in the above image, here we have two integer data sources i.e. DataSource 1 and Data Source 2. The DataSource 1 contains elements such as 1, 2, 3, 4, 5, 6 and the DataSource 2 contains elements such as 1, 3, 5, 8, 9, and 10. If we want to retrieve all the elements from both the collections by removing the duplicate element then we need to use the LINQ Union method.

##### ****Example1:****

The following example shows the use of the LINQ Union() Method using both Method and Query Syntax to fetch all the elements from both the collections by removing the duplicate elements.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** dataSource1 = new List**<int>()** **{** 1, 2, 3, 4, 5, 6 **}**;

List**<int>** dataSource2 = new List**<int>()** **{** 1, 3, 5, 8, 9, 10 **}**;

//Method Syntax

var MS = dataSource1.Union**(**dataSource2**)**.ToList**()**;

//Query Syntax

var QS = **(from** num in dataSource1

**select** num**)**

.Union**(**dataSource2**)**.ToList**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Run the application and you will see the output as expected.

**Note:** In query syntax, there is no such operator call Union, so here we need to use the mixed syntax i.e. using both query and method syntax.

##### ****Example2:****

Here we have two collections of countries and we need to return all the countries from both the collections by removing the duplicate country names.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** dataSource1 = **{** "India", "USA", "UK", "Canada", "Srilanka" **}**;

string**[]** dataSource2 = **{** "India", "uk", "Canada", "France", "Japan" **}**;

//Method Syntax

var MS = dataSource1.Union**(**dataSource2**)**.ToList**()**;

//Query Syntax

var QS = **(from** country in dataSource1

**select** country**)**

.Union**(**dataSource2**)**.ToList**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it will give us the following output.



As you can see it displays the country “**UK**” twice. This is because the default comparer that is being used by the LINQ Union method is case-insensitive.

So if you want to ignore the case-sensitive then you need to use the other overloaded version of the Union() method which takes **IEqualityComparer** as an argument. So, modify the program as shown below where we pass **StringComparer** as an argument.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** dataSource1 = **{** "India", "USA", "UK", "Canada", "Srilanka" **}**;

string**[]** dataSource2 = **{** "India", "uk", "Canada", "France", "Japan" **}**;

//Method Syntax

var MS = dataSource1.Union**(**dataSource2, StringComparer.OrdinalIgnoreCase**)**.ToList**()**;

//Query Syntax

var QS = **(from** country in dataSource1

**select** country**)**

.Union**(**dataSource2, StringComparer.OrdinalIgnoreCase**)**.ToList**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it should display the data as expected as shown below

 .

##### ****C# LINQ Union() Method with Complex Type:****

The LINQ Union() Method like other Set Operators such as Distinct, Expect, Intersect is also worked in a different manner when working with complex types such as Product, Employee, Student, etc. Let us understand this with an example.

Create a class file with the name **Student.cs** and then copy and paste the following code in it.

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

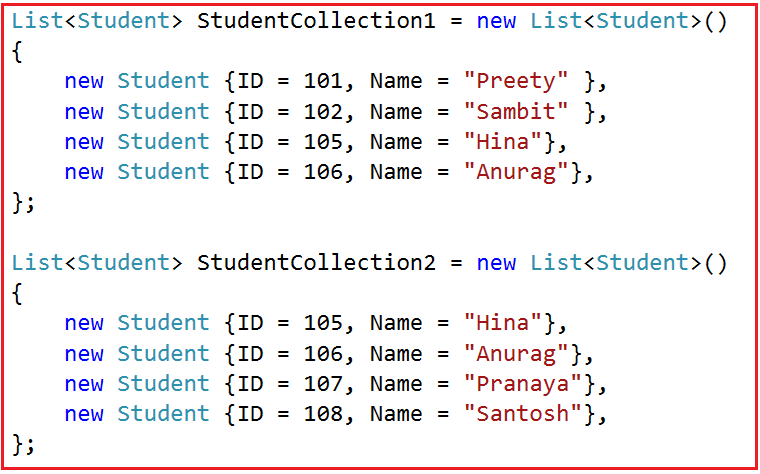
**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**}**

**}**

The above student class is created with just two properties. Let say, we have the following two data sources.



As you can see in the above image, we have two collections of student data. And if you notice we have two students which are appeared in both the collections.

##### ****Example3:****

Our requirement is to fetch all the student names from both the collections by removing the duplicate name.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentCollection1 = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

**}**;

List**<**Student**>** StudentCollection2 = new List**<**Student**>()**

**{**

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

new Student **{**ID = 107, Name = "Pranaya"**}**,

new Student **{**ID = 108, Name = "Santosh"**}**,

**}**;

//Method Syntax

var MS = StudentCollection1.Select**(**x =**>** x.Name**)**

.Union**(**StudentCollection2.Select**(**y =**>** y.Name**))**.ToList**()**;

//Query Syntax

var QS = **(from** std in StudentCollection1

**select** std.Name**)**

.Union**(**StudentCollection2.Select**(**y =**>** y.Name**))**.ToList**()**;

**foreach** **(**var name in MS**)**

**{**

Console.WriteLine**(**name**)**;

**}**

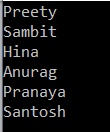
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Example4:****

Now we need to select all the information of all the students from both the collections by removing the duplicate students. In order to do this, let us modify the program class as shown below.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentCollection1 = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

**}**;

List**<**Student**>** StudentCollection2 = new List**<**Student**>()**

**{**

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

new Student **{**ID = 107, Name = "Pranaya"**}**,

new Student **{**ID = 108, Name = "Santosh"**}**,

**}**;

//Method Syntax

var MS = StudentCollection1.Union**(**StudentCollection2**)**.ToList**()**;

//Query Syntax

var QS = **(from** std in StudentCollection1

**select** std**)**.Union**(**StudentCollection2**)**.ToList**()**;

**foreach** **(**var student in MS**)**

**{**

Console.WriteLine**(**$" ID : {student.ID} Name : {student.Name}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Once you run the application, then you will see that it display all the students without removing the duplicate students. This is because the default comparer which is used for comparison is only checked whether two object references are equal and not the individual property values of the complex object. Let us see how to use an anonymous type to solve the above problem.

##### ****Using Anonymous Type for Union Operation using Linq Union Method:****

In this approach, we need to select all the individual properties to an anonymous type. The following program does exactly the same things.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentCollection1 = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

**}**;

List**<**Student**>** StudentCollection2 = new List**<**Student**>()**

**{**

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

new Student **{**ID = 107, Name = "Pranaya"**}**,

new Student **{**ID = 108, Name = "Santosh"**}**,

**}**;

//Method Syntax

var MS = StudentCollection1.Select**(**x =**>** new **{** x.ID, x.Name **})**

.Union**(**StudentCollection2.Select**(**x =**>** new **{** x.ID, x.Name **}))**.ToList**()**;

//Query Syntax

var QS = **(from** std in StudentCollection1

**select** new **{** std.ID, std.Name **})**

.Union**(**StudentCollection2.Select**(**x =**>** new **{** x.ID, x.Name **}))**.ToList**()**;

**foreach** **(**var student in MS**)**

**{**

Console.WriteLine**(**$" ID : {student.ID} Name : {student.Name}"**)**;

**}**

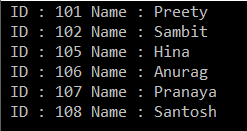
Console.ReadKey**()**;

**}**

**}**

**}**

Now run the application and it should display the output as expected as shown below.



Let us see how to achieve the same thing using Comparer.

##### ****Using IEqualityComparer :****

In this approach, we need to create a class and then we need to implement the **IEqualityComparer** interface. So, create a class file with the name **StudentComparer.cs** and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** StudentComparer : IEqualityComparer**<**Student**>**

**{**

**public** **bool** Equals**(**Student x, Student y**)**

**{**

**return** x.ID == y.ID && x.Name == y.Name;

**}**

**public** **int** GetHashCode**(**Student obj**)**

**{**

**return** obj.ID.GetHashCode**()** ^ obj.Name.GetHashCode**()**;

**}**

**}**

**}**

Now, we need to create an instance of **StudentComparer** class and then we need to pass that instance to the LINQ Union method as shown in the below program.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentCollection1 = new List**<**Student**>()**

**{**

new Student **{**ID = 101, Name = "Preety" **}**,

new Student **{**ID = 102, Name = "Sambit" **}**,

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

**}**;

List**<**Student**>** StudentCollection2 = new List**<**Student**>()**

**{**

new Student **{**ID = 105, Name = "Hina"**}**,

new Student **{**ID = 106, Name = "Anurag"**}**,

new Student **{**ID = 107, Name = "Pranaya"**}**,

new Student **{**ID = 108, Name = "Santosh"**}**,

**}**;

StudentComparer studentComparer = new StudentComparer**()**;

//Method Syntax

var MS = StudentCollection1

.Union**(**StudentCollection2, studentComparer**)**.ToList**()**;

//Query Syntax

var QS = **(from** std in StudentCollection1

**select** std**)**

.Union**(**StudentCollection2, studentComparer**)**.ToList**()**;

**foreach** **(**var student in MS**)**

**{**

Console.WriteLine**(**$" ID : {student.ID} Name : {student.Name}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

# Linq Concat Method in C#

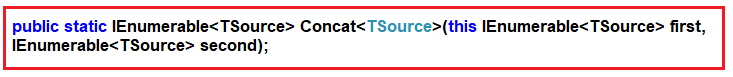
## ****Linq Concat Method in C# with Examples****

In this article, I am going to discuss the **Linq Concat Method in C#** with some examples. Please read our previous article before proceeding to this article where we discussed the [**Linq Union Method in C#**](https://dotnettutorials.net/lesson/linq-union-method/) with some examples. As part of this article, we are going to discuss the following pointers.

1. **What is the Concat Method in Linq?**
2. **Why do we need to use the Concat Method**?
3. **Examples using both Query and Method Syntax.**
4. **What are the differences between Concat and union operators in Linq?**

#### ****Linq Concat Method in C#:****

The **Linq Concat Method in C#** is used to concatenate two sequences into one sequence. There is only one version available for this method whose signature is given below.



##### ****Example1:****

In the following example, we have created two integer sequences and then concatenate two sequences into one sequence using the Concat operator.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** sequence1 = new List**<int>** **{** 1, 2, 3, 4 **}**;

List**<int>** sequence2 = new List**<int>** **{** 2, 4, 6, 8 **}**;

var result = sequence1.Concat**(**sequence2**)**;

**foreach** **(**var item in result**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

**Output:**

Concat Operator in Linq Output

If you notice in the above output then you will see that the duplicate elements are not removed. Now let us concatenate the above two sequences using the Union operator and observe what happened.

##### ****Concatenate using Union Operator:****

In the below example, we concatenate the two integer sequences into one sequence using the **Linq Union Operator**.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** sequence1 = new List**<int>** **{** 1, 2, 3, 4 **}**;

List**<int>** sequence2 = new List**<int>** **{** 2, 4, 6, 8 **}**;

var result = sequence1.Union**(**sequence2**)**;

**foreach** **(**var item in result**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

**Output:**

Union Operator in Linq Output

If you observe in the above output, then you will see that the duplicate elements are removed from the result set.

##### ****What is the difference between Concat and Union operators in Linq?****

The Concat operator is used to concatenate two sequences into one sequence without removing the duplicate elements. That means it simply returns the elements from the first sequence followed by the elements from the second sequence.

On the other hand, the Linq Union operator is also used to concatenate two sequences into one sequence by removing the duplicate elements.

**Note:** While working with the Concat operator if any of the sequences is null then it will throw an exception.

##### ****Example2:****

In the following example, the second sequence is null and while performing the concatenate operation using the concat operator it will throw an exception.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** sequence1 = new List**<int>** **{** 1, 2, 3, 4 **}**;

List**<int>** sequence2 = **null**;

var result = sequence1.Concat**(**sequence2**)**;

**foreach** **(**var item in result**)**

**{**

Console.WriteLine**(**item**)**;

**}**

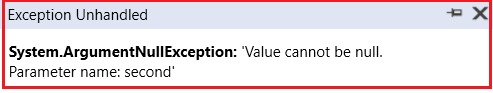
Console.ReadLine**()**;

**}**

**}**

**}**

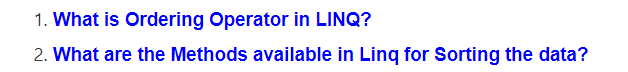
Now run the application and you will get the following exception.



# Ordering Operators in LINQ

## ****Ordering Operators in LINQ****

In this article, I am going to discuss the **Ordering Operators in LINQ** with examples. Please read our previous article where we discussed the [**Union Method in C#**](https://dotnettutorials.net/lesson/linq-union-method/) with examples. At the end of this article, you will learn the following two things.



##### ****What Are Ordering Operators?****

In simple terms, we can say that Ordering is nothing but a process to manage the data in a particular order. It is not changing the data or output rather this operation arranges the data in a particular order i.e. either ascending order or descending order.

In this case, the count is going to be the same but the order of the element is going to change.

We sort the data in two ways i.e. ascending or descending order. The order may be integer-based or any other data type based. For example

1. Name of Cities of a particular state in alphabetical order.
2. Students order by Roll Number in a class.

It is also possible to order based on multiple columns like Employee First and Last Name in ascending order while the Salary is on descending order.

##### ****What are the Methods available in Linq for Sorting the data?****

There are five methods provided by LINQ to sort the data. They are as follows

1. [**OrderBy**](https://dotnettutorials.net/lesson/linq-orderby-method/)
2. [**OrderByDescending**](https://dotnettutorials.net/lesson/linq-orderbydescending-method/)
3. [**ThenBy**](https://dotnettutorials.net/lesson/linq-thenby-and-thenbydescending/)
4. [**ThenByDescending**](https://dotnettutorials.net/lesson/linq-thenby-and-thenbydescending/)
5. [**Reverse**](https://dotnettutorials.net/lesson/linq-reverse-method/)

# Linq OrderBy Method in C#

## ****Linq OrderBy Method in C# with Examples****

In this article, I am going to discuss the **LINQ OrderBy Method in C#**with examples. Please read our previous article where we discussed the basics of [**Ordering Operators**](https://dotnettutorials.net/lesson/ordering-operators-in-linq/) in LINQ. As part of this article, we are going to discuss the following pointers related to the  LINQ OrderBy method.

1. **What is Linq OrderBy Method?**
2. **Example of Linq OrderBy Method using both Method and Query Syntax.**
3. **How to use Linq OrderBy Method with Complex Type in C#?**
4. **How to use the OrderBy method along with the Filtering method?**

##### ****What is Linq OrderBy Method?****

The Linq OrderBy method in C# is used to sort the data in ascending order. The most important point that you need to keep in mind is this method is not going to change the data rather it is just changing the order of the data.

You can use the OrderBy method on any data type i.e. you can use character, string, decimal, integer, etc. Let us understand the use of the LINQ OrderBy method in C# using both query syntax and method syntax.

##### ****Example1: Working with integer data****

In the below example we have a collection of integer data. And we sort the data in ascending order using the OrderBy method.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** intList = new List**<int>()** **{** 10, 45, 35, 29, 100, 69, 58, 50 **}**;

Console.WriteLine**(**"Before Sorting : "**)**;

**foreach** **(**var item in intList**)**

**{**

Console.Write**(**item + " "**)**;

**}**

//Sorting the data in Ascending Order

//Using Method Syntax

var MS = intList.OrderBy**(**num =**>** num**)**;

//Using Query Syntax

var QS = **(from** num in intList

**orderby** num

**select** num**)**.ToList**()**;

Console.WriteLine**()**;

Console.WriteLine**(**"After Sorting : "**)**;

**foreach** **(**var item in QS**)**

**{**

Console.Write**(**item + " "**)**;

**}**

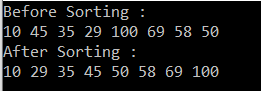
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Exampe2: Working with string data.****

In the below example we have a collection of string names. We then sort the data in ascending order using both method and query syntax.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** stringList = new List**<**string**>()** **{** "Preety", "Tiwary", "Agrawal", "Priyanka", "Dewangan",

"Hina","Kumar","Manoj", "Rout", "James"**}**;

//Using Method Syntax

var MS = stringList.OrderBy**(**name =**>** name**)**;

//Using Query Syntax

var QS = **(from** name in stringList

**orderby** name **ascending**

**select** name**)**.ToList**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**item + " "**)**;

**}**

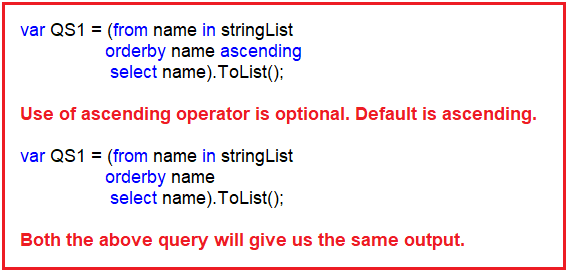
Console.ReadKey**()**;

**}**

**}**

**}**

**Note:** In query syntax, while we are sorting the data in ascending order then the use of ascending operator is optional. That means if we are not specifying anything then by default it is ascending. So the following two statements are the same.



##### ****Using LINQ OrderBy Method with Complex type:****

We are going to work with the following Student class. So, create a class file with the name Student.cs and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string FirstName **{** **get**; **set**; **}**

**public** string LastName **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetAllStudents**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**{**ID= 101,FirstName = "Preety",LastName = "Tiwary",Branch = "CSE"**}**,

new Student**{**ID= 102,FirstName = "Preety",LastName = "Agrawal",Branch = "ETC"**}**,

new Student**{**ID= 103,FirstName = "Priyanka",LastName = "Dewangan",Branch = "ETC"**}**,

new Student**{**ID= 104,FirstName = "Hina",LastName = "Sharma",Branch = "ETC"**}**,

new Student**{**ID= 105,FirstName = "Anugrag",LastName = "Mohanty",Branch = "CSE"**}**,

new Student**{**ID= 106,FirstName = "Anurag",LastName = "Sharma",Branch = "CSE"**}**,

new Student**{**ID= 107,FirstName = "Pranaya",LastName = "Kumar",Branch = "CSE"**}**,

new Student**{**ID= 108,FirstName = "Manoj",LastName = "Kumar",Branch = "ETC"**}**,

new Student**{**ID= 109,FirstName = "Pranaya",LastName = "Rout",Branch = "ETC"**}**,

new Student**{**ID= 110,FirstName = "Saurav",LastName = "Rout",Branch = "CSE"**}**

**}**;

**return** listStudents;

**}**

**}**

**}**

As you can see, we created the Student class with four properties such as**ID, FirstName, LastName, and Brach**. We then created one method (i.e. **GetAllStudents**) within the same class which is going to return a list of all students.

##### ****Example3: Sorting the Data in Ascending Order****

Here, we want to sort the data based on the Branch of the Student in ascending order.

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Method Syntax

var MS = Student.GetAllStudents**()**.OrderBy**(**x =**>** x.Branch**)**.ToList**()**;

//Query Syntax

var QS = **(from** std in Student.GetAllStudents**()**

**orderby** std.Branch

**select** std**)**;

**foreach** **(**var student in MS**)**

**{**

Console.WriteLine**(**" Branch: " + student.Branch + ", Name :" + student.FirstName + " " + student.LastName **)**;

**}**

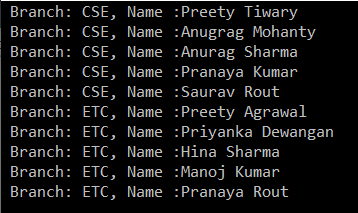
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Example4: Sorting with Filtering.****

Now we need to fetch only the CSE branch students and then we need to sort the data based on the FirstName in ascending order.

Note: The most important point that you need to remember is, you need to use the Where method before the OrderBy method. The following example shows the above using both query and method syntax.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Method Syntax

var MS = Student.GetAllStudents**()**

.Where**(**std =**>** std.Branch.ToUpper**()** == "CSE"**)**

.OrderBy**(**x =**>** x.FirstName**)**.ToList**()**;

//Query Syntax

var QS = **(from** std in Student.GetAllStudents**()**

**where** std.Branch.ToUpper**()** == "CSE"

**orderby** std.FirstName

**select** std**)**;

**foreach** **(**var student in QS**)**

**{**

Console.WriteLine**(**" Branch: " + student.Branch + ", Name :" + student.FirstName + " " + student.LastName **)**;

**}**

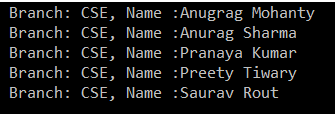
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



# Linq OrderByDescending Method in C#

## ****LINQ OrderByDescending Method in C# with Examples****

In this article, I am going to discuss the **LINQ OrderByDescending Method in C#** with examples. Please read our previous article where we discussed the [**OrderBy Method in C#**](https://dotnettutorials.net/lesson/linq-orderby-method/) with some examples. As part of this article, we are going to discuss the following pointers related to the  **LINQ OrderByDescending** method.

1. **What is Linq OrderByDescending Method in C#?**
2. **Example of Linq OrderByDescending Method using both Method and Query Syntax.**
3. **How to use Linq OrderByDescending Method with Complex Type in C#?**
4. **How to use the OrderByDescending method along with the Filtering method?**

#### ****What is Linq OrderByDescending Method in C#?****

The **LINQ OrderByDescending method in C#** is used to sort the data in descending order. The point that you need to remember is, the OrderByDescending method is not going to change the data, it is just changing the order of the data.

Like the **OrderBy method**, you can also use the **OrderByDescending** method on any data type such as string, character, float, integer, etc. Let us understand how to use the **OrderByDescending method in C#** using both query and method syntax.

##### ****Working with integer data****

In the following example, we have an integer collection. And we need to sort the data in descending order. Let us see how to do this using both query and method syntax.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** intList = new List**<int>()** **{** 10, 45, 35, 29, 100, 69, 58, 50 **}**;

Console.WriteLine**(**"Before Sorting the Data: "**)**;

**foreach** **(**var item in intList**)**

**{**

Console.Write**(**item + " "**)**;

**}**

//Sorting the data in Descending Order

//Using Method Syntax

var MS = intList.OrderByDescending**(**num =**>** num**)**;

//Using Query Syntax

var QS = **(from** num in intList

**orderby** num **descending**

**select** num**)**.ToList**()**;

Console.WriteLine**()**;

Console.WriteLine**(**"After Sorting the Data in Descending Order: "**)**;

**foreach** **(**var item in QS**)**

**{**

Console.Write**(**item + " "**)**;

**}**

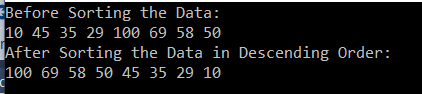
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Working with string data.****

In the following example, we have a collection of string data i.e. a collection of names. We want to sort the data in descending order using both method and query syntax. Let us see how we can do this.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** stringList = new List**<**string**>()** **{** "Preety", "Tiwary", "Agrawal", "Priyanka", "Dewangan",

"Hina","Kumar","Manoj", "Rout", "James"**}**;

//Using Method Syntax

var MS = stringList.OrderByDescending**(**name =**>** name**)**;

//Using Query Syntax

var QS = **(from** name in stringList

**orderby** name **descending**

**select** name**)**.ToList**()**;

**foreach** **(**var item in MS**)**

**{**

Console.WriteLine**(**item + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

##### ****LINQ OrderByDescending Method with Complex type in C#:****

In order to understand how to work with complex type, we are going to work with the following Student class. So, create a class file with the name Student.cs and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string FirstName **{** **get**; **set**; **}**

**public** string LastName **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetAllStudents**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**{**ID= 101,FirstName = "Preety",LastName = "Tiwary",Branch = "CSE"**}**,

new Student**{**ID= 102,FirstName = "Preety",LastName = "Agrawal",Branch = "ETC"**}**,

new Student**{**ID= 103,FirstName = "Priyanka",LastName = "Dewangan",Branch = "ETC"**}**,

new Student**{**ID= 104,FirstName = "Hina",LastName = "Sharma",Branch = "ETC"**}**,

new Student**{**ID= 105,FirstName = "Anugrag",LastName = "Mohanty",Branch = "CSE"**}**,

new Student**{**ID= 106,FirstName = "Anurag",LastName = "Sharma",Branch = "CSE"**}**,

new Student**{**ID= 107,FirstName = "Pranaya",LastName = "Kumar",Branch = "CSE"**}**,

new Student**{**ID= 108,FirstName = "Manoj",LastName = "Kumar",Branch = "ETC"**}**,

new Student**{**ID= 109,FirstName = "Pranaya",LastName = "Rout",Branch = "ETC"**}**,

new Student**{**ID= 110,FirstName = "Saurav",LastName = "Rout",Branch = "CSE"**}**

**}**;

**return** listStudents;

**}**

**}**

**}**

As you can see, we created the above Student class with four simple properties (ID, FirstName, LastName, and Brach). We then created one method (i.e. GetAllStudents) which will return the list of all students.

##### ****Sorting the Data in Descending Order****

Here, we want to sort the data based on the Branch in descending order.

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Method Syntax

var MS = Student.GetAllStudents**()**.OrderByDescending**(**x =**>** x.Branch**)**.ToList**()**;

//Query Syntax

var QS = **(from** std in Student.GetAllStudents**()**

**orderby** std.Branch **descending**

**select** std**)**;

**foreach** **(**var student in MS**)**

**{**

Console.WriteLine**(**" Branch: " + student.Branch + ", Name :" + student.FirstName + " " + student.LastName**)**;

**}**

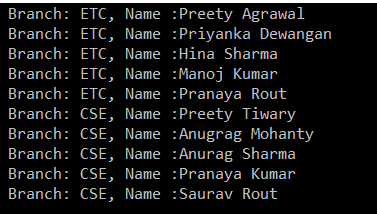
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Linq OrderByDescending with Filtering Operator.****

Now we need to fetch only the ETC branch students and then we need to sort the students based on their FirstName in descending order.

Note: The most important point that you need to keep in mind is, you need to use the Where extension method before the OrderByDescending method.

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Method Syntax

var MS = Student.GetAllStudents**()**

.Where**(**std =**>** std.Branch.ToUpper**()** == "ETC"**)**

.OrderByDescending**(**x =**>** x.FirstName**)**.ToList**()**;

//Query Syntax

var QS = **(from** std in Student.GetAllStudents**()**

**where** std.Branch.ToUpper**()** == "ETC"

**orderby** std.FirstName **descending**

**select** std**)**;

**foreach** **(**var student in QS**)**

**{**

Console.WriteLine**(**" Branch: " + student.Branch + ", Name :" + student.FirstName + " " + student.LastName**)**;

**}**

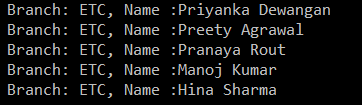
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



# Linq ThenBy and ThenByDescending Method in C#

## ****LINQ ThenBy and ThenByDescending Method in C# with Examples****

In this article, I am going to discuss the **LINQ ThenBy and ThenByDescending Method in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**OrderByDescending method in C#**](https://dotnettutorials.net/lesson/linq-orderbydescending-method/) with some examples. Here, in this article, we are going to discuss the following pointers related to the **LINQ ThenBy and ThenByDescending**Method in C#.

1. **Why we need the ThenBy and ThenByDescending Method in C#?**
2. **What are the ThenBy and ThenByDescending Method in C#?**
3. **Example of Linq ThenBy and ThenByDescending Method using both Method and Query Syntax.**
4. **How to use the ThenBy and ThenByDescending along with the Filtering method?**

##### ****Why we need the LINQ ThenBy and ThenByDescending Method in C#?****

The **LINQ OrderBy or OrderByDescending** method works fine when you want to sort the data based on a single value or a single expression. But if you want to sort the data based on multiple values or multiple expressions then you need to use the LINQ ThenBy and ThenByDescending Method along with OrderBy or OrderByDescending Method.

##### ****What are the Linq ThenBy and ThenByDescending Method in C#?****

The **Linq ThenBy Method** in C# is used to sort the data in ascending order from the second level onwards. On the other hand, the  Linq **ThenByDescending Method in C#** is used to sort the data in descending order also from the second level onwards.

These two methods are used along with **OrderBy** or **OrderByDescending** method. You can use the ThenBy or ThenByDescending method more than once in the same LINQ query.

The **OrderBy or OrderByDescending** method is generally used for primary sorting. **ThenBy** or **ThenByDescending** are used for secondary sorting and so on. For example, first, sort the student by First Name and then sort the student by the Last Name.

##### ****Example:****

We are going to use the following Student class in order to understand the use of the ThenBy and ThenByDescending method in C#.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string FirstName **{** **get**; **set**; **}**

**public** string LastName **{** **get**; **set**; **}**

**public** string Branch **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetAllStudents**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**{**ID= 101,FirstName = "Preety",LastName = "Tiwary",Branch = "CSE"**}**,

new Student**{**ID= 102,FirstName = "Preety",LastName = "Agrawal",Branch = "ETC"**}**,

new Student**{**ID= 103,FirstName = "Priyanka",LastName = "Dewangan",Branch = "ETC"**}**,

new Student**{**ID= 104,FirstName = "Hina",LastName = "Sharma",Branch = "ETC"**}**,

new Student**{**ID= 105,FirstName = "Anugrag",LastName = "Mohanty",Branch = "CSE"**}**,

new Student**{**ID= 106,FirstName = "Anurag",LastName = "Sharma",Branch = "CSE"**}**,

new Student**{**ID= 107,FirstName = "Pranaya",LastName = "Kumar",Branch = "CSE"**}**,

new Student**{**ID= 108,FirstName = "Manoj",LastName = "Kumar",Branch = "ETC"**}**,

new Student**{**ID= 109,FirstName = "Pranaya",LastName = "Rout",Branch = "ETC"**}**,

new Student**{**ID= 110,FirstName = "Saurav",LastName = "Rout",Branch = "CSE"**}**

**}**;

**return** listStudents;

**}**

**}**

**}**

##### ****Example: Using Method Syntax:****

First, we need to sort the student by First Name in ascending order and then we need to sort the student by the Last Name in ascending order.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Method Syntax

var MS = Student.GetAllStudents**()**

.OrderBy**(**x =**>** x.FirstName**)**

.ThenBy**(**y =**>** y.LastName**)**

.ToList**()**;

**foreach** **(**var student in MS**)**

**{**

Console.WriteLine**(** "First Name :" + student.FirstName + ", Last Name : " + student.LastName**)**;

**}**

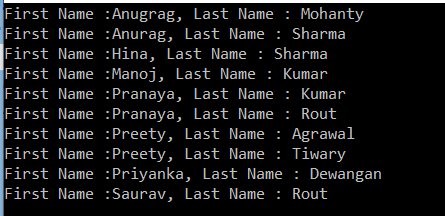
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Example: Using Query Syntax:****

We do not have any method called ThenBy and ThenByDescending in query syntax. So here we need to specify multiple values or expressions in the order by clause separated by a comma as shown in the below example.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Query Syntax

var QS = **(from** std in Student.GetAllStudents**()**

**orderby** std.FirstName, std.LastName

**select** std**)**;

**foreach** **(**var student in QS**)**

**{**

Console.WriteLine**(** "First Name :" + student.FirstName + ", Last Name : " + student.LastName**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

It will give us the same output as Method Syntax.

##### ****Example:****

First sort the data in ascending order based on Branch. Then sort the data in descending order based on First Name. Finally, sort the data on the ascending order based on the Last Name values.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MS = Student.GetAllStudents**()**

.OrderBy**(**x =**>** x.Branch**)**

.ThenByDescending**(**y =**>** y.FirstName**)**

.ThenBy**(**z =**>** z.LastName**)**

.ToList**()**;

//Query Syntax

var QS = **(from** std in Student.GetAllStudents**()**

**orderby** std.Branch **ascending**,

std.FirstName **descending**,

std.LastName

**select** std**)**.ToList**()**;

**foreach** **(**var student in QS**)**

**{**

Console.WriteLine**(** "Barnch " + student.Branch+ ", First Name :" + student.FirstName + ", Last Name : " + student.LastName**)**;

**}**

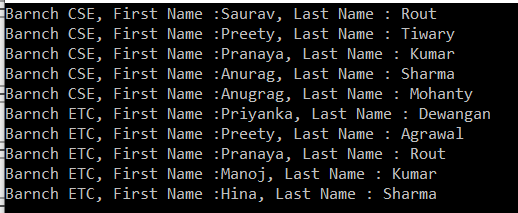
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Example: Using ThenBy and ThenByDescending along with the Where Method:****

Here first we need to fetch only the CSE branch students and then we need to sort the data as follows. First sort the data in ascending order based on First Name. Then sort the data in descending order based on the Last Name.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MS = Student.GetAllStudents**()**

.Where**(**std =**>** std.Branch == "CSE"**)**

.OrderBy**(**x =**>** x.FirstName**)**

.ThenByDescending**(**y =**>** y.LastName**)**

.ToList**()**;

//Query Syntax

var QS = **(from** std in Student.GetAllStudents**()**

**where** std.Branch == "CSE"

**orderby** std.FirstName,

std.LastName **descending**

**select** std**)**.ToList**()**;

**foreach** **(**var student in QS**)**

**{**

Console.WriteLine**(** "Barnch " + student.Branch+ ", First Name :" + student.FirstName + ", Last Name : " + student.LastName**)**;

**}**

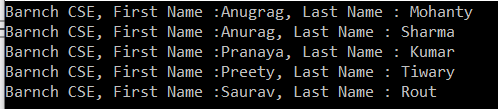
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



# LINQ Reverse Method in C#

## ****LINQ Reverse Method in C# with Examples****

In this article, I am going to discuss the **LINQ Reverse Method in C#** with examples. Please read our previous article before proceeding to this article where we discussed [**LINQ ThenBy and ThenByDescending Method in C#**](https://dotnettutorials.net/lesson/linq-thenby-and-thenbydescending/) with some examples. As part of this article, we are going to discuss the following pointers in details.

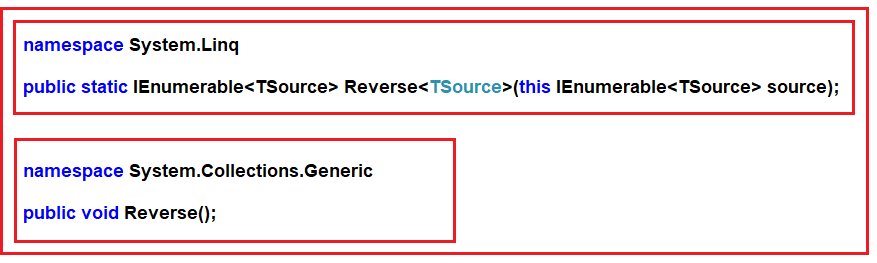
1. **What is Reverse Method in C#?**
2. **Different Types of Example using both Method and Query Syntax.**
3. **We will discuss the Reverse Method available in System.LInq and System.Collections.Generic namespaces.**
4. **Finally, we will discuss how to apply the LINQ Reverse method (the Reverse Method which is available in System.LInq namespace) on a collection of List<T> type.**

##### ****What is Reverse Method in C#?****

The LINQ Reverse method is used to reverse the data stored in a data source. That means this method will not change the data rather it simple reverse the data stored in the data source. So, as a result, we will get the output in reverse order.

##### ****C# Reverse Method Signature:****

The Reverse Method is implemented in two different namespaces such as **System.LInq** and **System.Collections.Generic** namespaces. Let us have a look at the signature or definition of the Reverse Method which is shown in the below image.



As you can see the Reverse method which belongs to **System.Linq** namespace implemented as an extension method on **IEnumerable<TSource>** interface and more importantly this method also returns an **IEnumerable<TSource>** type.

On the other hand, the Reverse method which belongs to the **System.Collections.Generic** namespace is not returning any value as the return type is void.

With this keep in mind, let us see some examples for a better understanding of the Reverse method.

##### ****Example1: System.Linq namespace Reverse method****

In order to understand this, we are going to work with an integer array as shown in the below example.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intArray = new **int[]** **{** 10, 30, 50, 40,60,20,70,100 **}**;

Console.WriteLine**(**"Before Reverse the Data"**)**;

**foreach** **(**var number in intArray**)**

**{**

Console.Write**(**number + " "**)**;

**}**

Console.WriteLine**()**;

IEnumerable**<int>** ArrayReversedData = intArray.Reverse**()**;

Console.WriteLine**(**"After Reverse the Data"**)**;

**foreach** **(**var number in ArrayReversedData**)**

**{**

Console.Write**(**number + " "**)**;

**}**

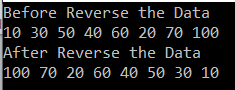
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Note:** In the above example, if you go to the definition of the Reverse method then you can see that this Reverse method belongs to the System.Linq namespace and hence we can able to store the data in a variable of type IEnumerable<int> as the source contains integer data.

##### ****Using Query Syntax:****

The most important point is that we do not have any such operator called Reverse available in Linq to write query syntax. So here we need to use the mixed syntax. So let us see how to rewrite the same example using mixed syntax.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intArray = new **int[]** **{** 10, 30, 50, 40,60,20,70,100 **}**;

Console.WriteLine**(**"Before Reverse the Data"**)**;

**foreach** **(**var number in intArray**)**

**{**

Console.Write**(**number + " "**)**;

**}**

Console.WriteLine**()**;

IEnumerable**<int>** ArrayReversedData = **(from** num in intArray

**select** num**)**.Reverse**()**;

Console.WriteLine**(**"After Reverse the Data"**)**;

**foreach** **(**var number in ArrayReversedData**)**

**{**

Console.Write**(**number + " "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

##### ****Example2:**** ****System.Collections.Generic namespace Reverse method****

In order to understand the above Reverse method, we need to create a collection of List<string> as shown in the below examples.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** stringList = new List**<**string**>()** **{** "Preety", "Tiwary", "Agrawal", "Priyanka", "Dewangan"**}**;

Console.WriteLine**(**"Before Reverse the Data"**)**;

**foreach** **(**var name in stringList**)**

**{**

Console.Write**(**name + " "**)**;

**}**

Console.WriteLine**()**;

//You cannot store the data like below as this method belongs to

//System.Collections.Generic namespace whose return type is void

//IEnumerable<int> ArrayReversedData = stringList.Reverse();

stringList.Reverse**()**;

Console.WriteLine**(**"After Reverse the Data"**)**;

**foreach** **(**var name in stringList**)**

**{**

Console.Write**(**name + " "**)**;

**}**

Console.ReadKey**()**;

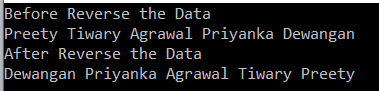
**}**

**}**

**}**

If you go to the definition of the above Reverse method, then you can see this method belongs to the System.Collections.Generic namespace and the return type of this Reverse method is Void.

Now run the application and you will get the output as expected as shown below.



##### ****How to apply the Linq Reverse method on a collection of List<T> type?****

If you want to apply the Reverse method which belongs to **System.Linq** namespace on a collection of type **List<T>**, then first you need to convert to the **List<T>** collection to as **IEnumerable** or **IQueryable** collection by using the **AsEnumerable()** or **AsQueryable()** method on the data source.

If you use the **AsEnumerable()** method then it will convert the collection to **IEnumerable** whereas if you use **AsQueryable()** method then it will convert the collection to **IQueryable**.

The following program shows how to apply the Linq Reverse method on a collection of type List<T>.

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** stringList = new List**<**string**>()** **{** "Preety", "Tiwary", "Agrawal", "Priyanka", "Dewangan" **}**;

Console.WriteLine**(**"Before Reverse the Data"**)**;

**foreach** **(**var name in stringList**)**

**{**

Console.Write**(**name + " "**)**;

**}**

Console.WriteLine**()**;

IEnumerable**<**string**>** ReverseData1 = stringList.AsEnumerable**()**.Reverse**()**;

IQueryable**<**string**>** ReverseData2 = stringList.AsQueryable**()**.Reverse**()**;

Console.WriteLine**(**"After Reverse the Data"**)**;

**foreach** **(**var name in ReverseData1**)**

**{**

Console.Write**(**name + " "**)**;

**}**

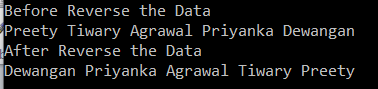
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

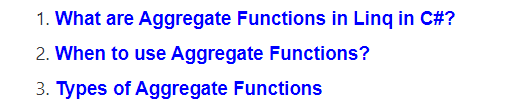


**Note:** The LINQ Reverse Method in C# will return the data as **IEnumerable<TSource>** or **IQuereable<TSource>** based on how we use the LINQ method.

# LINQ Aggregate Functions in C#

## ****LINQ Aggregate Functions in C#****

In this article, I am going to discuss the **Linq Aggregate Functions in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Reverse Method in C#**](https://dotnettutorials.net/lesson/linq-reverse-method/) with some examples. As part of this article, we are going to discuss the following concepts.



##### ****What are Linq Aggregate Functions in C#?****

The Linq aggregate functions are used to group together the values of multiple rows as the input and then return the output as a single value. So, simple word, we can say that the aggregate function in C# is always going to return a single value.

##### ****When to use the Aggregate Functions in C#?****

Whenever you want to perform some mathematical operations such as Sum, Count, Max, Min, Average, and Aggregate on the numeric property of a collection then you need to use the Linq Aggregate Functions.

##### ****What are the Aggregate Methods Provided by Linq?****

The following are the aggregate methods provided by Linq to perform mathematical operations on a collection.

1. [**Sum():**](https://dotnettutorials.net/lesson/linq-sum-method/) This method is used to calculate the total(sum) value of the collection.
2. [**Max():**](https://dotnettutorials.net/lesson/linq-max-method/) This method is used to find the largest value in the collection
3. [**Min():**](https://dotnettutorials.net/lesson/linq-min-method/) This method is used to find the smallest value in the collection
4. [**Average():**](https://dotnettutorials.net/lesson/linq-average-method/)This method is used to calculate the average value of the numeric type of the collection.
5. [**Count():**](https://dotnettutorials.net/lesson/linq-count-method/)This method is used to count the number of elements present in the collection.
6. [**Aggregate():**](https://dotnettutorials.net/lesson/linq-aggregate-method/)This method is used to Performs a custom aggregation operation on the values of a collection.

# Linq Sum in C#

## ****Linq Sum in C# with Examples****

In this article, I am going to discuss the **Linq Sum  in C#** with examples. Please read our previous article before proceeding to this article where we discussed the basics of [**Linq Aggregate Functions in C#**](https://dotnettutorials.net/lesson/linq-aggregate-functions/) with some examples. As part of this article, we are going to discuss the following pointers.

1. **What is Linq Sum in C#?**
2. **Multiple examples using both Method and Query syntax.**

##### ****What is Linq Sum in C#?****

The Linq Sum() Method belongs to the category of Aggregate Functions. The Linq Sum method in C# is used to calculates the total or sum of numeric values in the collection. Let us understand the Sum() method with some examples.

##### ****Example1:****

The following example calculates the sum of all integers present in the collection.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 10, 30, 50, 40, 60, 20, 70, 90, 80, 100 **}**;

//Using Method Syntax

**int** MSTotal = intNumbers.Sum**()**;

//Using Query Syntax

**int** QSTotal = **(from** num in intNumbers

**select** num**)**.Sum**()**;

Console.WriteLine**(**"Sum = " + QSTotal**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Sum Method in C# with Examples

Note: We don’t have any operator called sum in Linq query syntax. So here we need to use the mixed syntax.

##### ****Example2: Linq Sum Method with filter****

Now we need to calculate the sum of all numbers which is greater than 50.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 10, 30, 50, 40, 60, 20, 70, 90, 80, 100 **}**;

//Using Method Syntax

**int** MSTotal = intNumbers.Where**(**num =**>** num **>** 50**)**.Sum**()**;

//Using Query Syntax

**int** QSTotal = **(from** num in intNumbers

**where** num **>** 50

**select** num**)**.Sum**()**;

Console.WriteLine**(**"Sum = " + QSTotal**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

C# linq sum

##### ****Example3: Linq Sum Method with Predicate****

Instead of using the where method to filter the data, you can also use the other overloaded version of the Sum method which takes a Predicate and within that predicate, you can write the logic to filter the data as shown in the below example.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 10, 30, 50, 40, 60, 20, 70, 90, 80, 100 **}**;

//Using Method Syntax with a Predicate

**int** MSTotal = intNumbers.Sum**(**num =**>** **{**

**if** **(**num **>** 50**)**

**return** num;

**else**

**return** 0;

**})**;

Console.WriteLine**(**"Sum = " + MSTotal**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Linq sum

##### ****Linq Sum Method Working with Complex Type:****

We are going to work with the following Employee class.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** string Department **{** **get**; **set**; **}**

**public** **static** List**<**Employee**>** GetAllEmployees**()**

**{**

List**<**Employee**>** listStudents = new List**<**Employee**>()**

**{**

new Employee**{**ID= 101,Name = "Preety", Salary = 10000, Department = "IT"**}**,

new Employee**{**ID= 102,Name = "Priyanka", Salary = 15000, Department = "Sales"**}**,

new Employee**{**ID= 103,Name = "James", Salary = 50000, Department = "Sales"**}**,

new Employee**{**ID= 104,Name = "Hina", Salary = 20000, Department = "IT"**}**,

new Employee**{**ID= 105,Name = "Anurag", Salary = 30000, Department = "IT"**}**,

new Employee**{**ID= 106,Name = "Sara", Salary = 25000, Department = "IT"**}**,

new Employee**{**ID= 107,Name = "Pranaya", Salary = 35000, Department = "IT"**}**,

new Employee**{**ID= 108,Name = "Manoj", Salary = 11000, Department = "Sales"**}**,

new Employee**{**ID= 109,Name = "Sam", Salary = 45000, Department = "Sales"**}**,

new Employee**{**ID= 110,Name = "Saurav", Salary = 25000, Department = "Sales"**}**

**}**;

**return** listStudents;

**}**

**}**

**}**

This is a very simple Employee class with having four properties such as **ID, Name, Salary,**and **Department**. We also create one method i.e. **GetAllEmployees()** which will return the list of all the employees.

##### ****Example4:****

The following example calculates the Sum of Salaries of all the employees.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var TotalSalaryMS = Employee.GetAllEmployees**()**

.Sum**(**emp =**>** emp.Salary**)**;

//Using Query Syntax

var TotalSalaryQS = **(from** emp in Employee.GetAllEmployees**()**

**select** emp**)**.Sum**(**e =**>** e.Salary**)**;

Console.WriteLine**(**"Sum Of Salary = " + TotalSalaryMS**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

C# linq sum

##### ****Example5:****

The following example calculates the sum of the salary of all the employees who belong to the IT department.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var TotalSalaryMS = Employee.GetAllEmployees**()**

.Where**(**emp =**>** emp.Department == "IT"**)**

.Sum**(**emp =**>** emp.Salary**)**;

//Using Query Syntax

var TotalSalaryQS = **(from** emp in Employee.GetAllEmployees**()**

**where** emp.Department == "IT"

**select** emp**)**.Sum**(**e =**>** e.Salary**)**;

Console.WriteLine**(**"IT Department Total Salary = " + TotalSalaryQS**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Linq sum in C# with examples

##### ****Example6:****

Let’s rewrite the previous example using custom predicate.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax and Predicate

var TotalSalaryMS = Employee.GetAllEmployees**()**

.Sum**(**emp =**>** **{**

**if** **(**emp.Department == "IT"**)**

**return** emp.Salary;

**else**

**return** 0;

**})**;

Console.WriteLine**(**"IT Department Total Salary = " + TotalSalaryMS**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

C# linq sum examples

# Linq Max in C#

## ****Linq Max in C# with Examples****

In this article, I am going to discuss the **Linq Max in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Sum Aggregate Functions in C#**](https://dotnettutorials.net/lesson/linq-sum-method/) with some examples. As part of this article, we are going to discuss the following pointers.

1. **What is Linq Max in C#?**
2. **Multiple examples using both Method and Query syntax.**

##### ****What is Linq Max in C#?****

The Linq Max in C# is used to returns the largest numeric value from the collection on which it is applied. Let us understand the Max() method with some examples.

##### ****Example1:****

The following example returns the largest number from the collection.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 10, 80, 50, 90, 60, 30, 70, 40, 20, 100 **}**;

//Using Method Syntax

**int** MSLergestNumber = intNumbers.Max**()**;

//Using Query Syntax

**int** QSLergestNumber = **(from** num in intNumbers

**select** num**)**.Max**()**;

Console.WriteLine**(**"Largest Number = " + MSLergestNumber**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

Once you run the application, it will display the output as 100.

**Note:** We don’t have any operator called Max in Linq query syntax. So here we need to use the mixed syntax.

##### ****Example2: Linq Max with filter in C#****

Now we need to return the largest number from the collection where the number is less than 50.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 10, 80, 50, 90, 60, 30, 70, 40, 20, 100 **}**;

//Using Method Syntax

**int** MSLergestNumber = intNumbers.Where**(**num =**>** num **<** 50**)**.Max**()**;

//Using Query Syntax

**int** QSLergestNumber = **(from** num in intNumbers

**where** num **<** 50

**select** num**)**.Max**()**;

Console.WriteLine**(**"Largest Number = " + MSLergestNumber**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

It will print the output as 40.

##### ****Example3: Linq Max with Predicate****

Instead of using the where method to filter the data, you can also use the other overloaded version of the Max method which takes a Predicate and within that predicate, you can write the logic to filter the data as shown in the below example.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 10, 80, 50, 90, 60, 30, 70, 40, 20, 100 **}**;

//Using Method Syntax

**int** MSLergestNumber = intNumbers.Max**(**num =**>** **{**

**if** **(**num **<** 50**)**

**return** num;

**else**

**return** 0;

**})**;

Console.WriteLine**(**"Largest Number = " + MSLergestNumber**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

It should give the same output as the previous example.

##### ****Max Method Working with Complex Type in C#:****

We are going to work with the following Employee class.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** string Department **{** **get**; **set**; **}**

**public** **static** List**<**Employee**>** GetAllEmployees**()**

**{**

List**<**Employee**>** listStudents = new List**<**Employee**>()**

**{**

new Employee**{**ID= 101,Name = "Preety", Salary = 10000, Department = "IT"**}**,

new Employee**{**ID= 102,Name = "Priyanka", Salary = 15000, Department = "Sales"**}**,

new Employee**{**ID= 103,Name = "James", Salary = 50000, Department = "Sales"**}**,

new Employee**{**ID= 104,Name = "Hina", Salary = 20000, Department = "IT"**}**,

new Employee**{**ID= 105,Name = "Anurag", Salary = 30000, Department = "IT"**}**,

new Employee**{**ID= 106,Name = "Sara", Salary = 25000, Department = "IT"**}**,

new Employee**{**ID= 107,Name = "Pranaya", Salary = 35000, Department = "IT"**}**,

new Employee**{**ID= 108,Name = "Manoj", Salary = 11000, Department = "Sales"**}**,

new Employee**{**ID= 109,Name = "Sam", Salary = 45000, Department = "Sales"**}**,

new Employee**{**ID= 110,Name = "Saurav", Salary = 25000, Department = "Sales"**}**

**}**;

**return** listStudents;

**}**

**}**

**}**

In the above Employee class, we define four properties such as ID, Name, Salary, and Department along with the GetAllEmployees() which will return the list of all the employees.

##### ****Example4:****

The following example returns the highest salary among all the employees.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSHighestSalary = Employee.GetAllEmployees**()**

.Max**(**emp =**>** emp.Salary**)**;

//Using Query Syntax

var QSHighestSalary = **(from** emp in Employee.GetAllEmployees**()**

**select** emp**)**.Max**(**e =**>** e.Salary**)**;

Console.WriteLine**(**"Highest Salary = " + QSHighestSalary**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Linq Max in C# with Examples

##### ****Example5:****

The following example returns the Highest Salary of IT department.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSHighestSalary = Employee.GetAllEmployees**()**

.Where**(**emp =**>** emp.Department == "IT"**)**

.Max**(**emp =**>** emp.Salary**)**;

//Using Query Syntax

var QSHighestSalary = **(from** emp in Employee.GetAllEmployees**()**

**where** emp.Department == "IT"

**select** emp**)**.Max**(**e =**>** e.Salary**)**;

Console.WriteLine**(**"It Department Highest Salary = " + QSHighestSalary**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Linq Max in C#

##### ****Example6:****

Let’s rewrite the previous example using a custom predicate.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSHighestSalary = Employee.GetAllEmployees**()**

.Max**(**emp =**>** **{**

**if** **(**emp.Department == "IT"**)**

**return** emp.Salary;

**else**

**return** 0;

**})**;

Console.WriteLine**(**"It Department Highest Salary = " + MSHighestSalary**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

It will give the same output as the previous example.

# Linq Min Method in C#

## ****Linq Min Method in C# with Examples****

In this article, I am going to discuss the **Linq Min Method in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Linq Max Aggregate Functions in C#**](https://dotnettutorials.net/lesson/linq-max-method/) with some examples. As part of this article, we are going to discuss the following pointers.

1. **What is Linq Min Method in C#?**
2. **Multiple examples using both Method and Query syntax.**

##### ****What is Linq Min Method in C#?****

The Linq Min method is used to returns the lowest numeric value from the collection on which it is applied. Let us understand the Min() method with some examples.

##### ****Example1:****

The following example returns the lowest number from the collection.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 60, 80, 50, 90, 10, 30, 70, 40, 20, 100 **}**;

//Using Method Syntax

**int** MSLowestNumber = intNumbers.Min**()**;

//Using Query Syntax

**int** QSLowestNumber = **(from** num in intNumbers

**select** num**)**.Min**()**;

Console.WriteLine**(**"Lowest Number = " + MSLowestNumber**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

Once you run the application, it will display the output as 10.

**Note:** We don’t have any operator called Min in Linq to write the query syntax. So here we need to use the mixed syntax.

##### ****Example2: Min with filter****

Now we need to return the lowest number from the collection where the number is greater than 50.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 60, 80, 50, 90, 10, 30, 70, 40, 20, 100 **}**;

//Using Method Syntax

**int** MSLowestNumber = intNumbers.Where**(**num =**>** num **>** 50**)**.Min**()**;

//Using Query Syntax

**int** QSLowestNumber = **(from** num in intNumbers

**where** num **>** 50

**select** num**)**.Min**()**;

Console.WriteLine**(**"Lowest Number = " + MSLowestNumber**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

It will print the output as 60.

##### ****Example3: Min Method with Predicate****

Instead of using the where method to filter the data, you can also use the other overloaded version of the Min method which takes one Predicate and within that predicate, you can write your logic to filter the data as shown in the below example.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 60, 80, 50, 90, 10, 30, 70, 40, 20, 100 **}**;

//Using Method Syntax

**int** MSLowestNumber = intNumbers.Where**(**num =**>** num **>** 50**)**.Min**(**num =**>** **{**

**if** **(**num **>** 50**)**

**return** num;

**else**

**return** 0;

**})**;

Console.WriteLine**(**"Lowest Number = " + MSLowestNumber**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

It should give the same output as the previous example.

##### ****Working with Complex Type:****

We are going to work with the following Employee class.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** string Department **{** **get**; **set**; **}**

**public** **static** List**<**Employee**>** GetAllEmployees**()**

**{**

List**<**Employee**>** listStudents = new List**<**Employee**>()**

**{**

new Employee**{**ID= 101,Name = "Preety", Salary = 10000, Department = "IT"**}**,

new Employee**{**ID= 102,Name = "Priyanka", Salary = 15000, Department = "Sales"**}**,

new Employee**{**ID= 103,Name = "James", Salary = 50000, Department = "Sales"**}**,

new Employee**{**ID= 104,Name = "Hina", Salary = 20000, Department = "IT"**}**,

new Employee**{**ID= 105,Name = "Anurag", Salary = 30000, Department = "IT"**}**,

new Employee**{**ID= 106,Name = "Sara", Salary = 25000, Department = "IT"**}**,

new Employee**{**ID= 107,Name = "Pranaya", Salary = 35000, Department = "IT"**}**,

new Employee**{**ID= 108,Name = "Manoj", Salary = 11000, Department = "Sales"**}**,

new Employee**{**ID= 109,Name = "Sam", Salary = 45000, Department = "Sales"**}**,

new Employee**{**ID= 110,Name = "Saurav", Salary = 25000, Department = "Sales"**}**

**}**;

**return** listStudents;

**}**

**}**

**}**

In the above Employee class, we define four properties such as ID, Name, Salary, and Department along with the GetAllEmployees() which will return the list of all the employees.

##### ****Example4:****

The following example returns the lowest salary among all the employees.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSLowestSalary = Employee.GetAllEmployees**()**

.Min**(**emp =**>** emp.Salary**)**;

//Using Query Syntax

var QSLowestSalary = **(from** emp in Employee.GetAllEmployees**()**

**select** emp**)**.Min**(**e =**>** e.Salary**)**;

Console.WriteLine**(**"Lowest Salary = " + MSLowestSalary**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Linq Min Method in C# with Examples

##### ****Example5:****

The following example returns the Lowest Salary of IT department.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSLowestSalary = Employee.GetAllEmployees**()**

.Where**(**emp =**>** emp.Department == "IT"**)**

.Min**(**emp =**>** emp.Salary**)**;

//Using Query Syntax

var QSLowestSalary = **(from** emp in Employee.GetAllEmployees**()**

**where** emp.Department == "IT"

**select** emp**)**.Min**(**e =**>** e.Salary**)**;

Console.WriteLine**(**"IT Department Lowest Salary = " + QSLowestSalary**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Linq Min Method in C# with Examples

# Linq Average Method in C#

# ****Linq Average Method in C# with Examples****

In this article, I am going to discuss the **Linq Average Method in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Linq Min Aggregate Function**](https://dotnettutorials.net/lesson/linq-min-method/) with some examples. As part of this article, we are going to discuss the following pointers.

1. **What is Linq Average Method in C#?**
2. **Multiple examples using both Method and Query syntax.**

##### ****What is Linq Average Method in C#?****

The Linq Average method is used to calculate the average of numeric values from the collection on which it is applied. This Average method can return nullable or non-nullable decimal, float or double value. Let us understand the Average() method with some examples.

##### ****Example1:****

The following example calculates the average value of all the integers present in the collection.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 60, 80, 50, 90, 10, 30, 70, 40, 20, 100 **}**;

//Using Method Syntax

var MSAverageValue = intNumbers.Average**()**;

//Using Query Syntax

var QSAverageValue = **(from** num in intNumbers

**select** num**)**.Average**()**;

Console.WriteLine**(**"Average Value = " + MSAverageValue**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

C# Average Method with Examples

**Note:** We don’t have any operator called Average in Linq to write the query syntax. So here we need to use the mixed syntax.

##### ****Example2: Linq Average Method with filter****

Now we need to return the average value from the collection where the number is greater than 50.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 60, 80, 50, 90, 10, 30, 70, 40, 20, 100 **}**;

//Using Method Syntax

var MSAverageValue = intNumbers.Where**(**num =**>** num **>** 50**)**.Average**()**;

//Using Query Syntax

var QSAverageValue = **(from** num in intNumbers

**where** num **>** 50

**select** num**)**.Average**()**;

Console.WriteLine**(**"Average Value = " + MSAverageValue**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

It will print the output as 80.

##### ****Working with Complex Type:****

We are going to work with the following Employee class.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** string Department **{** **get**; **set**; **}**

**public** **static** List**<**Employee**>** GetAllEmployees**()**

**{**

List**<**Employee**>** listStudents = new List**<**Employee**>()**

**{**

new Employee**{**ID= 101,Name = "Preety", Salary = 10000, Department = "IT"**}**,

new Employee**{**ID= 102,Name = "Priyanka", Salary = 15000, Department = "Sales"**}**,

new Employee**{**ID= 103,Name = "James", Salary = 50000, Department = "Sales"**}**,

new Employee**{**ID= 104,Name = "Hina", Salary = 20000, Department = "IT"**}**,

new Employee**{**ID= 105,Name = "Anurag", Salary = 30000, Department = "IT"**}**,

new Employee**{**ID= 106,Name = "Sara", Salary = 25000, Department = "IT"**}**,

new Employee**{**ID= 107,Name = "Pranaya", Salary = 35000, Department = "IT"**}**,

new Employee**{**ID= 108,Name = "Manoj", Salary = 11000, Department = "Sales"**}**,

new Employee**{**ID= 109,Name = "Sam", Salary = 45000, Department = "Sales"**}**,

new Employee**{**ID= 110,Name = "Saurav", Salary = 25000, Department = "Sales"**}**

**}**;

**return** listStudents;

**}**

**}**

**}**

We create the Employee class with four properties such as ID, Name, Salary, and Department along with the GetAllEmployees() which in turn going to return the list of all the employees.

##### ****Example3:****

The following example returns the average salary of all the employees.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSAverageSalary = Employee.GetAllEmployees**()**

.Average**(**emp =**>** emp.Salary**)**;

//Using Query Syntax

var QSAverageSalary = **(from** emp in Employee.GetAllEmployees**()**

**select** emp**)**.Average**(**e =**>** e.Salary**)**;

Console.WriteLine**(**"Average Salary = " + MSAverageSalary**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Average Method in C# with Examples

##### ****Example4:****

The following example calculates the Average Salary of the IT department.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSAverageSalary = Employee.GetAllEmployees**()**

.Where**(**emp =**>** emp.Department == "IT"**)**

.Average**(**emp =**>** emp.Salary**)**;

//Using Query Syntax

var QSAverageSalary = **(from** emp in Employee.GetAllEmployees**()**

**where** emp.Department == "IT"

**select** emp**)**.Average**(**e =**>** e.Salary**)**;

Console.WriteLine**(**"IT Department Average Salary = " + MSAverageSalary**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Linq Average Method in C# with Examples

# Linq Count Method in C#

## ****Linq Count Method in C# with Examples****

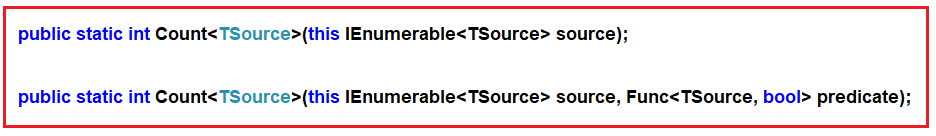
In this article, I am going to discuss the **Linq Count Method in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Linq Average Aggregate method**](https://dotnettutorials.net/lesson/linq-average-method/) with some examples. As part of this article, we are going to discuss the following pointers.

1. **What is Linq Count Method in C#?**
2. **Multiple examples using both Method and Query syntax.**

##### ****What is Linq Count Method in C#?****

The Linq Count Method used to return the number of elements present in the collection or the number of elements that have satisfied a given condition.

There are two overloaded versions of the Linq Count() extension method is available as shown below.



As you can see from the above definitions, the first overloaded version does not take any parameter and it simply returns the count of the number of elements present in the collections. On the other hand, the second overloaded version takes one predicate as a parameter and then it returns the count of the number of elements which have satisfied the given condition which we can specify using a lambda expression or by using a predicate function.

The return type of Count() function is always going to be int. Let us understand the Count() method with some examples.

##### ****Example1:****

The following example returns the number of elements present in the collection.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 60, 80, 50, 90, 10, 30, 70, 40, 20, 100 **}**;

//Using Method Syntax

**int** MSCount = intNumbers.Count**()**;

//Using Query Syntax

var QSCount = **(from** num in intNumbers

**select** num**)**.Count**()**;

Console.WriteLine**(**"No of Elements = " + MSCount**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Count Method in Linq C#

**Note:** We don’t have any operator called Count in Linq to write the query syntax. So here we need to use the mixed syntax.

##### ****Example2: Count with filter****

Now we need to return the number of elements present in the collection which are greater than 40.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = new **int[]** **{** 60, 80, 50, 90, 10, 30, 70, 40, 20, 100 **}**;

//Using Method Syntax

**int** MSCount = intNumbers.Where**(**num =**>** num **>** 40**)**.Count**()**;

//Using Query Syntax

var QSCount = **(from** num in intNumbers

**where** num **>** 40

**select** num**)**.Count**()**;

Console.WriteLine**(**"No of Elements = " + MSCount**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Count Method in C# with Examples

##### ****Working with Complex Type:****

We are going to work with the following Employee class.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** string Department **{** **get**; **set**; **}**

**public** **static** List**<**Employee**>** GetAllEmployees**()**

**{**

List**<**Employee**>** listStudents = new List**<**Employee**>()**

**{**

new Employee**{**ID= 101,Name = "Preety", Salary = 10000, Department = "IT"**}**,

new Employee**{**ID= 102,Name = "Priyanka", Salary = 15000, Department = "Sales"**}**,

new Employee**{**ID= 103,Name = "James", Salary = 50000, Department = "Sales"**}**,

new Employee**{**ID= 104,Name = "Hina", Salary = 20000, Department = "IT"**}**,

new Employee**{**ID= 105,Name = "Anurag", Salary = 30000, Department = "IT"**}**,

new Employee**{**ID= 106,Name = "Sara", Salary = 25000, Department = "IT"**}**,

new Employee**{**ID= 107,Name = "Pranaya", Salary = 35000, Department = "IT"**}**,

new Employee**{**ID= 108,Name = "Manoj", Salary = 11000, Department = "Sales"**}**,

new Employee**{**ID= 109,Name = "Sam", Salary = 45000, Department = "Sales"**}**,

new Employee**{**ID= 110,Name = "Saurav", Salary = 25000, Department = "Sales"**}**

**}**;

**return** listStudents;

**}**

**}**

**}**

We create the Employee class with four properties such as ID, Name, Salary, and Department along with the GetAllEmployees() which in turn going to return the list of all the employees.

##### ****Example3:****

The following example returns the number of employees.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSCount = Employee.GetAllEmployees**()**.Count**()**;

//Using Query Syntax

var QSCount = **(from** emp in Employee.GetAllEmployees**()**

**select** emp**)**.Count**()**;

Console.WriteLine**(**"Total No of Employees = " + QSCount**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

C# Count Method with Examples

##### ****Example4:****

The following example returns the number of employees in the IT department.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSCount = Employee.GetAllEmployees**()**

.Where**(**emp =**>** emp.Department == "IT"**)**

.Count**()**;

//Using Query Syntax

var QSCount = **(from** emp in Employee.GetAllEmployees**()**

**where** emp.Department == "IT"

**select** emp**)**.Count**()**;

Console.WriteLine**(**"Total No of Employees of IT Department = " + QSCount**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Count Method in Linq

# Linq Aggregate Method in C#

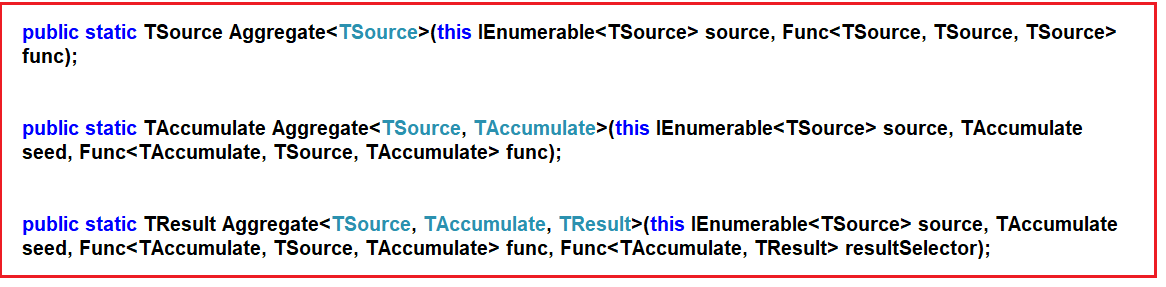
## ****Linq Aggregate Method in C# with Examples****

In this article, I am going to discuss the **Linq Aggregate Method in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Linq Count method**](https://dotnettutorials.net/lesson/linq-count-method/) with some examples. As part of this article, we are going to discuss the following pointers.

1. **What is Linq Aggregate Method in C#?**
2. **Multiple examples using both Method and Query syntax.**

##### ****What is Linq Aggregate Method in C#?****

The Linq Aggregate extension method performs an accumulative operation. There are three overloaded versions of this method is available in **System.Linq** namespace as shown in the below image.



Let us understand the use of the Aggregate method with some examples.

##### ****Example1: Comma-separated string.****

Let us consider we have the following string array

**string[] skills = { “C#.NET”, “MVC”, “WCF”, “SQL”, “LINQ”,  “ASP.NET”};**

Our requirement is to combine all the above strings present in the skill array into a single comma-separated string as shown below.

**C#.NET, MVC, WCF, SQL, LINQ, ASP.NET**

**Program without using Linq Aggregate method:**

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** skills = **{** "C#.NET", "MVC", "WCF", "SQL", "LINQ", "ASP.NET" **}**;

string result = string.Empty;

**foreach(**string skill in skills**)**

**{**

result = result + skill + ", ";

**}**

//Find the index position of last comma

**int** lastIndex = result.LastIndexOf**(**","**)**;

//Remove the last comma

result = result.Remove**(**lastIndex**)**;

Console.WriteLine**(**result**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Linq Aggregate Method in C# with Examples

Now let us see how to achieve the same output using the Linq Aggregate method.

##### ****Program using Linq Aggregate Method:****

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** skills = **{** "C#.NET", "MVC", "WCF", "SQL", "LINQ", "ASP.NET" **}**;

string result = skills.Aggregate**((**s1, s2**)** =**>** s1 + ", " + s2**)**;

Console.WriteLine**(**result**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

C# Aggregate Method

##### ****How does the above Aggregate Method work?****

The lambda expression **(s1, s2) => s1 + “, ” + s2** will be treated like **s1 = s1 + “, ” + s2** where s1 will be accumulated for each item present in the collection. As a result, the Aggregate function will return the comma-separated string. Please have a look at how the comma-separated string is generated step by step.

1. **Step1**. First “**C#.NET**” is concatenated with “MVC” to produce the result “**C#.NET, MVC**“.
2. **Step2**. Result in Step 1 i.e. “**C#.NET, MVC**” is then concatenated with “**WCF**” to produce the result “**C#.NET, MVC, WCF**“.
3. **Step3**: Result in Step 2 i.e. “**C#.NET, MVC, WCF**” is then concatenated with “**SQL**” to produce the result “**C#.NET, MVC, WCF, SQL**“.
4. **Step4**: Result in Step 3 i.e. “**C#.NET, MVC, WCF, SQL**” is then concatenated with “**LINQ**” to produce the result “**C#.NET, MVC, WCF, SQL, LINQ**“.
5. **Step5**: Result in Step 4 i.e. “**C#.NET, MVC, WCF, SQL, LINQ**” is then concatenated with “**ASP.NET**” to produce the final result “**C#.NET, MVC, WCF, SQL, LINQ, ASP.NET**” what we see in the output.

##### ****Example2: Product of integer numbers****

Consider we have the following integer array

**int[] intNumbers = { 3, 5, 7, 9 };**

Our requirement is to compute the product of all numbers.

##### ****Program without using Aggregate Method:****

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = **{** 3, 5, 7, 9 **}**;

**int** result = 1;

**foreach(int** num in intNumbers**)**

**{**

result = result \* num;

**}**

Console.WriteLine**(**result**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

##### ****Program using Aggregate Method:****

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = **{** 3, 5, 7, 9 **}**;

**int** result = intNumbers.Aggregate**((**n1, n2**)** =**>** n1 \* n2**)**;

Console.WriteLine**(**result**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**How does the above Aggregate Method work?**

1. **Step1**: First it multiplies (3X5) to produce the result as 15
2. **Step2**: Result of Step 1 i.e. 15 is then multiplied with 7 to produce the result as 105
3. **Step3**: Result of Step 2 i.e. 105 is then multiplied with 9 to produce the final result as 945.

##### ****Aggregate Method with the seed parameter:****

The second overloaded version of the Aggregate method takes the first parameter as the seed value to accumulate. The Second parameter is Func type delegate: Let us understand the use of the seed parameter with an example. Let us see how to pass the seed value as 2 with our previous example.

**int result = intNumbers.Aggregate(2, (n1, n2) => n1 \* n2);**

**The complete example is given below.**

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** intNumbers = **{** 3, 5, 7, 9 **}**;

**int** result = intNumbers.Aggregate**(**2, **(**n1, n2**)** =**>** n1 \* n2**)**;

Console.WriteLine**(**result**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 1980**

##### ****How does it work?****

1. Step1: First it multiplies (2\*3) to produce the result as 6
2. Step2: Result of Step 1 i.e. 6 is then multiplied with 5 to produce the result as 30
3. Step3: Result of Step 2 i.e. 30 is then multiplied with 7 to produce the result as 210.
4. Step4: Result of Step 3 i.e. 210 is then multiplied with 9 to produce the final result as 1890.

##### ****Aggregate Method with Complex Type:****

We are going to work with the following Employee class.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**public** string Department **{** **get**; **set**; **}**

**public** **static** List**<**Employee**>** GetAllEmployees**()**

**{**

List**<**Employee**>** listStudents = new List**<**Employee**>()**

**{**

new Employee**{**ID= 101,Name = "Preety", Salary = 10000, Department = "IT"**}**,

new Employee**{**ID= 102,Name = "Priyanka", Salary = 15000, Department = "Sales"**}**,

new Employee**{**ID= 103,Name = "James", Salary = 50000, Department = "Sales"**}**,

new Employee**{**ID= 104,Name = "Hina", Salary = 20000, Department = "IT"**}**,

new Employee**{**ID= 105,Name = "Anurag", Salary = 30000, Department = "IT"**}**,

**}**;

**return** listStudents;

**}**

**}**

**}**

**Note:** While working with the complex type we need to use either the second third overloaded version of the Aggregate method:

##### ****Example3:****

The following example uses the Aggregate method to add the salary of all the employees.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** Salary = Employee.GetAllEmployees**()**

.Aggregate**<**Employee, **int>(**0,

**(**TotalSalary, emp**)** =**>** TotalSalary += emp.Salary**)**;

Console.WriteLine**(**Salary**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

Please note here we passed the seed value as 0. Once you run the application, it gives you the output as expected.

##### ****Example4:****

In the following example, we pass a string as the seed value to the Aggregate extension method. Here the seed value is “**Employee Names**”.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string CommaSeparatedEmployeeNames = Employee.GetAllEmployees**()**.Aggregate**<**Employee, string**>(**

"Employee Names : ", // seed value

**(**employeeNames, employee**)** =**>** employeeNames = employeeNames + employee.Name + ", "**)**;

**int** LastIndex = CommaSeparatedEmployeeNames.LastIndexOf**(**","**)**;

CommaSeparatedEmployeeNames = CommaSeparatedEmployeeNames.Remove**(**LastIndex**)**;

Console.WriteLine**(**CommaSeparatedEmployeeNames**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: Employee Names : Preety, Priyanka, James, Hina, Anurag**

In the above example, the first parameter of the Aggregate method is the “**Employee Names:** ” string that will be accumulated with all employee names. The comma in the lambda expression will be passed as the second parameter.

##### ****Aggregate Method with Result Selector****

The third overload version requires the third parameter of the Func delegate expression for the result selector so that we can formulate the result. In our previous example, once we get the comma-separated string then we remove the last comma using some additional logic. Let us see how we can do the same using the third parameter.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string CommaSeparatedEmployeeNames = Employee.GetAllEmployees**()**.Aggregate**<**Employee, string, string**>(**

"Employee Names : ", // seed value

**(**employeeNames, employee**)** =**>** employeeNames = employeeNames + employee.Name + ",",

employeeNames =**>** employeeNames.Substring**(**0, employeeNames.Length - 1**))**;

Console.WriteLine**(**CommaSeparatedEmployeeNames**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

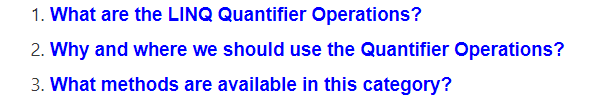
**Output: Employee Names : Preety,Priyanka,James,Hina,Anurag**

In the above example, we have specified a lambda expression i.e. **employeeNames => employeeNames.Substring(0, employeeNames.Length – 1)** to remove the last comma in the string result.

# LINQ Quantifiers Operators

# ****LINQ Quantifiers Operators in C#****

In this article, I am going to discuss the **LINQ Quantifiers Operators**in C# with examples. Please read our previous article before proceeding to this article where we discussed the [**LINQ Aggregate Method in C#**](https://dotnettutorials.net/lesson/linq-aggregate-method/)with some examples. As part of this article, we are going to discuss the following pointers in detail.



##### ****What are the LINQ Quantifier Operations?****

We need to use the LINQ Quantifier Operators on a data source when we want to check if some or all of the elements of that data source satisfy a condition or not. That means, here we have a data source and also we have a condition. Then we need to check whether all or some of the elements of that data source satisfied the condition or not.

All the methods in quantifier operations are always going to return a Boolean value. That means if the all or some of the elements in the data source satisfy the given condition then it is going to return true else it is going to return false.

Note: The condition that we specify may be for some or all of the elements.

##### ****Examples:****

We need to use the quantifier operations on the below scenarios.

1. When we want to check whether all the employees are present on a specific day or not
2. In a specific class of students, we need to check if any of the students having marks more than 90%
3. When we need to check if any of the employees having the names James is present or not

##### ****What methods are available in this category?****

The following three methods belong to the Quantifier Operations category.

[**All**](https://dotnettutorials.net/lesson/linq-all-operator/)**:** This specifies whether all the elements of a data source satisfy a given condition or not.

[**Any**](https://dotnettutorials.net/lesson/linq-any-operator/)**:** This specifies whether at least one of the elements of a data source satisfies the condition or not.

[**Contains**](https://dotnettutorials.net/lesson/linq-contains-method/)**:** This method is used to check whether the data source contains a specified element or not.

**Note:** All of the above three methods return Boolean true or false depending on whether all or some of the elements in a data source satisfy a condition.

# Linq All Operator in C#

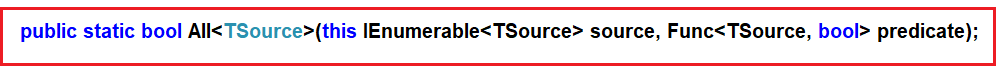
## ****Linq All Operator in C# with Examples****

In this article, I am going to discuss the **Linq All Operator in C#** with examples. Please read our previous article before proceeding to this article where we discussed the basics of [**Quantifiers Operators**](https://dotnettutorials.net/lesson/linq-quantifiers-operators/). As part of this article, we are going to discuss the following pointers related to Linq All Operator.

1. **What is Linq All Operator?**
2. **Multiple examples using both Method and Query syntax.**

##### ****What is Linq All Operator in C#?****

The **Linq All Operator in C#** is used to check whether all the elements of a data source satisfy a given condition or not. If all the elements satisfy the condition, then it returns true else return false. There is no overloaded version is available for the All method. The definition is given below.



As you can see from the above image, the Linq ALL extension method takes one predicate as a parameter.

##### ****Example1:****

The following example returns true as all the elements are greater than 10 in the integer array.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** IntArray = **{** 11, 22, 33, 44, 55 **}**;

var Result = IntArray.All**(**x =**>** x **>** 10**)**;

Console.WriteLine**(**"Is All Numbers are greater than 10 : " + Result**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

C# All Operator

##### ****Example2:****

The following example returns true as all the names are not greater than 5 characters.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** stringArray = **{** "James", "Sachin", "Sourav", "Pam", "Sara" **}**;

var Result = stringArray.All**(**name =**>** name.Length **>** 5**)**;

Console.WriteLine**(**"Is All Names are greater than 5 Characters : " + Result**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

All Operator in Linq

##### ****Working with Complex Type:****

We are going to work with the following Student and Subject class. So, create a class file with the name **Student.cs** and then copy and paste the following code.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** TotalMarks **{** **get**; **set**; **}**

**public** List**<**Subject**>** Subjects **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetAllStudnets**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**{**ID= 101,Name = "Preety", TotalMarks = 265,

Subjects = new List**<**Subject**>()**

**{**

new Subject**(){**SubjectName = "Math", Marks = 80**}**,

new Subject**(){**SubjectName = "Science", Marks = 90**}**,

new Subject**(){**SubjectName = "English", Marks = 95**}**

**}}**,

new Student**{**ID= 102,Name = "Priyanka", TotalMarks = 278,

Subjects = new List**<**Subject**>()**

**{**

new Subject**(){**SubjectName = "Math", Marks = 90**}**,

new Subject**(){**SubjectName = "Science", Marks = 95**}**,

new Subject**(){**SubjectName = "English", Marks = 93**}**

**}}**,

new Student**{**ID= 103,Name = "James", TotalMarks = 240,

Subjects = new List**<**Subject**>()**

**{**

new Subject**(){**SubjectName = "Math", Marks = 70**}**,

new Subject**(){**SubjectName = "Science", Marks = 80**}**,

new Subject**(){**SubjectName = "English", Marks = 90**}**

**}}**,

new Student**{**ID= 104,Name = "Hina", TotalMarks = 275,

Subjects = new List**<**Subject**>()**

**{**

new Subject**(){**SubjectName = "Math", Marks = 90**}**,

new Subject**(){**SubjectName = "Science", Marks = 90**}**,

new Subject**(){**SubjectName = "English", Marks = 95**}**

**}}**,

new Student**{**ID= 105,Name = "Anurag", TotalMarks = 255,

Subjects = new List**<**Subject**>()**

**{**

new Subject**(){**SubjectName = "Math", Marks = 80**}**,

new Subject**(){**SubjectName = "Science", Marks = 90**}**,

new Subject**(){**SubjectName = "English", Marks = 85**}**

**}**

**}**,

**}**;

**return** listStudents;

**}**

**}**

**public** **class** Subject

**{**

**public** string SubjectName **{** **get**; **set**; **}**

**public** **int** Marks **{** **get**; **set**; **}**

**}**

**}**

##### ****Example3:****

Check whether all the students having total marks greater than 250. As you can see the student James total mark is 240 which is less than 250. So here it will give you the output as false.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

**bool** MSResult = Student.GetAllStudnets**()**.All**(**std =**>** std.TotalMarks **>** 250**)**;

//Using Query Syntax

**bool** QSResult = **(from** std in Student.GetAllStudnets**()**

**select** std**)**.All**(**std =**>** std.TotalMarks **>** 250**)**;

Console.WriteLine**(**MSResult**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:** False

##### ****Example4:****

Now we need to fetch all the student details whose mark on each subject is greater than 80.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSResult = Student.GetAllStudnets**()**

.Where**(**std =**>** std.Subjects.All**(**x =**>** x.Marks **>** 80**))**.ToList**()**;

//Using Query Syntax

var QSResult = **(from** std in Student.GetAllStudnets**()**

**where** std.Subjects.All**(**x =**>** x.Marks **>** 80**)**

**select** std**)**.ToList**()**;

**foreach** **(**var item in QSResult**)**

**{**

Console.WriteLine**(**item.Name + " " + item.TotalMarks**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

LINQ All Operator

# Linq Any in C#

## ****Linq Any in C# with Examples****

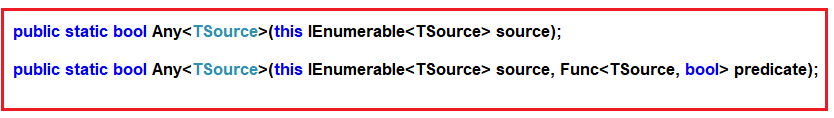
In this article, I am going to discuss the **Linq Any in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**LINQ ALL Operator**](https://dotnettutorials.net/lesson/linq-all-operator/) with some different kinds of examples. As part of this article, we are going to discuss the following pointers.

1. **What is Linq Any in C#?**
2. **Multiple examples using both Method and Query syntax.**

##### ****What is Linq Any in C#?****

The **C# Linq Any Operator** is used to check whether at least one of the elements of a data source satisfies a given condition or not. If any of the elements satisfy the given condition, then it returns true else return false.

It is also used to check whether a collection contains some data or not. That means it checks the length of the collection also. If it contains any data then it returns true else return false. There are two overloaded versions of this method is available. They are as follows.



As you can see from the above image, the first overloaded version of ANY extension method does not take any parameter while the other overloaded version takes a predicate as a parameter.

##### ****Example1: Using First Overloaded Version of Linq Any Operator****

The following example returns true as the collection contains at least one element.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** IntArray = **{** 11, 22, 33, 44, 55 **}**;

//Using Method Syntax

var ResultMS = IntArray.Any**()**;

//Using Query Syntax

var ResultQS = **(from** num in IntArray

**select** num**)**.Any**()**;

Console.WriteLine**(**"Is there any element in the collection : " + ResultMS**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Using First Overloaded Version of Any Operator in C#

##### ****Example2: Using second Overloaded version of C# Linq Any Operator****

The following program returns false as there is no element that is less than 10.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** IntArray = **{** 11, 22, 33, 44, 55 **}**;

var Result = IntArray.All**(**x =**>** x **>** 10**)**;

Console.WriteLine**(**"Is All Numbers are greater than 10 : " + Result**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Using second Overloaded version of C# Linq Any Operator

##### ****Example3:****

The following example returns true as some of the names are greater than 5 characters.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string**[]** stringArray = **{** "James", "Sachin", "Sourav", "Pam", "Sara" **}**;

//Method Syntax

var ResultMS = stringArray.Any**(**name =**>** name.Length **>** 5**)**;

//Query Syntax

var ResultQS = **(from** name in stringArray

**select** name**)**.Any**(**name =**>** name.Length **>** 5**)**;

Console.WriteLine**(**"Is Any name with length greater than 5 Characters : " + ResultMS**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Linq Any in C#

##### ****Working with Complex Type:****

We are going to work with the following Student and Subject class. So, create a class file with the name **Student.cs** and then copy and paste the following code.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** TotalMarks **{** **get**; **set**; **}**

**public** List**<**Subject**>** Subjects **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetAllStudnets**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**{**ID= 101,Name = "Preety", TotalMarks = 265,

Subjects = new List**<**Subject**>()**

**{**

new Subject**(){**SubjectName = "Math", Marks = 80**}**,

new Subject**(){**SubjectName = "Science", Marks = 90**}**,

new Subject**(){**SubjectName = "English", Marks = 95**}**

**}}**,

new Student**{**ID= 102,Name = "Priyanka", TotalMarks = 278,

Subjects = new List**<**Subject**>()**

**{**

new Subject**(){**SubjectName = "Math", Marks = 90**}**,

new Subject**(){**SubjectName = "Science", Marks = 95**}**,

new Subject**(){**SubjectName = "English", Marks = 93**}**

**}}**,

new Student**{**ID= 103,Name = "James", TotalMarks = 240,

Subjects = new List**<**Subject**>()**

**{**

new Subject**(){**SubjectName = "Math", Marks = 70**}**,

new Subject**(){**SubjectName = "Science", Marks = 80**}**,

new Subject**(){**SubjectName = "English", Marks = 90**}**

**}}**,

new Student**{**ID= 104,Name = "Hina", TotalMarks = 275,

Subjects = new List**<**Subject**>()**

**{**

new Subject**(){**SubjectName = "Math", Marks = 90**}**,

new Subject**(){**SubjectName = "Science", Marks = 90**}**,

new Subject**(){**SubjectName = "English", Marks = 95**}**

**}}**,

new Student**{**ID= 105,Name = "Anurag", TotalMarks = 255,

Subjects = new List**<**Subject**>()**

**{**

new Subject**(){**SubjectName = "Math", Marks = 80**}**,

new Subject**(){**SubjectName = "Science", Marks = 90**}**,

new Subject**(){**SubjectName = "English", Marks = 85**}**

**}**

**}**,

**}**;

**return** listStudents;

**}**

**}**

**public** **class** Subject

**{**

**public** string SubjectName **{** **get**; **set**; **}**

**public** **int** Marks **{** **get**; **set**; **}**

**}**

**}**

##### ****Example4:****

Check whether any students having total marks greater than 250. As you can see excepts James all the students having a total mark greater than 250. So here it will give you the output as true.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

**bool** MSResult = Student.GetAllStudnets**()**.Any**(**std =**>** std.TotalMarks **>** 250**)**;

//Using Query Syntax

**bool** QSResult = **(from** std in Student.GetAllStudnets**()**

**select** std**)**.Any**(**std =**>** std.TotalMarks **>** 250**)**;

Console.WriteLine**(**MSResult**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:** False

##### ****Example5:****

Now we need to fetch all the student details whose mark on any subject is greater than 80.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSResult = Student.GetAllStudnets**()**

.Where**(**std =**>** std.Subjects.Any**(**x =**>** x.Marks **>** 90**))**.ToList**()**;

//Using Query Syntax

var QSResult = **(from** std in Student.GetAllStudnets**()**

**where** std.Subjects.Any**(**x =**>** x.Marks **>** 90**)**

**select** std**)**.ToList**()**;

**foreach** **(**var item in QSResult**)**

**{**

Console.WriteLine**(**item.Name + " " + item.TotalMarks**)**;

**}**

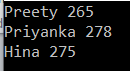
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



# Linq Contains in C#

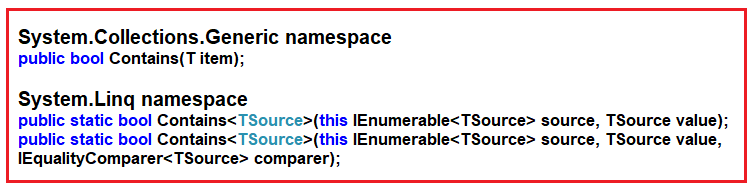
## ****Linq Contains in C# with Examples****

In this article, I am going to discuss the **Linq Contains in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Linq Any Operator in C#**](https://dotnettutorials.net/lesson/linq-any-operator/) with some different types of examples. As part of this article, we are going to discuss the following pointers.

1. **What is Linq Contains in C#?**
2. **Multiple examples using both Method and Query syntax.**

##### ****What is Linq Contains in C#?****

The **Linq Contains Method in C#** is used to check whether a sequence or collection (i.e. data source) contains a specified element or not. If the data source contains the specified element, then it returns true else return false. There Contains method in C# is implemented in two different namespaces as shown in the below image.



The Contains method belongs to **System.Collections.Generic** namespace takes one element as an input parameter and if that element present in the data source then it returns true else false.

There are two overloaded versions available for the Contains method that belongs to **System.Linq** namespace and of the method take **IEqualityComparer**.

**Note:** This method works in a different manner when working with complex type objects. For complex type objects, it only checks the reference, not the values. In order to work with values, we need to use **IEqualityComparer**.

##### ****Example1:****

The following example returns true as the data source (i.e. IntArray) contains the element 33.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** IntArray = **{** 11, 22, 33, 44, 55 **}**;

//Using Method Syntax

var IsExistsMS = IntArray.Contains**(**33**)**;

//Using Query Syntax

var IsExistsQS = **(from** num in IntArray

**select** num**)**.Contains**(**33**)**;

Console.WriteLine**(**IsExistsMS**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: True**

##### ****Example2:****

The following example returns False as the sequence or data source does not contain any element with the name Anurag.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** namesList = new List**<**string**>(){** "James", "Sachin", "Sourav", "Pam", "Sara" **}**;

//Using Method Syntax

//This method belongs to System.Collections.Generic namespace

var IsExistsMS1 = namesList.Contains**(**"Anurag"**)**;

//This method belongs to System.Linq namespace

var IsExistsMS2 = namesList.AsEnumerable**()**.Contains**(**"Anurag"**)**;

//Using Query Syntax

var IsExistsQS = **(from** num in namesList

**select** num**)**.Contains**(**"Anurag"**)**;

Console.WriteLine**(**IsExistsQS**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: False**

##### ****Working with Complex Type:****

We are going to work with the following Student and Subject class. So, create a class file with the name **Student.cs** and then copy and paste the following code.

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** TotalMarks **{** **get**; **set**; **}**

**}**

**}**

##### ****Example3:****

The following example returns True as the object that we pass to the Contains method is exists in the data source.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

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

var student1 = new Student**()** **{** ID = 101, Name = "Priyanka", TotalMarks = 275 **}**;

var student2 = new Student**()** **{** ID = 102, Name = "Preety", TotalMarks = 375 **}**;

students.Add**(**student1**)**;

students.Add**(**student1**)**;

//Using Method Syntax

var IsExistsMS = students.Contains**(**student1**)**;

//Using Query Syntax

var IsExistsQS = **(from** num in students

**select** num**)**.Contains**(**student1**)**;

Console.WriteLine**(**IsExistsMS**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:** True

Note: While working with complex type, the Contains method checks the object reference, not the values of the object. In the above example, the object reference of the object we passed is available in the data source, so it returns true.

##### ****Example4:****

The following example returns false even though the values that we passed is available in the data source. This is because the Linq Contains Method in C# does not check the values rather it checks the object reference and in this case, the object references are different.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** students = new List**<**Student**>()**

**{**

new Student**(){**ID = 101, Name = "Priyanka", TotalMarks = 275 **}**,

new Student**(){**ID = 102, Name = "Preety", TotalMarks = 375 **}**

**}**;

//Using Method Syntax

var IsExistsMS = students.Contains**(**new Student**()** **{** ID = 101, Name = "Priyanka", TotalMarks = 275 **})**;

var student1 = new Student**()** **{** ID = 101, Name = "Priyanka", TotalMarks = 275 **}**;

//Using Query Syntax

var IsExistsQS = **(from** num in students

**select** num**)**.Contains**(**student1**)**;

Console.WriteLine**(**IsExistsMS**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: False**

If you want to check the values rather than the reference then you need to create a class and need to implement the **IEqualityComparere** interface. Then you need to use the overloaded version of the Contains method which takes **IEqualityComparere** as a parameter.

##### ****Creating StudentComparer class:****

Create a class file with the name **StudentComparer** and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** StudentComparer : IEqualityComparer**<**Student**>**

**{**

**public** **bool** Equals**(**Student x, Student y**)**

**{**

//If both object refernces are equal then return true

**if(object**.ReferenceEquals**(**x, y**))**

**{**

**return** **true**;

**}**

//If one of the object refernce is null then return false

**if** **(**x **is** **null** || y **is** **null)**

**{**

**return** **false**;

**}**

**return** x.ID == y.ID && x.Name == y.Name && x.TotalMarks == y.TotalMarks;

**}**

**public** **int** GetHashCode**(**Student obj**)**

**{**

//If obj is null then return 0

**if(**obj **is** **null)**

**{**

**return** 0;

**}**

**int** IDHashCode = obj.ID.GetHashCode**()**;

**int** NameHashCode = obj.Name == **null** ? 0 : obj.Name.GetHashCode**()**;

**int** TotalMarksHashCode = obj.TotalMarks.GetHashCode**()**;

**return** IDHashCode ^ NameHashCode ^ TotalMarksHashCode;

**}**

**}**

**}**

##### ****Example5:****

Now pass the **StudentComparer** instance to the contains method as shown below.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** students = new List**<**Student**>()**

**{**

new Student**(){**ID = 101, Name = "Priyanka", TotalMarks = 275 **}**,

new Student**(){**ID = 102, Name = "Preety", TotalMarks = 375 **}**

**}**;

//Createing Student Comparer Instance

StudentComparer studentComparer = new StudentComparer**()**;

//Using Method Syntax

var IsExistsMS = students.Contains**(**new Student**()** **{** ID = 101, Name = "Priyanka", TotalMarks = 275 **}**, studentComparer**)**;

var student1 = new Student**()** **{** ID = 101, Name = "Priyanka", TotalMarks = 275 **}**;

//Using Query Syntax

var IsExistsQS = **(from** num in students

**select** num**)**.Contains**(**student1, studentComparer**)**;

Console.WriteLine**(**IsExistsMS**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: True**

# Linq GroupBy in C#

## ****Linq GroupBy in C# with Examples****

In this article, I am going to discuss the **Linq GroupBy in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**Linq Contains Method in C#**](https://dotnettutorials.net/lesson/linq-contains-method/) with examples. As part of this article, we will discuss the following pointers which are related to C# groupby.

1. **What is Linq GroupBy in C#?**
2. **How to use the Linq GroupBy Method using both Query and Method Syntax?**
3. **How to use the GroupBy Method along with OrderBy Method in C# using both Method and Query Syntax?**

##### ****What is Linq GroupBy Method in C#?****

The **Linq** **GroupBy in C#** belongs to the Grouping Operators category and exactly does the same thing as the Group By clause does in SQL Query. This method takes a flat sequence of elements and then organizes the elements into groups (i.e. **IGrouping<TKey, TSource>**) based on a given key.

If you go the definition of **GroupBy** method then you will see that it return an **IEnumerable<IGrouping<TKey, TSource>>** where **TKey** is nothing but the **Key** value on which the grouping has been formed and **TSource** is the collection of elements that matches the grouping key value. If this is not clear at the moment then don’t worry, let us try to understand this with some examples.

##### ****Helper class used in this demo:****

Let us first create a helper class file with the name Student.cs and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *GroupByDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string Barnch **{** **get**; **set**; **}**

**public** **int** Age **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetStudents**()**

**{**

**return** new List**<**Student**>()**

**{**

new Student **{** ID = 1001, Name = "Preety", Gender = "Female",

Barnch = "CSE", Age = 20 **}**,

new Student **{** ID = 1002, Name = "Snurag", Gender = "Male",

Barnch = "ETC", Age = 21 **}**,

new Student **{** ID = 1003, Name = "Pranaya", Gender = "Male",

Barnch = "CSE", Age = 21 **}**,

new Student **{** ID = 1004, Name = "Anurag", Gender = "Male",

Barnch = "CSE", Age = 20 **}**,

new Student **{** ID = 1005, Name = "Hina", Gender = "Female",

Barnch = "ETC", Age = 20 **}**,

new Student **{** ID = 1006, Name = "Priyanka", Gender = "Female",

Barnch = "CSE", Age = 21 **}**,

new Student **{** ID = 1007, Name = "santosh", Gender = "Male",

Barnch = "CSE", Age = 22 **}**,

new Student **{** ID = 1008, Name = "Tina", Gender = "Female",

Barnch = "CSE", Age = 20 **}**,

new Student **{** ID = 1009, Name = "Celina", Gender = "Female",

Barnch = "ETC", Age = 22 **}**,

new Student **{** ID = 1010, Name = "Sambit", Gender = "Male",

Barnch = "ETC", Age = 21 **}**

**}**;

**}**

**}**

**}**

##### ****Example:****

The following example organizes the students into groups based on their branch (i.e. branch will act as the key). That means students with the same branch will be stored in the same group where each group having a key and an inner collection. Here, the key will be branch and the collection will be the student belongs to that particular branch.

**Note:** Please go through the comment line which is self-explained.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *GroupByDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var GroupByMS = Student.GetStudents**()**.GroupBy**(**s =**>** s.Barnch**)**;

//Using Query Syntax

IEnumerable**<**IGrouping**<**string, Student**>>** GroupByQS = **(from** std in Student.GetStudents**()**

**group** std **by** std.Barnch**)**;

//It will iterate through each groups

**foreach(**var **group** in GroupByMS**)**

**{**

Console.WriteLine**(group**.Key +" : " + **group**.Count**())**;

//Iterate through each student of a group

**foreach(**var student in **group)**

**{**

Console.WriteLine**(**" Name :" + student.Name + ", Age: " + student.Age + ", Gender :" + student.Gender**)**;

**}**

**}**

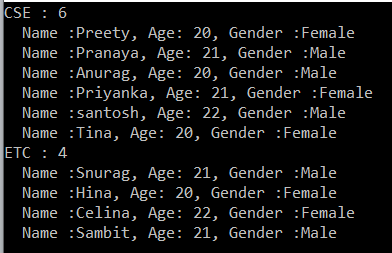
Console.Read**()**;

**}**

**}**

**}**

**Output:**



**Note:** Each group has a key and you can access the key-value by using the key property. Along the same line, you can use the count property to check how many elements are there in that group.

##### ****Example:****

In the following example, we get the employees by Gender. But here we first sort the data by Gender in descending order and then sort the student by their name in ascending order.

**using** *System;*

**using** *System.Linq;*

**namespace** *GroupByDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var GroupByMS = Student.GetStudents**()**.GroupBy**(**s =**>** s.Gender**)**

//First sorting the data based on key in Descending Order

.OrderByDescending**(**c =**>** c.Key**)**

.Select**(**std =**>** new

**{**

Key = std.Key,

//Sorting the data based on name in descending order

Students = std.OrderBy**(**x =**>** x.Name**)**

**})**;

//Using Query Syntax

var GroupByQS = **from** std in Student.GetStudents**()**

**group** std **by** std.Gender **into** stdGroup

**orderby** stdGroup.Key **descending**

**select** new

**{**

Key = stdGroup.Key,

Students = stdGroup.OrderBy**(**x =**>** x.Name**)**

**}**;

//It will iterate through each groups

**foreach** **(**var **group** in GroupByQS**)**

**{**

Console.WriteLine**(group**.Key +" : " + **group**.Students.Count**())**;

//Iterate through each student of a group

**foreach(**var student in **group**.Students**)**

**{**

Console.WriteLine**(**" Name :" + student.Name + ", Age: " + student.Age + ", Branch :" + student.Barnch**)**;

**}**

**}**

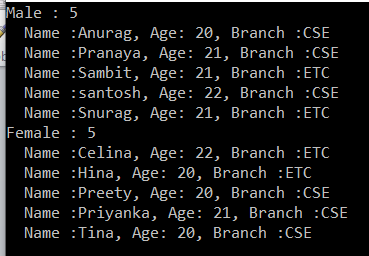
Console.Read**()**;

**}**

**}**

**}**

**Output:**



**Note:** Here the result is projected to an anonymous type.

# GroupBy By Multiple Keys in Linq

In this article, I am going to discuss the **GroupBy By Multiple Keys in Linq** using C# with some examples. Please read our previous article before proceeding to this article where we discussed the [**Linq GroupBy Method in C#**](https://dotnettutorials.net/lesson/groupby-in-linq/) with examples in C#. As part of this article, we will discuss the following pointers.

1. **Why do we need to group the data based on multiple keys?**
2. **How to use the Linq GroupBy Method with Multiple Keys in C#?**
3. **Examples using both Method and Query Syntax.**

##### ****Why do we need to group the data based on Multiple keys?****

In real-time applications, we need to group the data based on multiple keys. So in this article, we will see how to group the data based on multiple keys. But before that, you just need to remember one thing that when you are using multiple keys in Group By operator then the data returned is an anonymous type.

##### ****Student Helper Class:****

We are going to use the following Student class. Please create a class with the name Student and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *GroupByDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string Barnch **{** **get**; **set**; **}**

**public** **int** Age **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetStudents**()**

**{**

**return** new List**<**Student**>()**

**{**

new Student **{** ID = 1001, Name = "Preety", Gender = "Female",

Barnch = "CSE", Age = 20 **}**,

new Student **{** ID = 1002, Name = "Snurag", Gender = "Male",

Barnch = "ETC", Age = 21 **}**,

new Student **{** ID = 1003, Name = "Pranaya", Gender = "Male",

Barnch = "CSE", Age = 21 **}**,

new Student **{** ID = 1004, Name = "Anurag", Gender = "Male",

Barnch = "CSE", Age = 20 **}**,

new Student **{** ID = 1005, Name = "Hina", Gender = "Female",

Barnch = "ETC", Age = 20 **}**,

new Student **{** ID = 1006, Name = "Priyanka", Gender = "Female",

Barnch = "CSE", Age = 21 **}**,

new Student **{** ID = 1007, Name = "santosh", Gender = "Male",

Barnch = "CSE", Age = 22 **}**,

new Student **{** ID = 1008, Name = "Tina", Gender = "Female",

Barnch = "CSE", Age = 20 **}**,

new Student **{** ID = 1009, Name = "Celina", Gender = "Female",

Barnch = "ETC", Age = 22 **}**,

new Student **{** ID = 1010, Name = "Sambit", Gender = "Male",

Barnch = "ETC", Age = 21 **}**

**}**;

**}**

**}**

**}**

##### ****Example:****

The following example group the students first by **Branch**and then by **Gender**. The student groups first sorted by **Branch in descending order**and then by **Gender in ascending order**. Finally, the students in each group are sorted by their **names in ascending order**.

**using** *System;*

**using** *System.Linq;*

**namespace** *GroupByDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var GroupByMultipleKeysMS = Student.GetStudents**()**

.GroupBy**(**x =**>** new **{** x.Barnch, x.Gender **})**

.OrderByDescending**(**g =**>** g.Key.Barnch**)**.ThenBy**(**g =**>** g.Key.Gender**)**

.Select**(**g =**>** new

**{**

Branch = g.Key.Barnch,

Gender = g.Key.Gender,

Students = g.OrderBy**(**x =**>** x.Name**)**

**})**;

//Using Query Syntax

var GroupByMultipleKeysQS = **from** student in Student.GetStudents**()**

**group** student **by** new

**{**

student.Barnch,

student.Gender

**}** **into** stdGroup

**orderby** stdGroup.Key.Barnch **descending**,

stdGroup.Key.Gender **ascending**

**select** new

**{**

Branch = stdGroup.Key.Barnch,

Gender = stdGroup.Key.Gender,

Students = stdGroup.OrderBy**(**x =**>** x.Name**)**

**}**;

//It will iterate through each group

**foreach** **(**var **group** in GroupByMultipleKeysQS**)**

**{**

Console.WriteLine**(**$"Barnch : {group.Branch} Gender: {group.Gender} No of Students = {group.Students.Count()}"**)**;

//It will iterate through each item of a group

**foreach** **(**var student in **group**.Students**)**

**{**

Console.WriteLine**(**$" ID: {student.ID}, Name: {student.Name}, Age: {student.Age} "**)**;

**}**

Console.WriteLine**()**;

**}**

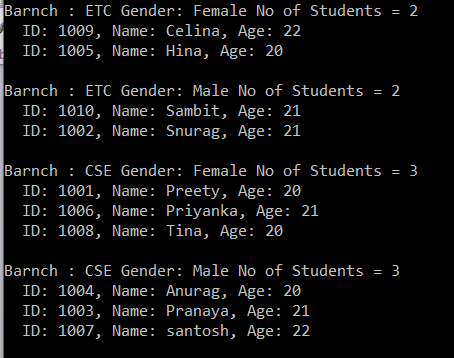
Console.Read**()**;

**}**

**}**

**}**

**Output:**



# Linq ToLookup Method in C#

## ****Linq ToLookup Method in C# with example****

In this article, I am going to discuss the **Linq ToLookup Method in C#** with some examples. Please read our previous article before proceeding to this article where we discussed the [**Linq GroupBy Operator with Multiple Keys**](https://dotnettutorials.net/lesson/groupby-multiple-keys-in-linq/) with some examples. As part of this article, we are going to discuss the following pointers.

1. **What is Linq ToLookup Method?**
2. **Examples of using the ToLookup method in C# using bothe Method and Query Syntax.**
3. **How to use the ToLookup method with the OrderBy method?**
4. **How to use the Linq ToLookup method in C# with Multiple Keys in C#?**
5. **What is the difference between ToLookup and GroupBy method in Linq?**

##### ****What is Linq ToLookup Operator?****

The **Linq ToLookup Method in C#** exactly does the same thing as the **GroupBy Operator does in Linq**. The only difference between these two methods is the GroupBy method uses deferred execution whereas the execution of the ToLookup method is immediate. Please read the following article to understand what is Deferred and Immediate Execution in Linq queries.

[**Deferred Execution VS Immediate Execution in C#.**](https://dotnettutorials.net/lesson/deferred-execution-vs-immediate-execution-in-linq/)

##### ****Student Helper class:****

We are going to use the following Student Helper class in this demo. Please create a class file and then just copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *GroupByDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Gender **{** **get**; **set**; **}**

**public** string Barnch **{** **get**; **set**; **}**

**public** **int** Age **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetStudents**()**

**{**

**return** new List**<**Student**>()**

**{**

new Student **{** ID = 1001, Name = "Preety", Gender = "Female",

Barnch = "CSE", Age = 20 **}**,

new Student **{** ID = 1002, Name = "Snurag", Gender = "Male",

Barnch = "ETC", Age = 21 **}**,

new Student **{** ID = 1003, Name = "Pranaya", Gender = "Male",

Barnch = "CSE", Age = 21 **}**,

new Student **{** ID = 1004, Name = "Anurag", Gender = "Male",

Barnch = "CSE", Age = 20 **}**,

new Student **{** ID = 1005, Name = "Hina", Gender = "Female",

Barnch = "ETC", Age = 20 **}**,

new Student **{** ID = 1006, Name = "Priyanka", Gender = "Female",

Barnch = "CSE", Age = 21 **}**,

new Student **{** ID = 1007, Name = "santosh", Gender = "Male",

Barnch = "CSE", Age = 22 **}**,

new Student **{** ID = 1008, Name = "Tina", Gender = "Female",

Barnch = "CSE", Age = 20 **}**,

new Student **{** ID = 1009, Name = "Celina", Gender = "Female",

Barnch = "ETC", Age = 22 **}**,

new Student **{** ID = 1010, Name = "Sambit", Gender = "Male",

Barnch = "ETC", Age = 21 **}**

**}**;

**}**

**}**

**}**

##### ****Example:****

The following example uses the ToLookup Method to organize the students into groups based on Branch as the key. Here, the key will be branch and the collection will be the student belongs to that particular branch.

**using** *System;*

**using** *System.Linq;*

**namespace** *GroupByDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var GroupByMS = Student.GetStudents**()**.ToLookup**(**s =**>** s.Barnch**)**;

//Using Query Syntax

var GroupByQS = **(from** std in Student.GetStudents**()**

**select** std**)**.ToLookup**(**x =**>** x.Barnch**)**;

//It will iterate through each group

**foreach** **(**var **group** in GroupByMS**)**

**{**

Console.WriteLine**(group**.Key + " : " + **group**.Count**())**;

//Iterate through each student of a group

**foreach** **(**var student in **group)**

**{**

Console.WriteLine**(**" Name :" + student.Name + ", Age: " + student.Age + ", Gender :" + student.Gender**)**;

**}**

**}**

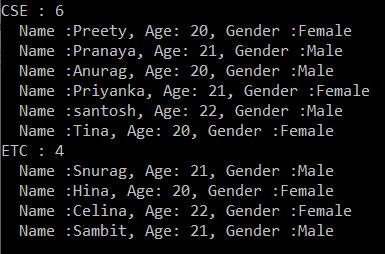
Console.Read**()**;

**}**

**}**

**}**

**Output:**



##### ****Example:****

In the following example, we use the Linq ToLookup operator to get the employees by Gender. But here we sort the data first by Gender in descending order and then sort the data by the name in ascending order.

**using** *System;*

**using** *System.Linq;*

**namespace** *GroupByDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

var GroupByMS = Student.GetStudents**()**.ToLookup**(**s =**>** s.Gender**)**

.OrderByDescending**(**c =**>** c.Key**)**

.Select**(**std =**>** new

**{**

Key = std.Key,

Students = std.OrderBy**(**x =**>** x.Name**)**

**})**;

**foreach** **(**var **group** in GroupByMS**)**

**{**

Console.WriteLine**(group**.Key + " : " + **group**.Students.Count**())**;

**foreach** **(**var student in **group**.Students**)**

**{**

Console.WriteLine**(**" Name :" + student.Name + ", Age: " + student.Age + ", Branch :" + student.Barnch**)**;

**}**

**}**

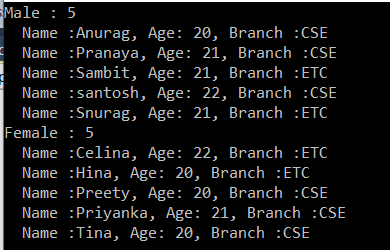
Console.Read**()**;

**}**

**}**

**}**

**Output:**



##### ****Using ToLookup with Multiple Keys in C#:****

The following example uses ToLookup to group the students first by Branch and then by Gender. The student’s groups first sorted by Branch in descending order and then by Gender in ascending order. Finally, we sort the data in each group by the name in ascending order.

**using** *System;*

**using** *System.Linq;*

**namespace** *GroupByDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

var ToLookupwithMultipleKeys = Student.GetStudents**()**

.ToLookup**(**x =**>** new **{** x.Barnch, x.Gender **})**

.OrderByDescending**(**g =**>** g.Key.Barnch**)**.ThenBy**(**g =**>** g.Key.Gender**)**

.Select**(**g =**>** new

**{**

Branch = g.Key.Barnch,

Gender = g.Key.Gender,

Students = g.OrderBy**(**x =**>** x.Name**)**

**})**;

**foreach** **(**var **group** in ToLookupwithMultipleKeys**)**

**{**

Console.WriteLine**(**$"Barnch : {group.Branch} Gender: {group.Gender} No of Students = {group.Students.Count()}"**)**;

**foreach** **(**var student in **group**.Students**)**

**{**

Console.WriteLine**(**$" ID: {student.ID}, Name: {student.Name}, Age: {student.Age} "**)**;

**}**

Console.WriteLine**()**;

**}**

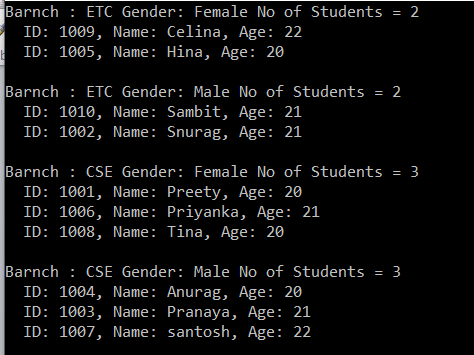
Console.Read**()**;

**}**

**}**

**}**

**Output:**



# Left Join in Linq

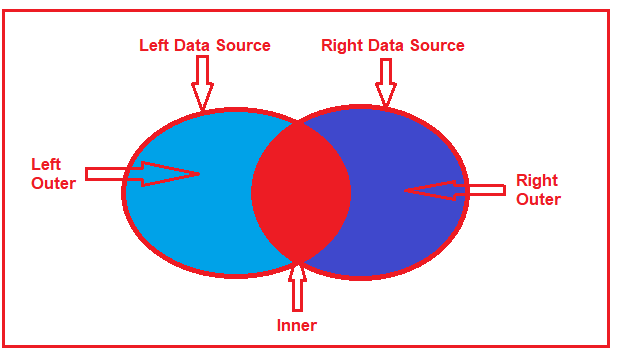
## ****Left Join in Linq using Method and Query Syntax****

In this article, I am going to discuss **Left Join in Linq** with some examples using both Method and Query Syntax. The Linq Left Join in C# is also called as Left Outer Join in Linq**.** Please read the following three articles before proceeding to this article as they both are required to understand the Left Outer Join.

1. [**Inner Join in Linq**](https://dotnettutorials.net/lesson/inner-join-in-linq/)
2. [**Group Join in Linq**](https://dotnettutorials.net/lesson/linq-group-join/)
3. [**Select Many In Linq**](https://dotnettutorials.net/lesson/selectmany-in-linq/)

##### ****What is Left Join in Linq?****

The left join or left outer join is a join in which each data from the first data source is going to be returned irrespective of whether it has any correlated data present in the second data source or not. Please have a look at the following diagram which shows the graphical representation of Left Outer Join.



So, in simple words, we can say that the Left Outer Join is going to return all the matching data from both the data sources as well as all the non-matching data from the left data source. In such cases, for the non-matching data, it will take null values for the second data source.

In order to implement the Linq Left Join in C#, it’s mandatory to use the “**INTO**” keyword along with the “**DefaultIfEmpty()”** method.

##### ****Model Classes and Data Sources:****

We are going to use the following Employee and Address models in this demo. Please create a class file and then copy and paste the following code.

**using** *System.Collections.Generic;*

**namespace** *LINQJoin*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** AddressId **{** **get**; **set**; **}**

**public** **static** List**<**Employee**>** GetAllEmployees**()**

**{**

**return** new List**<**Employee**>()**

**{**

new Employee **{** ID = 1, Name = "Preety", AddressId = 1**}**,

new Employee **{** ID = 2, Name = "Priyanka", AddressId =2**}**,

new Employee **{** ID = 3, Name = "Anurag", AddressId = 0**}**,

new Employee **{** ID = 4, Name = "Pranaya", AddressId = 0**}**,

new Employee **{** ID = 5, Name = "Hina", AddressId = 5**}**,

new Employee **{** ID = 6, Name = "Sambit", AddressId = 6**}**

**}**;

**}**

**}**

**public** **class** Address

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string AddressLine **{** **get**; **set**; **}**

**public** **static** List**<**Address**>** GetAddress**()**

**{**

**return** new List**<**Address**>()**

**{**

new Address **{** ID = 1, AddressLine = "AddressLine1"**}**,

new Address **{** ID = 2, AddressLine = "AddressLine2"**}**,

new Address **{** ID = 5, AddressLine = "AddressLine5"**}**,

new Address **{** ID = 6, AddressLine = "AddressLine6"**}**,

**}**;

**}**

**}**

**}**

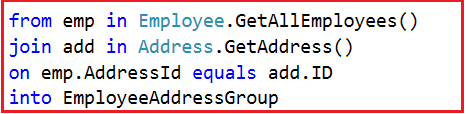
As you can see, here we also created two methods which will be going to return the respective Employees and Addresses which are going to be our data source. Here we hard-coded the data sources but in real-time you will get the data from a database. If you further notice we have two employees with address id 0 that means these two employees do not have a matching address in the address data source.

##### ****Left Outer Join using Query Syntax:****

In order to perform the left outer join using query syntax, you need to call the **DefaultIfEmpty()** method on the results of a group join. Let’s see the step by step procedure to implement the left outer join in Linq.

##### ****Step1:****

The first step to implement a left outer join is to perform an inner join by using a group join. In the below example, the list of Employees is inner-joined to the list of Addresses based on the Address Id of Employee object that matches the ID of the Address object. The following code does the same.



##### ****Step2:****

In the second step, we need to include each element of the first (i.e. left) data source in the result set irrespective of whether that element has no matches in the second (i.e. right) data source. In order to do this, we need to call the **DefaultIfEmpty()** method on each sequence of matching elements from the group join.

In our example, we need to call the **DefaultIfEmpty()** method on each sequence of matching Address objects. The **DefaultIfEmpty()** method returns a collection that contains a single, default value if the sequence of matching Address object is empty for any Employee object which will ensure that each Employee object is represented in the result collection. The following code exactly does the same thing.

DefaultIfEmplty in Outer Join

**Note:** The default value for a reference type is null. So, you need to check for the null reference before accessing each element of Address collection.

**The complete code is given below.**

**using** *System.Linq;*

**using** *System;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

var QSOuterJoin = **from** emp in Employee.GetAllEmployees**()**

**join** **add** in Address.GetAddress**()**

**on** emp.AddressId **equals** **add**.ID

**into** EmployeeAddressGroup

**from** address in EmployeeAddressGroup.DefaultIfEmpty**()**

**select** new **{**emp, address **}**;

**foreach** **(**var item in QSOuterJoin**)**

**{**

Console.WriteLine**(**$"Name : {item.emp.Name}, Address : {item.address?.AddressLine} "**)**;

**}**

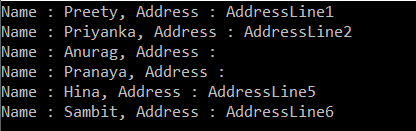
Console.ReadLine**()**;

**}**

**}**

**}**

**Output:**



##### ****Left Outer Join in Linq using Method Syntax:****

In order to implement Left Outer Join in Linq using Method Syntax we need to use the **GroupJoin()** method along with **SelectMany()** and **DefaultIfEmpty()** methods. So, let us rewrite the previous example using Method Syntax as shown below.

**using** *System.Linq;*

**using** *System;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

var MSOuterJOIN = Employee.GetAllEmployees**()**

.GroupJoin**(**

Address.GetAddress**()**,

emp =**>** emp.AddressId,

**add** =**>** **add**.ID,

**(**emp, **add)** =**>** new **{** emp, **add** **}**

**)**

.SelectMany**(**

x =**>** x.add.DefaultIfEmpty**()**,

**(**employee, address**)** =**>** new**{** employee, address **}**

**)**;

**foreach** **(**var item in MSOuterJOIN**)**

**{**

Console.WriteLine**(**$"Name : {item.employee.emp.Name}, Address : {item.address?.AddressLine} "**)**;

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

It will give you the same output as the previous example. I feel it always better to use Query Syntax over Method Syntax to perform left outer join in Linq as it is simple and easy to understand.

##### ****Anonymous type with user-defined properties in the ResultSet:****

Let us see how to return an anonymous type with user-defined properties using Linq left join in C#.

**using** *System.Linq;*

**using** *System;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Using Method Syntax

var MSOuterJOIN = Employee.GetAllEmployees**()**

.GroupJoin**(**

Address.GetAddress**()**,

emp =**>** emp.AddressId,

**add** =**>** **add**.ID,

**(**emp, **add)** =**>** new **{** emp, **add** **}**

**)**

.SelectMany**(**

x =**>** x.add.DefaultIfEmpty**()**,

**(**employee, address**)** =**>** new

**{**

EmployeeName = employee.emp.Name,

AddressLine = address == **null** ? "NA" : address.AddressLine

**}**

**)**;

//Using Query Syntax

var QSOuterJoin = **from** emp in Employee.GetAllEmployees**()**

**join** **add** in Address.GetAddress**()**

**on** emp.AddressId **equals** **add**.ID

**into** EmployeeAddressGroup

**from** address in EmployeeAddressGroup.DefaultIfEmpty**()**

**select** new

**{**

EmployeeName = emp.Name,

AddressLine = address == **null** ? "NA" : address.AddressLine

**}**;

**foreach** **(**var item in MSOuterJOIN**)**

**{**

Console.WriteLine**(**$"Name : {item.EmployeeName}, Address : {item.AddressLine} "**)**;

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

**Note:** If you want to perform Right outer join then simply exchange the data source.

# Linq Cross Join

## ****Linq Cross Join using Method and Query Syntax****

In this article, I am going to discuss **Linq Cross Join** using both method and query syntax examples. Please read our previous article before proceeding to this article where we discussed [**Left Outer Join**](https://dotnettutorials.net/lesson/left-outer-join-in-linq/) in Linq.

##### ****What is Linq Cross Join?****

When combining two data sources (or you can two collections) using Linq Cross Join, then each element in the first data source (i.e. first collection) will be mapped with each and every element in the second data source (i.e. second collection). So, in simple words, we can say that the Cross Join produces the Cartesian Products of the collections or data sources involved in the join.

In Cross Join we don’t require the common key or property as the “on” keyword which is used to specify the Join Key is not required. And moreover, there is no filtering of data. So, the total number of elements in the resultant sequence will be the product of the two data sources involved in the join. If the first data source contains 5 elements and the second data source contains 3 elements then the resultant sequence will contain (5\*3) 15 elements.

##### ****Model classes and Data Sources:****

We are going to use the following Student and Subject model classes in this demo. Please create a class file and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *LINQJoin*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetAllStudnets**()**

**{**

**return** new List**<**Student**>()**

**{**

new Student **{** ID = 1, Name = "Preety"**}**,

new Student **{** ID = 2, Name = "Priyanka"**}**,

new Student **{** ID = 3, Name = "Anurag"**}**,

new Student **{** ID = 4, Name = "Pranaya"**}**,

new Student **{** ID = 5, Name = "Hina"**}**

**}**;

**}**

**}**

**public** **class** Subject

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string SubjectName **{** **get**; **set**; **}**

**public** **static** List**<**Subject**>** GetAllSubjects**()**

**{**

**return** new List**<**Subject**>()**

**{**

new Subject **{** ID = 1, SubjectName = "ASP.NET"**}**,

new Subject **{** ID = 2, SubjectName = "SQL Server" **}**,

new Subject **{** ID = 5, SubjectName = "Linq"**}**

**}**;

**}**

**}**

**}**

As you can see we created two methods to return the respective data sources.

##### ****Example1: Cross Join Using Query Syntax****

Cross Join Students with Subjects using Query Syntax.

**using** *System.Linq;*

**using** *System;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

var CrossJoinResult = **from** employee in Student.GetAllStudnets**()**

**from** subject in Subject.GetAllSubjects**()**

**select** new **{**

Name = employee.Name,

SubjectName = subject.SubjectName

**}**;

**foreach** **(**var item in CrossJoinResult**)**

**{**

Console.WriteLine**(**$"Name : {item.Name}, Subject: {item.SubjectName}"**)**;

**}**

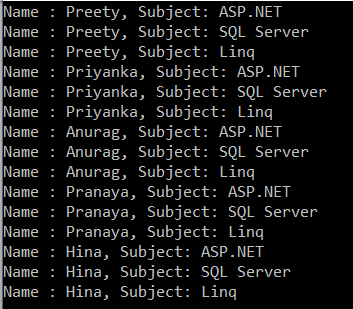
Console.ReadLine**()**;

**}**

**}**

**}**

**Output:**



We have 5 students in the student’s collection and 3 subjects in the subject’s collection. In the result set, we have 15 elements, i.e. the Cartesian product of the elements involved in the joins.

##### ****Example2: Cross Join using Method Syntax.****

In order to implement the **Cross Join**using method syntax, we need to use either the **SelectMany()** method or the **Join()** method as shown in the below example.

**using** *System.Linq;*

**using** *System;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Cross Join using SelectMany Method

var CrossJoinResult = Student.GetAllStudnets**()**

.SelectMany**(**sub =**>** Subject.GetAllSubjects**()**,

**(**std, sub**)** =**>** new

**{**

Name = std.Name,

SubjectName = sub.SubjectName

**})**;

//Cross Join using Join Method

var CrossJoinResult2 = Student.GetAllStudnets**()**

.Join**(**Subject.GetAllSubjects**()**,

std =**>** **true**,

sub =**>** **true**,

**(**std, sub**)** =**>** new

**{**

Name = std.Name,

SubjectName = sub.SubjectName

**}**

**)**;

**foreach** **(**var item in CrossJoinResult2**)**

**{**

Console.WriteLine**(**$"Name : {item.Name}, Subject: {item.SubjectName}"**)**;

**}**

Console.ReadLine**()**;

**}**

**}**

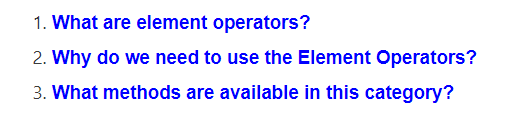
**}**

It will give you the same result as the previous example.

# Element Operators in LINQ

## ****Element Operators in LINQ****

In this article, I am going to give you a brief introduction to the **Element Operators in LINQ**. Please read our previous article where we discussed the [**Linq Cross Join**](https://dotnettutorials.net/lesson/linq-cross-join/) with examples. As part of this article, we are going to discuss the following pointers in detail.



##### ****What are Element Operators in Linq?****

The Element Operators in Linq are used to return a single element from a data source using the element index or based on a predicate i.e. a condition. These Element Operators can be used with a single data source or on a query of multiple data sources.

##### ****Why do we need to use the Element Operators in Linq?****

If you want to perform the following operations, then you need to use the Element Operators.

1. Select the First record from a data source.
2. Fetch a specific record from the data source.
3. Select the last record from a data source.

##### ****What methods are available in the Element Operators category?****

The following methods are provided by Linq to perform element operations.

1. [**ElementAt and ElementAtOrDefault**](https://dotnettutorials.net/lesson/elementat-and-elementatordefault-in-linq/)
2. [**First and FirstOrDefault**](https://dotnettutorials.net/lesson/first-and-firstordefault-methods-in-linq/)
3. [**Last and LastOrDefault**](https://dotnettutorials.net/lesson/last-and-lastordefault-methods-in-linq/)
4. [**Single and SingleOrDefault**](https://dotnettutorials.net/lesson/single-and-singleordefault-methods-in-linq/)
5. [**DefaultIfEmpty**](https://dotnettutorials.net/lesson/defaultifempty-method-in-linq/)

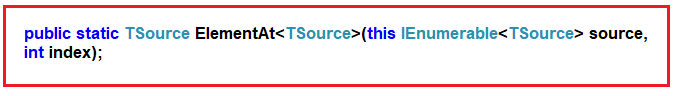
# ElementAt and ElementAtOrDefault in Linq

## ****ElementAt and ElementAtOrDefault in Linq****

In this article, I am going to discuss the **ElementAt and ElementAtOrDefault operator in Linq** with some examples. Please read our previous article before proceeding to this article, where we discussed the basics of [**Element Operators**](https://dotnettutorials.net/lesson/element-operators-in-linq/) in Linq.

##### ****ElementAt Operator in Linq:****

The ElementAt operator is used to return an element from a specific index. If the data source is empty or if the provided index value is out of range, then we will get ArgumentOutOfRangeException. Let us see the signatures of this method as shown below.



As you can see, this method takes one parameter i.e. the index position. Then it will return the element present in that index position of the data source. There is no overloaded version available for this method.

##### ****Example1: Return the element present in index position 1.****

Please have a look at the following program. Here we have created one data source which contains integer numbers. Then we fetch the element present in index position 1 by using the ElementAt method and to that method, we pass the value 1.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers =new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.ElementAt**(**1**)**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

Run the application and it should print 2 in the output window as in index position 1 the value 2 is there.

##### ****Example2: Index Out of Range****

As our data source contains 10 elements so the index is going to start from 0 to 9. Now let see what happen I try to fetch the element from index position 10 or try to pass a negative index value as shown below.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers =new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

//int MethodSyntax = numbers.ElementAt(-1);

**int** MethodSyntax = numbers.ElementAt**(**10**)**;

Console.WriteLine**(**MethodSyntax**)**;

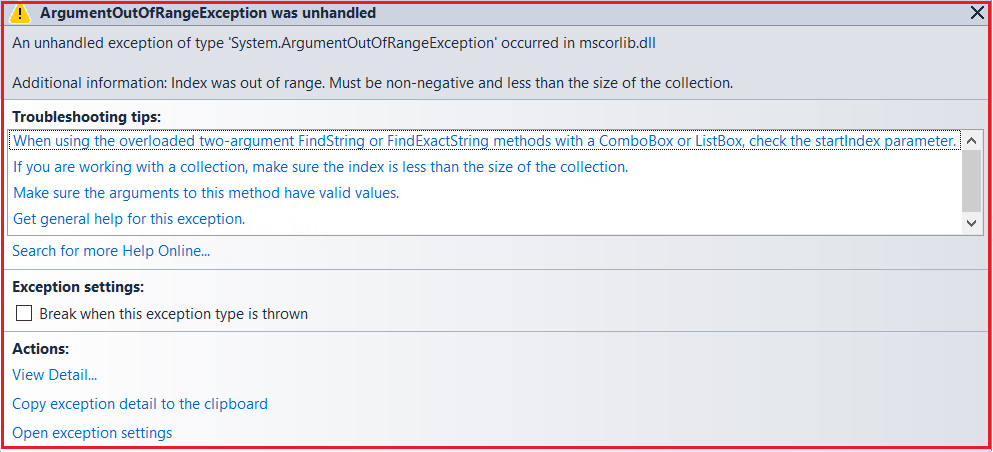
Console.ReadLine**()**;

**}**

**}**

**}**

Now if you run the application, then you will get the following exception.



##### ****Example3: Empty Data source****

Let see what happens if we apply the ElementAt operator on an empty data source as shown in the following example.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers =new List**<int>()** **{** **}**;

**int** MethodSyntax = numbers.ElementAt**(**1**)**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

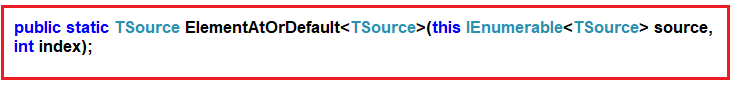
Again you will get the same exception as the previous example.

**Note:** If the data source is empty or if you specify a negative value for the index position or if you specify the index position which is out of range then you will get a runtime exception.

If you don’t want that exception then you need to use the ElementAtOrDefault method.

##### ****ElementAtOrDefault Method in Linq:****

The ElementAtOrDefault method does the same thing as the ElementAt method except that this method does not throw an exception when the data source is empty or when the supplied index value is out of range. In such cases, it will return the default value based on the data type of the element the data source contain. Please have a look at the definition of this method as shown in the below image.



Like the ElementAt method, this method also does not have an overloaded version. Let us understand this method with some examples.

##### ****Example4: Fetch the element from index position 1.****

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers =new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.ElementAtOrDefault**(**1**)**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 2**

##### ****Example5: Fetch the element from index position 10.****

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers =new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.ElementAtOrDefault**(**10**)**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 0**

Here it will print the value 0. This is because the data source contains integers. And the default for integer is 0.

##### ****Example6: Using Query Syntax****

There is no such operator call ElementAt Or ElementAtOrDefault is available to write the query syntax, If you want then you can combine both the method syntax and query syntax to write the code as shown below.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQJoin*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** QuerySyntax1 = **(from** num in numbers

**select** num**)**.ElementAt**(**1**)**;

**int** QuerySyntax2 = **(from** num in numbers

**select** num**)**.ElementAtOrDefault**(**1**)**;

Console.WriteLine**(**QuerySyntax1**)**;

Console.WriteLine**(**QuerySyntax2**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

##### ****What is the difference between the ElementAt and ElementAtOrDefault method?****

Both methods are used to return an element from the specified index. But if the element is not available at the specified index position, then the ElementAt method will throw an exception while the ElementAtOrDefault method will not throw an exception instead it returns a default value based on the data type of the element.

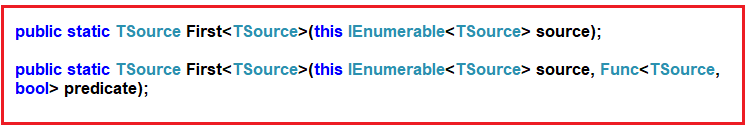
# First and FirstOrDefault Methods in Linq

## ****First and FirstOrDefault Methods in Linq****

In this article, I am going to discuss the **First and FirstOrDefault Methods in Linq** with some examples. Please read our previous article before proceeding to this article, where we discussed the [**ElementAt and ElementAtOrDefault method**](https://dotnettutorials.net/lesson/elementat-and-elementatordefault-in-linq/) in Linq.

##### ****First Method in Linq:****

The Linq First Method is used to return the first element from a data source. If the data source is empty, then this method will throw an exception. There are two overloaded versions available for this method as shown below.



As you can see, the first overloaded version does not take any parameter and it simply returns the first element from the data source. The second overloaded version takes a predicate as a parameter using which we can specify a condition and then it will return the first element which satisfied the specified condition.

##### ****Example1: Return the first element from a data source.****

In the below example, we have created one data source which contains integer numbers from 1 to 10. So, our requirement is to fetch the first element from the data source i.e. in the output we want to display the value 1 as it is the first element in the data source.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.First**()**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

Now, run the application and it should display 1 in the output window as expected.

##### ****Example2: Fetch the first element from the data source which is divisible by 2.****

Here we need to use the second overloaded version of the First method using which we can specify our condition. The following program uses the second overloaded version to return the first element from the data source which is divisible by 2.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.First**(**num =**>** num % 2 == 0**)**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 2**

This is because 2 is the first element in the data source which is divisible by 2.

##### ****Example3: InvalidOperationException****

Whenever the data source is empty or if the specified condition does not return any data, then we will get the **InvalidOperationException** as shown in the below example.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Empty Data Source

List**<int>** numbersEmpty = new List**<int>()** **{** **}**;

**int** MethodSyntax1 = numbersEmpty.First**()**;

//Specified condition doesnot return any element

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax2 = numbers.First**(**num =**>** num **>** 50**)**;

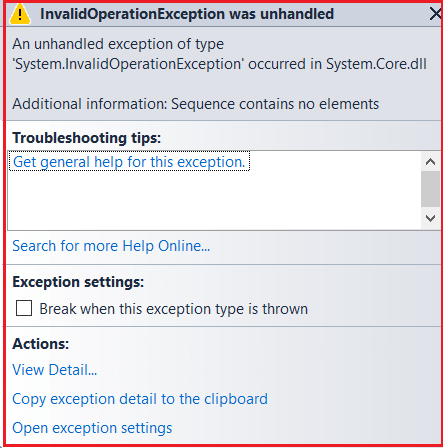
Console.ReadLine**()**;

**}**

**}**

**}**

Now if you run the application, you will get the following exception.

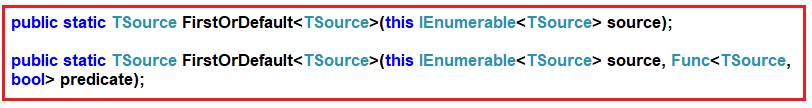


If you don’t want that Invalid Operation Exception, instead you want a default value based on the data type then you need to use the FirstOrDefault method.

Note: For the reference types, the default is NULL and for the value types, the default depends on the actual type expected.

##### ****FirstOrDefault Method in Linq:****

The Linq FirstOrDefault method exactly does the same thing as the First method except that this method does not throw Invalid Operation Exception instead it returns the default value based on the data type of the element. Like the First method, there are also two overloaded versions available for the FirstOrDefault method as shown below.



##### ****Example4: Fetch the element from the data source using FirstOrDefault.****

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.FirstOrDefault**()**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 1**

##### ****Example5: Fetch the element from the data source which is greater than 5.****

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.FirstOrDefault**(**num =**>** num **>** 5**)**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 6**

##### ****Example6:****

In the below example, we have two data sources. The first data source does not contain any item and here we are trying the fetch the first element. The second data source contains 10 elements from the number 1 to 10 and from this data source we are trying to retrieve the first element which is greater than 50.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Empty Data Source

List**<int>** numbersEmpty = new List**<int>()** **{** **}**;

**int** MethodSyntax1 = numbersEmpty.FirstOrDefault**()**;

Console.WriteLine**(**MethodSyntax1**)**;

//Specified condition doesnot return any element

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax2 = numbers.FirstOrDefault**(**num =**>** num **>** 50**)**;

Console.WriteLine**(**MethodSyntax2**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

Here it will print the values as 0 and 0. This is because the data source contains integers. And the default for integer is 0.

##### ****Example7: First and FirstOrDefault method Using Query Syntax****

There is no such operator called First or FirstOrDefault is available to write the query syntax, If you want then you can combine both the method syntax and query syntax to write the code as shown below.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** QuerySyntax1 = **(from** num in numbers

**select** num**)**.First**()**;

**int** QuerySyntax2 = **(from** num in numbers

**select** num**)**.FirstOrDefault**()**;

Console.WriteLine**(**QuerySyntax1**)**;

Console.WriteLine**(**QuerySyntax2**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

##### ****What is the difference between First and FirstOrDefault Methods in Linq?****

Both First and FirstOrDefault methods in Linq are used to return the first element from a data source. But if the data source is empty or if the specified condition does not return any data, then the First method will throw an exception while the FirstOrDefault method will not throw an exception instead it returns a default value based on the data type of the element.

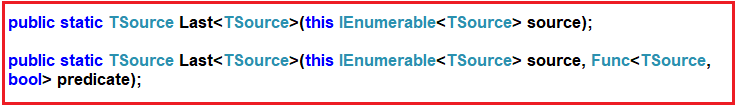
# Last and LastOrDefault Methods in Linq

## ****Last and LastOrDefault Methods in Linq****

In this article, I am going to discuss the **Last and LastOrDefault Methods in Linq** with some examples. Please read our previous article before proceeding to this article, where we discussed the [**First and FirstOrDefault methods in Linq**](https://dotnettutorials.net/lesson/first-and-firstordefault-methods-in-linq/)with some examples.

##### ****Last Method in Linq:****

The Last Method in Linq is used to return the last element from a data source. There are two overloaded versions available for the Last method as shown below.



As you can see, the first overloaded version of the Last method does not take any parameter. It simply returns the last element from the data source and if the data source is empty then it throws an exception.

The second overloaded version of the Last method takes a predicate as a parameter. Using this predicate we can specify a condition and then it returns the last element which satisfied the specified condition. In this case, if no element satisfied the specified condition, then it will throw an exception.

##### ****Example1: Return the last element from a data source.****

In the following example, we have created a data source that contains integer numbers from 1 to 10. So, as per our requirement, we need to fetch the last element from the data source i.e. the value 10 as it is the last element present in the data source.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.Last**()**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

Now, run the application and it should print 10 in the output window as expected.

##### ****Example2: Return the last element from the data source which is less than 5.****

Here we need to specify a condition, so we need to use the second overloaded version of the Last method which takes a predicate as a parameter. The following program uses the second overloaded version of the Last method to return the last element from the data source which is less than 5.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.Last**(**num =**>** num **<** 5**)**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 4**

This is because 4 is the last element in the sequence which is less than 5

##### ****Example3: InvalidOperationException****

If the data source is empty or if no element is satisfied with the given condition, then it will throw the **InvalidOperationException** as shown in the below example.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Empty Data Source

List**<int>** numbersEmpty = new List**<int>()** **{** **}**;

**int** MethodSyntax1 = numbersEmpty.Last**()**;

//Specified condition does not return any element

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax2 = numbers.Last**(**num =**>** num **>** 50**)**;

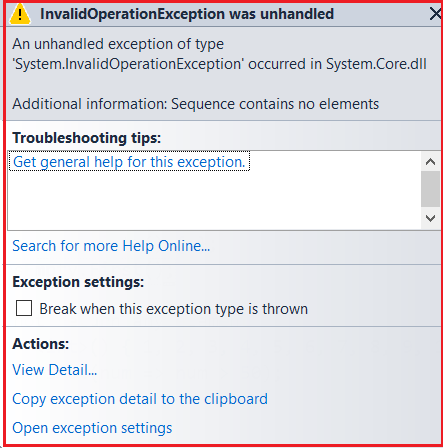
Console.ReadLine**()**;

**}**

**}**

**}**

Now if you run the application, you will get the following exception.

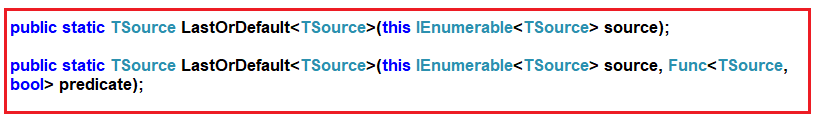


If you don’t want that Invalid Operation Exception, instead if you want a default value to be return based on the data type then you need to use the LastOrDefault method.

**Note:** the default is NULL for the reference types, and for the value types, the default value depends on the actual type expected.

##### ****LastOrDefault Method in Linq:****

The LastOrDefault method exactly does the same thing as the Linq Last method except that the LastOrDefault method does not throw Invalid Operation Exception instead it returns the default value based on the data type of the element. Like the Last method, there are also two overloaded versions available for the LastOrDefault method as shown below.



##### ****Example4: Fetch the last element from the data source using FirstOrDefault.****

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.LastOrDefault**()**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 10**

##### ****Example5: Fetch the last element from the data source which is less than 5.****

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax = numbers.LastOrDefault**(**num =**>** num **<** 5**)**;

Console.WriteLine**(**MethodSyntax**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 4**

##### ****Example6:****

In the below example, we have two data sources. The first data source does not contain any element and from this data source, we are trying the fetch the last element using the LastOrDefault method. The second data source contains 10 elements from the number 1 to 10 and from this data source we are trying to retrieve the last element which is greater than 50.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Empty Data Source

List**<int>** numbersEmpty = new List**<int>()** **{** **}**;

**int** MethodSyntax1 = numbersEmpty.LastOrDefault**()**;

Console.WriteLine**(**MethodSyntax1**)**;

//Specified condition doesnot return any element

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** MethodSyntax2 = numbers.LastOrDefault**(**num =**>** num **>** 50**)**;

Console.WriteLine**(**MethodSyntax2**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

Here it will print the values as 0 and 0. This is because the data source contains integers. And the default for integer is 0.

##### ****Example7: Last and LastOrDefault method Using Query Syntax****

There is no such operator called Last and LastOrDefault is available to write the query syntax, If you want then you can combine both the method syntax and query syntax to write the code as shown in the below example.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

**int** QuerySyntax1 = **(from** num in numbers

**select** num**)**.Last**()**;

**int** QuerySyntax2 = **(from** num in numbers

**select** num**)**.LastOrDefault**()**;

Console.WriteLine**(**QuerySyntax1**)**;

Console.WriteLine**(**QuerySyntax2**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

##### ****What is the difference between Last and LastOrDefault methods in Linq?****

Both Last and LastOrDefault methods in Linq are used to return the last element from a data source. But if the data source is empty or if no element is satisfied with the specified condition, then the Last method will throw an exception while the LastOrDefault method will not throw an exception instead it returns a default value based on the data type of the element.

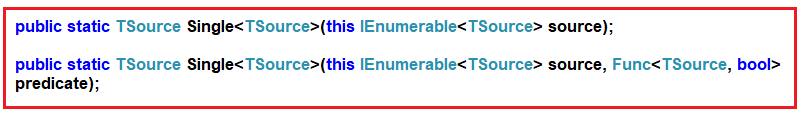
# Single and SingleOrDefault Methods in Linq

## ****Single and SingleOrDefault Methods in Linq****

In this article, I am going to discuss the **Single and SingleOrDefault Methods in Linq** with some examples. Please read our previous article before proceeding to this article, where we discussed the [**Last and LastOrDefault methods in Linq**](https://dotnettutorials.net/lesson/last-and-lastordefault-methods-in-linq/) with some examples.

#### ****Single Method in Linq:****

The Linq Single Method is used to returns a single element from a data source or you can say from a sequence. There are two overloaded versions available for this Linq Single Method, which are shown in the below image.



As you can see, the first overloaded version of the Single method does not take any parameter. This method simply returns the only element from a sequence. If the data source is empty or if the data source contains more than one element, then it throws an exception.

On the other hand, the second overloaded version of the Single method takes one predicate as a parameter and using this predicate you can specify a condition. This method returns the only element from the sequence which satisfied the given condition. In this case, the method will throw an exception when any of the following condition is true.

1. If the data source is empty.
2. When the given condition does not satisfy any element in the sequence.
3. If the given condition satisfies more than one element.

##### ****Example1: Linq Single Method****

In the below example, we have created the data source which contains only one element. Now when we apply the Single method on this data source then it will return the only element which is present in the data source.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Sequence contains one element

List**<int>** numbers = new List**<int>()** **{** 10 **}**;

**int** number = numbers.Single**()**;

Console.WriteLine**(**number**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 10**

##### ****Example2: Using Single Method with Empty Data source****

In the following example, the data source is empty. When we apply the Single method on an empty data source then it will throw an exception as shown in the below example.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Sequence contains no element

List**<int>** numbers = new List**<int>()** **{** **}**;

**int** number = numbers.Single**()**;

Console.WriteLine**(**number**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

Now run the application and you will get the following exception.



As you can see when we run the application, we will get the above Invalid Operation Exception and it clearly shows the reason that the sequence contains no elements.

##### ****Example3:****

In the following example, we applied the Single method on a data source that contains more than one element.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//Sequence contains more than one element

List**<int>** numbers = new List**<int>()** **{**10, 20, 30 **}**;

**int** number = numbers.Single**()**;

Console.WriteLine**(**number**)**;

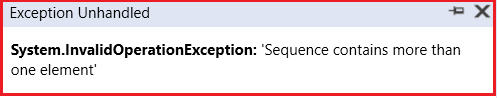
Console.ReadLine**()**;

**}**

**}**

**}**

When you run the above application you will get the following exception. This is because now the sequence contains more than one element.



Note: If your sequence contains more than one element and you need to fetch a single element based on some condition then you need to use the other overloaded versions of the Single method which takes one predicate as a parameter.

##### ****Example4:****

In the following example, we use the Single method which takes one predicate as a parameter. Using that predicate we specify our condition which is going to return only one element from the sequence.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**10, 20, 30 **}**;

**int** number = numbers.Single**(**num =**>** num == 20**)**;

Console.WriteLine**(**number**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: 20**

Here we will get the output as expected as the specified condition returns a single element from the sequence.

##### ****Example5:****

In the following example, the specified condition returns more than one element and hence we will get an exception saying the sequence contains more than one matching element.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**10, 20, 30 **}**;

**int** number = numbers.Single**(**num =**>** num **>** 10**)**;

Console.WriteLine**(**number**)**;

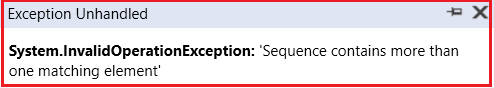
Console.ReadLine**()**;

**}**

**}**

**}**

Now run the application and you will get the following exception.



##### ****Example6:****

In the below example, the specified Single method does not return any data and hence we will get sequence contains no matching element exception.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**10, 20, 30 **}**;

**int** number = numbers.Single**(**num =**>** num **<** 10**)**;

Console.WriteLine**(**number**)**;

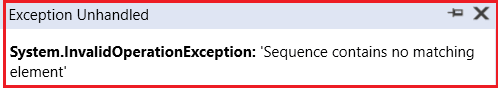
Console.ReadLine**()**;

**}**

**}**

**}**

When you run the application you will get the following exception.



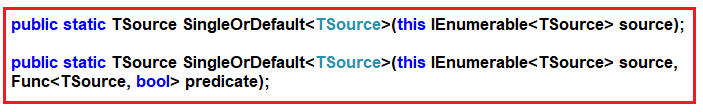
**Note:** If you don’t want to throw an exception when the sequence is empty or if the specified condition does not return an element from a sequence then you need to use the Linq SingleOrDefault method.

#### ****SingleOrDefault Method in Linq:****

The Linq SingleOrDefault method is very much similar to the Linq Single method except that this method will not throw an exception when the sequence is empty or when no element in the sequence satisfied the given condition.

**Note:** The most important point that you need to keep in mind is, like the Single method, the SingleOrDefault method still throws an exception when the sequence contains more than one matching element for the given condition.

There are two overloaded versions available for this method as shown in the below image.



As you can see the first overloaded version of this method does not take any parameter. It simply returns the only element from the sequence. It the sequence is empty then it returns the default value without throwing an exception.

Using the second overloaded version of this method you can specify a condition. If the specified condition does not return any data then it will not throw an exception instead it returns the default value.

##### ****Example7:****

In the following example, the specified condition does not match any elements in the sequence. So, in this case, the SingleOrDefault method will return the “0” as “0” is the default value for the integer data type.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**10, 20, 30 **}**;

**int** number = numbers.SingleOrDefault**(**num =**>** num **<** 10**)**;

Console.WriteLine**(**number**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

##### ****Example8:****

In the following example, the sequence is empty. But as we use the SingleOrDefault method we will not get any exception when we run the below example instead we will get the default value 0.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** **}**;

**int** number = numbers.SingleOrDefault**(**num =**>** num **<** 10**)**;

Console.WriteLine**(**number**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

##### ****Example9:****

In the following example, we will get an exception. This is because now the sequence contains more than elements for the specified condition.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**10, 20, 30 **}**;

**int** number = numbers.SingleOrDefault**(**num =**>** num **>** 10**)**;

Console.WriteLine**(**number**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

###### ****What is the difference between Single and SingleOrDefault methods in Linq?****

Both Single and SingleOrDefault methods in Linq are used to returns a single element from a sequence. But if the sequence is empty or if no element is satisfied with the given condition, then the Single method will throw an exception while the SingleOrDefault method will not throw an exception instead it returns a default value.

# DefaultIfEmpty Method in Linq

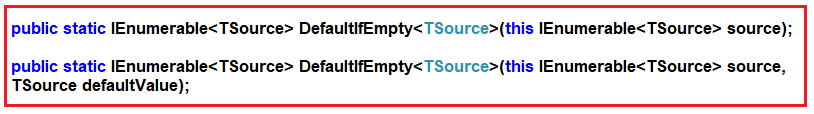
## ****DefaultIfEmpty Method in Linq****

In this article, I am going to discuss the **DefaultIfEmpty Method in Linq** with some examples. Please read our previous article before proceeding to this article, where we discussed the [**Single and SingleOrDefault method**](https://dotnettutorials.net/lesson/single-and-singleordefault-methods-in-linq/) with some examples.

#### ****Linq DefaultIfEmpty Method:****

If the sequence or data source on which the DefaultIfEmpty method is called is not empty, then the values of the original sequence or data source are going to be returned. On the other hand, if the sequence or data source is empty, then it returns a sequence with the default values based on the data type.

There are two overloaded versions available for this DefaultIfEmpty method in Linq which are shown in the below image.



The first overloaded version does not take any parameter and in this case, if the sequence is empty then it will return the default values based on the data type.

In the second overloaded version of the DefaultIfEmpty method, you can pass the default value and if the sequence is empty then this default value (what you pass to the method) is going to be returned by the method.

##### ****Example1:****

In the following example, the sequence is not empty. So, it is going to return a copy of the original values.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**10, 20, 30**}**;

IEnumerable**<int>** result = numbers.DefaultIfEmpty**()**;

**foreach** **(int** num in result**)**

**{**

Console.WriteLine**(**num**)**;

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

**Output:**

C:\Users\Pranaya\Pictures\DefaultIfEmpty Method in Linq Output.png

##### ****Example2:****

In the below example, the sequence is empty. So, in this case, it is going to return “0” as the default value. This is because 0 is the default value for the integer data type.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{}**;

IEnumerable**<int>** result = numbers.DefaultIfEmpty**()**;

**foreach** **(int** num in result**)**

**{**

Console.WriteLine**(**num**)**;

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

**Output:** **0**

##### ****Example3:****

In the following example, the sequence is empty but here we have supplied a default value (i.e. 5) to the DefaultIfEmpty method. So, in this case, the default value that we supplied (5) is going to be returned.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{}**;

IEnumerable**<int>** result = numbers.DefaultIfEmpty**(**5**)**;

**foreach** **(int** num in result**)**

**{**

Console.WriteLine**(**num**)**;

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

**Output:** 5

Note: If we supplied a default value but the sequence is not empty then, in that case, the original values that are present in the sequence is going to be returned.

##### ****Example4:****

In the below example, we have supplied a default value i.e. 5 to the DefaultIfEmpty method but the sequence is not empty. So, in this case, the elements which are present in the sequence are going to be returned.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**10, 20, 30**}**;

IEnumerable**<int>** result = numbers.DefaultIfEmpty**(**5**)**;

**foreach** **(int** num in result**)**

**{**

Console.WriteLine**(**num**)**;

**}**

Console.ReadLine**()**;

**}**

**}**

**}**

**Output:**

C:\Users\Pranaya\Pictures\DefaultIfEmpty Method in Linq Output.png

# SequenceEqual Operator in LINQ

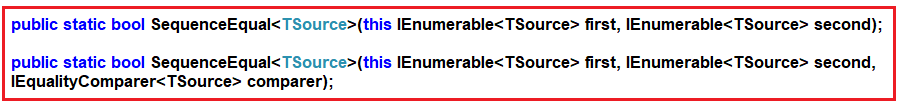
## ****SequenceEqual Operator in LINQ****

In this article, I am going to discuss the **SequenceEqual Operator in LINQ** with some examples. Please read our previous article before proceeding to this article where we discussed the [**DefaultIfEmpty method**](https://dotnettutorials.net/lesson/defaultifempty-method-in-linq/) in LINQ.

The **SequenceEqual Operator in Linq** is used to check whether two sequences are equal or not. If two sequences are equal then it returns true else it returns false.

Two sequences are considered to be equal when both the sequences have the same number of elements as well as the same values should be present in the same order.

There are two overloaded versions available for this SequenceEqual method as shown in the following image.



As you can see the second overloaded versions take an extra IEqualityComparer parameter.

##### ****Exampel1:****

In the following example, we created two sequences to store the cities. As you can see both the collections contains the same number of elements. Further, if you notice all the elements are present in the same order in both the collection. So here the sequences are equal and the SequenceEqual method is going to return true.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** cityList1 = new List**<**string**>** **{** "Delhi", "Mumbai", "Hyderabad" **}**;

List**<**string**>** cityList2 = new List**<**string**>** **{** "Delhi", "Mumbai", "Hyderabad" **}**;

**bool** IsEqual = cityList1.SequenceEqual**(**cityList2**)**;

Console.WriteLine**(**IsEqual**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: True**

##### ****Example2:****

The default comparer to the SequencesEqual method uses is case sensitive. So, in the below example, it returns false as the values are in case-sensitive.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** cityList1 = new List**<**string**>** **{** "DELHI", "mumbai", "Hyderabad" **}**;

List**<**string**>** cityList2 = new List**<**string**>** **{** "delhi", "MUMBAI", "Hyderabad" **}**;

**bool** IsEqual = cityList1.SequenceEqual**(**cityList2**)**;

Console.WriteLine**(**IsEqual**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: False**

##### ****Example3:****

If we want the comparison to be case-insensitive, then you need to use the other overloaded version of the SequenceEqual() method which takes IEqualityComparer as a parameter as shown in the below example.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** cityList1 = new List**<**string**>** **{** "DELHI", "mumbai", "Hyderabad" **}**;

List**<**string**>** cityList2 = new List**<**string**>** **{** "delhi", "MUMBAI", "Hyderabad" **}**;

**bool** IsEqual = cityList1.SequenceEqual**(**cityList2, StringComparer.OrdinalIgnoreCase**)**;

Console.WriteLine**(**IsEqual**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: True**

**Note:** If you go the definition of StringComparer class, then you will see that this class implements the IEqualityComparer interface.

##### ****Example4:****

In the following example, the SequenceEqual method returns false. This is because the data are not present in the same order in both the sequences.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** cityList1 = new List**<**string**>** **{** "Delhi", "Mumbai", "Hyderabad" **}**;

List**<**string**>** cityList2 = new List**<**string**>** **{** "Delhi", "Hyderabad", "Mumbai" **}**;

**bool** IsEqual = cityList1.SequenceEqual**(**cityList2, StringComparer.OrdinalIgnoreCase**)**;

Console.WriteLine**(**IsEqual**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: False**

##### ****Example5:****

If you want to solve the problem of the previous example, then first you need to sort the data and then apply the SequenceEqual method as shown in the below example.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** cityList1 = new List**<**string**>** **{** "Delhi", "Mumbai", "Hyderabad" **}**;

List**<**string**>** cityList2 = new List**<**string**>** **{** "Delhi", "Hyderabad", "Mumbai" **}**;

**bool** IsEqual = cityList1.OrderBy**(**city =**>** city**)**

.SequenceEqual**(**cityList2.OrderBy**(**city =**>** city**)**, StringComparer.OrdinalIgnoreCase**)**;

Console.WriteLine**(**IsEqual**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: True**

##### ****Working with Complex Type:****

We are going to work with the following Student class in this demo. So, create a class file with the name **Student.cs** and then copy and paste the following code in it.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetStudents1**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**{**ID= 101,Name = "Preety"**}**,

new Student**{**ID= 102,Name = "Priyanka"**}**

**}**;

**return** listStudents;

**}**

**public** **static** List**<**Student**>** GetStudents2**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**{**ID= 101,Name = "Preety"**}**,

new Student**{**ID= 102,Name = "Priyanka"**}**

**}**;

**return** listStudents;

**}**

**}**

**}**

Here in the above class, we created two methods i.e. GetStudents1 and GetStudents2. Further, if you notice these two methods are going to return the same data.

**Please modify the Main method of the Program class as shown below**.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentList1 = Student.GetStudents1**()**;

List**<**Student**>** StudentList2 = Student.GetStudents2**()**;

**bool** IsEqual = StudentList1.SequenceEqual**(**StudentList2**)**;

Console.WriteLine**(**IsEqual**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: False**

Both the sequence contains the same data but here we are getting the output as **False**. This is because when comparing the complex types, the default comparer which is used by the SequenceEqual method will only check if the object references are equal or not.

##### ****How to solve the above problem?****

There are many ways we can solve the above problem as follows.

1. We need to use the other overloaded version of the SequenceEqual method to which we can pass a custom class that implements the **IEqualityComparer** interface.
2. Project the properties into a new anonymous type, which overrides **Equals()** and **GetHashCode()** methods.
3. In the Student class override the**Equals()** and **GetHashCode()** methods.

##### ****Creating Custom StudentComparer class:****

Create a class file with the name **StudentComparer.cs** and then copy and paste the following code. As you can see this class implements the IEqualityComparer interface.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** StudentComparer : IEqualityComparer**<**Student**>**

**{**

**public** **bool** Equals**(**Student x, Student y**)**

**{**

**return** x.ID == y.ID && x.Name == y.Name;

**}**

**public** **int** GetHashCode**(**Student obj**)**

**{**

**return** obj.ID.GetHashCode**()** ^ obj.Name.GetHashCode**()**;

**}**

**}**

**}**

##### ****Modifying the Main method:****

Now in the Main method of the Program class, we need to create an instance of StudentComparer class and then we need to pass that instance to the SequenceEqual method as shown below.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentList1 = Student.GetStudents1**()**;

List**<**Student**>** StudentList2 = Student.GetStudents2**()**;

StudentComparer studentComparer = new StudentComparer**()**;

**bool** IsEqual = StudentList1.SequenceEqual**(**StudentList2, studentComparer**)**;

Console.WriteLine**(**IsEqual**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

With the above changes, now run the application and it should return True.

##### ****Using Anonymous Type:****

In the below example, we are projecting the data into an anonymous type.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentList1 = Student.GetStudents1**()**;

List**<**Student**>** StudentList2 = Student.GetStudents2**()**;

var IsEqual = StudentList1

.Select**(**std =**>** new **{** std.ID, std.Name **})**

.SequenceEqual**(**StudentList2.Select**(**std =**>** new **{** std.ID, std.Name **}))**;

Console.WriteLine**(**IsEqual**)**;

Console.ReadLine**()**;

**}**

**}**

**}**

**Output: True**

##### ****Overriding the Equals and GetHashCode method in Student Class:****

Modify the Student class as shown below.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Student

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **static** List**<**Student**>** GetStudents1**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**{**ID= 101,Name = "Preety"**}**,

new Student**{**ID= 102,Name = "Priyanka"**}**

**}**;

**return** listStudents;

**}**

**public** **static** List**<**Student**>** GetStudents2**()**

**{**

List**<**Student**>** listStudents = new List**<**Student**>()**

**{**

new Student**{**ID= 101,Name = "Preety"**}**,

new Student**{**ID= 102,Name = "Priyanka"**}**

**}**;

**return** listStudents;

**}**

**public** **override** **bool** Equals**(object** x**)**

**{**

**return** this.ID == **((**Student**)**x**)**.ID && this.Name == **((**Student**)**x**)**.Name;

**}**

**public** **override** **int** GetHashCode**()**

**{**

**return** this.ID.GetHashCode**()** ^ this.Name.GetHashCode**()**;

**}**

**}**

**}**

Now modify the Main method of the Program class as shown below.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**Student**>** StudentList1 = Student.GetStudents1**()**;

List**<**Student**>** StudentList2 = Student.GetStudents2**()**;

var IsEqual = StudentList1.SequenceEqual**(**StudentList2**)**;

Console.WriteLine**(**IsEqual**)**;

Console.ReadLine**()**;

**}**

**}**

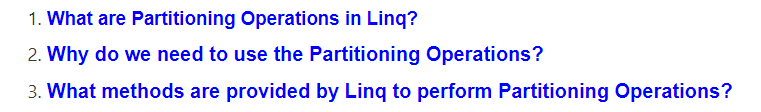
**}**

**Output: True**

# Partitioning Operators in Linq

## ****Partitioning Operators in Linq****

In this article, I am going to give you a brief introduction to **Partitioning Operations in Linq**. Please read our previous article before proceeding to this article where we discussed the [**SequenceEqual Method in Linq**](https://dotnettutorials.net/lesson/sequenceequal-operator-in-linq/) with some examples. As part of this article, we are going to discuss the following three-pointers.



##### ****What are Partitioning Operations in Linq?****

The Partitioning Operations in Linq are used to divide a sequence or you can say data source into two parts and then return one of them as output without changing the positions of the elements.

##### ****Why do we need Partitioning Operators?****

We need to use Partitioning operators when we want to perform the following operations.

1. When you want to select the top n number of records from a data source.
2. If you want to select records from a data source until a specified condition is true.
3. Select records from a data source except for the first n number of records.
4. Skip records from a data source until a specified condition is true and then select all records.
5. It can be used to implement pagination for a data source.

##### ****Partitioning Methods Provided by Linq:****

The following four methods are provided by LINQ to perform Partitioning Operations

1. [**Take**](https://dotnettutorials.net/lesson/take-operator-in-linq/)
2. [**Skip**](https://dotnettutorials.net/lesson/skip-method-linq/)
3. [**TakeWhile**](https://dotnettutorials.net/lesson/takewhile-method-linq/)
4. [**SkipWhile**](https://dotnettutorials.net/lesson/skipwhile-method-linq/)

# Take Operator in Linq

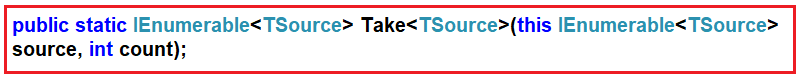
## ****Take Operator in Linq with Examples****

In this article, I am going to discuss the **Take Operator in Linq** with some examples. Please read our previous article before proceeding to this article where we gave a brief introduction to [**Partitioning Operators**](https://dotnettutorials.net/lesson/partitioning-operators-in-linq/)in Linq. As part of this article, we are going to discuss the following concepts related to the Take Operator in C#.

1. **What is Take Operator and its need in C#.NET?**
2. **Examples using Take Operator.**
3. **Take Operator in LINQ using Filter.**
4. **What happens when Applying the Where Operator after the Take Operator?**
5. **Take Method on the Data Source which is null?**

#### ****What is Take Operator and its need in C#.NET?****

The **Take Operator in Linq** is used to fetch the first **“n”** number of elements from the data source where **“n”** is an integer which is passed as a parameter to the Take method. There is only one version available for this Take method as shown below.



As you can see in the above image, the take method takes one integer count parameter and this method going to return that number of contiguous elements from the data source. If the data source is null then it is going to throw an exception.

You can use the Take method with both the **Method and Query Syntax** but the most important point that you need to remember is it will not make any changes to the positions of the elements.

##### ****Take Method in Linq with Method Syntax:****

In the following example, we created one integer collection which contains 10 elements. Then we retrieve the first 4 elements from the collection using the Method syntax.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

List**<int>** ResultMS = numbers.Take**(**4**)**.ToList**()**;

**foreach(**var num in ResultMS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 1 2 3 4**

##### ****Take Method in Linq with Query Syntax:****

Let us rewrite the previous example using query syntax. There is no such operator called Take is available to write the query syntax, So, here we need to use the mixed syntax as shown below.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

List**<int>** ResultQS = **(from** num in numbers

**select** num**)**.Take**(**4**)**.ToList**()**;

**foreach(**var num in ResultQS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 1 2 3 4**

##### ****Take Method with Filtering Operator in C#.NET:****

Our requirement is to take four elements from the collection but the condition is that the elements should be greater than 3. In such cases, first, we need to filter the data using Where operator and then we need to take the data using Take operator as shown in the below example.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

//Using Method Syntax

**int[]** ResultMS = numbers.Where**(**num =**>** num **>** 3**)**.Take**(**4**)**.ToArray**()**;

//Using Mixed Syntax

List**<int>** ResultQS = **(from** num in numbers

**where** num **>** 3

**select** num**)**.Take**(**4**)**.ToList**()**;

**foreach(**var num in ResultMS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 4 5 6 7**

##### ****Applying Filtering after the Take method in C#:****

If you apply the Filtering Operator after the Take method then you will not get the result as expected as shown in the below example.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

//Using Method Syntax

**int[]** ResultMS = numbers.Take**(**4**)**.Where**(**num =**>** num **>** 2**)**.ToArray**()**;

//Using Mixed Syntax

List**<int>** ResultQS = **(from** num in numbers

**select** num**)**.Take**(**4**)**.Where**(**num =**>** num **>** 2**)**.ToList**()**;

**foreach(**var num in ResultQS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 3 4**

As you can see in the above output, we are getting only 2 elements. This is because of the wrong use of Where operator. In this case, first, the “**Take**” method is executed which will return four elements and then on this result set the “W**here**” method is applied which further filter the data based on the condition specified. So, it is always a good programming practice to use the Where operator first and then the Take operator.

##### ****What happens when the Data Source is null?****

When we are applying the Take operator on a data source which is null, then we will get an exception i.e. ArgumentNullException as shown in the below example.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//data source is null

List**<int>** numbers = **null**;

**int[]** ResultMS = numbers.Where**(**num =**>** num **>** 3**)**.Take**(**4**)**.ToArray**()**;

**foreach(**var num in ResultMS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

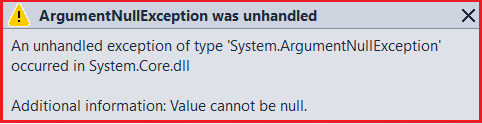
Console.ReadKey**()**;

**}**

**}**

**}**

When you run the application, you will get the following exception.



# TakeWhile Method in Linq

## ****TakeWhile Method in Linq with Examples****

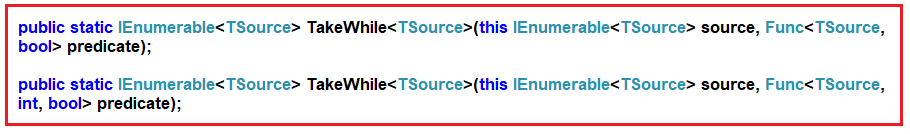
In this article, I am going to discuss the **TakeWhile Method in Linq** with some examples. Please read our previous article before proceeding to this article where we discussed the [**Take Method**](https://dotnettutorials.net/lesson/take-operator-in-linq/) with some examples. As part of this article, we are going to discuss the following pointers related to the TakeWhile Method in C#.

1. **What are the need and use of the TakeWhile Method in C#.NET?**
2. **Examples using both Method and Query Syntax.**
3. **Understanding TakeWhile Operator with Filtering Operator.**
4. **What is the difference between where and TakeWhile method in Linq?**
5. **Different Types of examples using the TakeWhile Operator in Linq.**

##### ****What are the need and use of the TakeWhile Method in C#.NET?****

The TakeWhile Method in Linq is used to fetch all the elements from a data source or sequence until a specified condition is true. Once the condition is failed, then the TakeWhile method will not check the rest of the elements presents in the data source even though the condition is true for the remaining elements.

The TakeWhile method will not make any changes to the positions of the elements. There are two overloaded versions available for this method as shown in the below image.



The First version of the TakeWhile method returns elements from a sequence as long as the specified condition is true.

The second overloaded version of this method returns elements from a sequence as long as the given condition is true. The second parameter of the function represents the index of the source elements. The element’s index can be used in the logic of the predicate function. If this is not clear at the moment then don’t worry we will discuss this with an example.

##### ****Example using TakeWhile Method in Linq:****

In the following example, we created a data source that contains numbers from 1 to 10. Then we fetch the records from the data source using the TakeWhile method with the condition that the number is less than 6.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

List**<int>** ResultMS = numbers.TakeWhile**(**num =**>** num **<** 6**)**.ToList**()**;

**foreach(**var num in ResultMS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 1 2 3 4 5**

##### ****TakeWhile Method along with the Where Filtering Method in C#.NET.****

Now you may have one question in your mind, that the same thing can be achieved using the Linq Where extension method as shown below.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

List**<int>** ResultMS = numbers.Where**(**num =**>** num **<** 6**)**.ToList**()**;

**foreach(**var num in ResultMS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 1 2 3 4 5**

##### ****What is the difference between where and TakeWhile method in Linq?****

Before understanding the difference let us modify the program as shown below and look at the output. Here we move the values 4 and 5 to the end of the collection.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 6, 7, 8, 9, 10, 4, 5 **}**;

List**<int>** Result1 = numbers.TakeWhile**(**num =**>** num **<** 6**)**.ToList**()**;

Console.Write**(**"Result Of TakeWhile Method: "**)**;

**foreach** **(**var num in Result1**)**

**{**

Console.Write**(**$"{ num} "**)**;

**}**

Console.WriteLine**()**;

//Using Where Method

List**<int>** Result2 = numbers.Where**(**num =**>** num **<** 6**)**.ToList**()**;

Console.Write**(**"Result Of Where Method: "**)**;

**foreach** **(**var num in Result2**)**

**{**

Console.Write**(**$"{ num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Difference between Where and TakeWhile method in Linq

As you can see in the above output, the “**TakeWhile**” method fetches the values 1, 2, and 3 while the “**Where**” method fetches the values 1, 2, 3, 4, and 5.

So, the difference is that the TakeWhile method checks the conditions from the beginning of the data source. As long as the condition is true it fetches the data. So, in our example, for the first three elements, the condition is true so it fetches the first three elements. When it checks the fourth elements the condition becomes false. So from that point, it will not check the rest of the elements in the data source even though some of the elements (i.e. 4 and 5 present at the end of the collections) satisfying the condition.

On the other hand, the Where method checks each and every element present in the collection. The elements which satisfy the condition will be returned. It does not matter the position of the elements in the sequence.

##### ****Example:****

In the following example, we have a collection of names. Here, we need to return names’ starting from the beginning until a name is hit that does not have a length greater than 3 characters.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** names = new List**<**string**>()** **{** "Sara", "Rahul", "John", "Pam", "Priyanka" **}**;

List**<**string**>** namesResult = names.TakeWhile**(**name =**>** name.Length **>** 3**)**.ToList**()**;

**foreach** **(**var name in namesResult**)**

**{**

Console.Write**(**$"{ name} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: Sara Rahul John**

##### ****Example:****

Let us use the other overloaded version which takes an index as a parameter. Here, in the following example, we will check the length of the name should be greater than its index position.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** names = new List**<**string**>()** **{** "Sara", "Rahul", "John", "Pam", "Priyanka" **}**;

List**<**string**>** namesResult = names.TakeWhile**((**name, index**)** =**>** name.Length **>** index**)**.ToList**()**;

**foreach** **(**var name in namesResult**)**

**{**

Console.Write**(**$"{ name} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: Sara Rahul John**

In the sequence the fourth element i.e. Pam has length 3 and its index position is 3. So, here the condition is false and hence it will not check the next element. As a result, it only fetches the first three elements from the sequence.

# Skip Method in Linq

## ****Skip Method in Linq with Examples****

In this article, I am going to discuss the **Skip Method in Linq** with some examples. Please read our previous article before proceeding to this article where we discussed the [**TakeWhile Method**](https://dotnettutorials.net/lesson/takewhile-method-linq/) with some examples. As part of this article, we are going to discuss the following pointers related to the Linq Skip Method.

1. **What is the Skip Method and its need in C#.NET?**
2. **Examples of Skip Method using both Method and Query Syntax.**
3. **Using the Skip Method with the Where Filtering Method.**
4. **What happens When we Applied the Filtering Operator after the Skip Operator in C#.NET?**
5. **Example when the Data source is null.**

##### ****What the Skip Method is and its need in C#.NET?****

The Skip Method in Linq is used to skip or bypass the first “**n**” number of elements from a data source or sequence and then returns the remaining elements from the data source as output. Here “**n**” is an integer value passed to the Skip method as a parameter. There is only one version available for this Skip method as shown below.



As you can see in the above image, the Skip method takes one integer count parameter and this method going to skip that number of elements from the beginning of the data source. If the data source is null then it is going to throw an exception.

##### ****Skip Method Using the Method syntax in LINQ:****

In the below example, the integer collection contains 10 elements. Here, we need to skip the first four elements and then need to retrieve the remaining elements.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

List**<int>** ResultMS = numbers.Skip**(**4**)**.ToList**()**;

**foreach** **(**var num in ResultMS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 5 6 7 8 9 10**

##### ****Skip Method Using the Query Syntax in LINQ:****

Let us rewrite the previous example using the query syntax. There is no such operator called Skip is available to write the query syntax. So, here we need to use the mixed syntax as shown below.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

List**<int>** ResultQS = **(from** num in numbers

**select** num**)**.Skip**(**4**)**.ToList**()**;

**foreach** **(**var num in ResultQS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 5 6 7 8 9 10**

##### ****Understanding the LINQ Skip Method with the Where Filtering Method:****

Our requirement is to skip four elements from the collection but the condition is that the elements should be greater than 3. In such cases, first, we need to filter the data using the Where extension method and then we need to skip or bypass the data using the Skip method as shown in the below example.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

//Using Method Syntax

**int[]** ResultMS = numbers.Where**(**num =**>** num **>** 3**)**.Skip**(**4**)**.ToArray**()**;

//Using Mixed Syntax

List**<int>** ResultQS = **(from** num in numbers

**where** num **>** 3

**select** num**)**.Skip**(**4**)**.ToList**()**;

**foreach** **(**var num in ResultMS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 8 9 10**

##### ****Applying Filter Method after the Skip method in C#.NET:****

If you apply the Filtering Operator such as “**Where**” after the “**Skip**” method then you will not get the output as expected as shown in the below example.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

//Using Method Syntax

**int[]** ResultMS = numbers.Skip**(**4**)**.Where**(**num =**>** num **<** 7**)**.ToArray**()**;

//Using Mixed Syntax

List**<int>** ResultQS = **(from** num in numbers

**select** num**)**.Skip**(**4**)**.Where**(**num =**>** num **<** 7**)**.ToList**()**;

**foreach** **(**var num in ResultQS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 5 6**

As you can see in the above output, we are getting only 2 elements. This is because of the wrong use of Where operator. In this case, first, the “**Skip**” method is executed which will skip the first four elements and return the remaining six elements and then on this result set the “**Where**” method is executed which will further filter the data based on the specified condition.

##### ****What happens when the Data Source is null?****

When we are applying the Skip Method on a data source which is null, then we will get an exception i.e. **ArgumentNullException** as shown in the below example.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

//data source is null

List**<int>** numbers = **null**;

**int[]** ResultMS = numbers.Skip**(**4**)**.ToArray**()**;

**foreach** **(**var num in ResultMS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

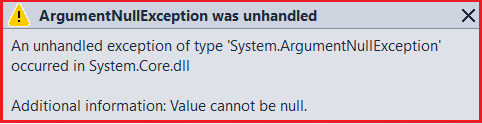
Console.ReadKey**()**;

**}**

**}**

**}**

When you run the application, you will get the following exception.



# SkipWhile Method in Linq

## ****SkipWhile Method in Linq with Examples****

In this article, I am going to discuss the **SkipWhile Method in Linq** with some examples. Please read our previous article before proceeding to this article where we discussed the [**Linq Skip Method**](https://dotnettutorials.net/lesson/skip-method-linq/)with some examples. As part of this article, we are going to discuss the following pointers related to the Linq Skip Method.

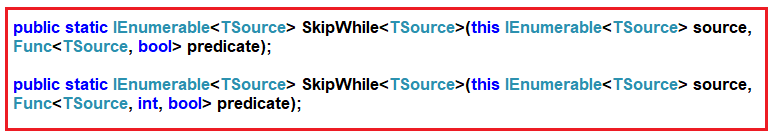
1. **What the SkipWhile Method is and its need in C#.NET?**
2. **Examples of SkipWhille Method using both Method and Query Syntax.**
3. **Example using Index.**

#### ****What the SkipWhile Method is and its need in C#.NET?****

The SkipWhile Method in Linq is used to skip or bypass all the elements from a data source or sequence as long as the given the condition specified by the predicate is true and then returns the remaining element from the sequence as an output.

The most important point that you need to remember is, once the condition is failed then the SkipWhile method will not check the rest of the elements even though the condition is true for some of the remaining elements.

There are two overloaded versions available for this method as shown in the below image.



The First overloaded version of the SkipWhile method Bypasses the elements in a sequence as long as a specified condition is true and then returns the remaining elements as an output.

The second overloaded version of this method bypass or skips the elements from a sequence as long as the given condition is true and then returns the remaining elements. The second parameter of the function represents the index of the source elements. The element’s index can be used in the logic of the predicate function. If this is not clear at the moment then don’t worry we will discuss this with an example.

##### ****Example1:****

In the below example, we created a data source that contains numbers from 1 to 10. Then we are fetching the elements from the collection using the SkipWhile method with the condition that the number is less than 5.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 **}**;

List**<int>** ResultMS = numbers.SkipWhile**(**num =**>** num **<** 5**)**.ToList**()**;

**foreach** **(**var num in ResultMS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 5 6 7 8 9 10**

##### ****Example2:****

Let us modify the position of the elements in the collection. Here we are moving the elements 2 and 3 to the end of the collection.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<int>** numbers = new List**<int>()** **{** 1, 4, 5, 6, 7, 8, 9, 10, 2, 3 **}**;

List**<int>** ResultMS = numbers.SkipWhile**(**num =**>** num **<** 5**)**.ToList**()**;

**foreach** **(**var num in ResultMS**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 5 6 7 8 9 10 2 3**

As you can see we are getting 2 and 3 in the output even they are smaller than 5. This is because the **SkipWhile** method checks the conditions from the beginning of the data source. As long as the condition is true it bypasses the data. So, in our example, for the first two elements, the condition is true so it skips the first two elements. When it checks the third elements the condition becomes false. So from that point, it will not check the rest of the elements in the data source even though some of the elements (i.e. 2 and 3 present at the end of the collections) satisfying the condition.

##### ****Example3:****

In the following example, we have a collection of names. Here, we need to skip the names’ starting from the beginning whose length is less than 4.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** names = new List**<**string**>()** **{** "Pam", "Rahul", "Kim", "Sara", "Priyanka" **}**;

List**<**string**>** namesResult = names.SkipWhile**(**name =**>** name.Length **<** 4**)**.ToList**()**;

**foreach** **(**var name in namesResult**)**

**{**

Console.Write**(**$"{ name} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: Rahul Kim Sara Priyanka**

##### ****Example4: Using Index****

Let us use the other overloaded version of the SkipWhile method which takes the index position of the element as a parameter. Here, in the following example, we will check the length of the name should be greater than its index position.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

List**<**string**>** names = new List**<**string**>()** **{** "Sara", "Rahul", "John", "Pam", "Priyanka" **}**;

List**<**string**>** namesResult = names.SkipWhile**((**name, index**)** =**>** name.Length **>** index**)**.ToList**()**;

**foreach** **(**var name in namesResult**)**

**{**

Console.Write**(**$"{ name} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: Pam Priyanka**

In the sequence the fourth element i.e. Pam has length 3 and its index position is 3. So, here the condition is false and hence it will not check the next element. As a result, it only fetches the last two elements from the sequence.

# Paging Using Skip and Take Method

## ****Paging Using Skip and Take Method****

In this article, I am going to discuss how to implement **Paging Using Skip and Take Method** of Linq. Before proceeding to this article, I strongly recommended you to read [**Skip Method in Linq**](https://dotnettutorials.net/lesson/skip-method-linq/)and [**Take Method in Linq**](https://dotnettutorials.net/lesson/take-operator-in-linq/)articles where we discussed the Skip and Take method in detail. As part of this article, we are going to discuss the following concepts.

1. **Why do we need a Paging in Real-time application?**
2. **What is paging?**
3. **Advantages and disadvantages of using Paging.**
4. **Example of implementing Paging using Skip and Take method in C#.NET**

##### ****Why we need Paging in Real-time application?****

Suppose we have a data source with lots of records and we need to display those records in a view. We can display all the records in a view at once. If we do so then we will get the following disadvantages.

1. Network issue (As huge data is traveled)
2. Memory Management (Due to heavy data showing in the view, it may cause memory issue and maybe the page becomes unresponsive)
3. Performance (we will get bad performance as it takes more time to travel in the network)

So, in order to solve the above problems, we need to go for Paging.

##### ****What is paging?****

Paging is nothing but a process in which we will divide a large data source into multiple pages. At one page we need to display a certain number of records. And next records can be visible with next – previous buttons or scroll or using any other techniques.

##### ****Advantages of using Paging****:

We will get the following advantages

1. Faster data transfer. This is because fewer data will be traveled in the network.
2. Improve memory management. This is because we are not showing all the data in a view.
3. Better performance.

**Drawback:**

In a client-server architecture, the number of requests between the client and server is increased. In such cases, you may get the data at once and store it locally and then implement the paging at the client-side.

##### ****How to implement paging?****

We can implement the paging using the Linq Skip and Take method. Here we need to understand two things one is PageNumber and the other one is the number of records per page.

Let say Page Number = PN and Number Of Records Per Page = NRP, then you need to use the following formula

**Result = DataSource.Skip((PN – 1) \* NRP).Take(NRP)**

##### ****Example:****

Here we are going to use the following Employee class. So, create a class file with the name Employee.cs and then copy and paste the following code.

**using** *System.Collections.Generic;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

Public **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** string Department **{** **get**; **set**; **}**

**public** **static** List**<**Employee**>** GetAllEmployees**()**

**{**

**return** new List**<**Employee**>()**

**{**

new Employee**()** **{**ID = 1, Name = "Pranaya", Department = "IT" **}**,

new Employee**()** **{**ID = 2, Name = "Priyanka", Department = "IT" **}**,

new Employee**()** **{**ID = 3, Name = "Preety", Department = "IT" **}**,

new Employee**()** **{**ID = 4, Name = "Sambit", Department = "IT" **}**,

new Employee**()** **{**ID = 5, Name = "Sudhanshu", Department = "HR" **}**,

new Employee**()** **{**ID = 6, Name = "santosh", Department = "HR" **}**,

new Employee**()** **{**ID = 7, Name = "Happy", Department = "HR" **}**,

new Employee**()** **{**ID = 8, Name = "Raja", Department = "IT" **}**,

new Employee**()** **{**ID = 9, Name = "Tarun", Department = "IT" **}**,

new Employee**()** **{**ID = 10, Name = "Bablu", Department = "IT" **}**,

new Employee**()** **{**ID = 11, Name = "Hina", Department = "HR" **}**,

new Employee**()** **{**ID = 12, Name = "Anurag", Department = "HR" **}**,

new Employee**()** **{**ID = 13, Name = "Dillip", Department = "HR" **}**,

new Employee**()** **{**ID = 14, Name = "Manoj", Department = "HR" **}**,

new Employee**()** **{**ID = 15, Name = "Lima", Department = "IT" **}**,

new Employee**()** **{**ID = 16, Name = "Sona", Department = "IT" **}**,

**}**;

**}**

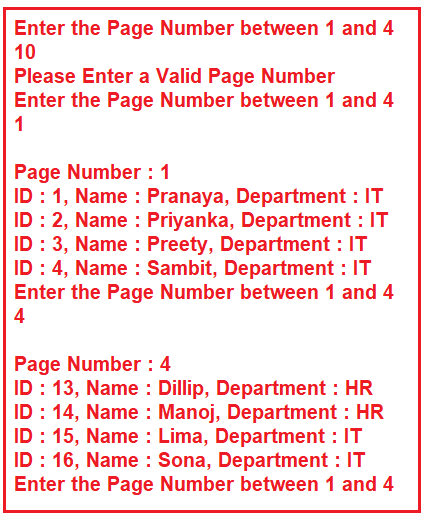
**}**

**}**

##### ****Business Requirement:****

First, the program should prompt the user to enter the page number. The page number must be between 1 and 4. If the user does not enter a valid page number, then the program should prompt the user to enter a valid page number. Once the user entered a valid page number, then the program should display the data.

For example, the output of the program something that looks as shown below.



**Note:** The condition in the do-while loop puts the program in an infinite loop. In order to end the program, simply close the console window.

The following console application implements paging using the Skip and Take method of Linq.

**using** *System;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int** RecordsPerPage = 4;

**int** PageNumber = 0;

**do**

**{**

Console.WriteLine**(**"Enter the Page Number between 1 and 4"**)**;

**if** **(int**.TryParse**(**Console.ReadLine**()**, out PageNumber**))**

**{**

**if(**PageNumber **>** 0 && PageNumber **<** 5**)**

**{**

var employees = Employee.GetAllEmployees**()**

.Skip**((**PageNumber - 1**)** \* RecordsPerPage**)**

.Take**(**RecordsPerPage**)**.ToList**()**;

Console.WriteLine**()**;

Console.WriteLine**(**"Page Number : " + PageNumber**)**;

**foreach** **(**var emp in employees**)**

**{**

Console.WriteLine**(**$"ID : {emp.ID}, Name : {emp.Name}, Department : {emp.Department}"**)**;

**}**

**}**

**else**

**{**

Console.WriteLine**(**"Please Enter a Valid Page Number"**)**;

**}**

**}**

**else**

**{**

Console.WriteLine**(**"Please Enter a Valid Page Number"**)**;

**}**

**}** **while** **(true)**;

**}**

**}**

**}**

# Linq Range Method in C#

## ****Linq Range Method in C# with Examples****

In this article, I am going to discuss the **Linq Range Method in C#** with some examples. The Linq Range Method belongs to the Generation Operator category. So let us first discuss what Generation Operator is and then we will see the Range method. Please read our previous article before proceeding to this article where we discussed [**How to Implement Paging using LINQ Skip and Take**](https://dotnettutorials.net/lesson/paging-using-skip-and-take-method/) method.

#### ****What is Generation Operator?****

As of now, we have discussed so many LINQ extension methods such as Select, Where, Any, etc. The Enumerable class also provides three non-extension methods are as follows.

1. **Range()**
2. **Repeat<T>()**
3. **Empty<T>()**

These methods allow us to create some specific type of array, sequence, or collection using a single expression instead of creating them manually and populating them using some kind of loops. That means these methods return a new sequence or collection that implements the **IEnumerable<T>** interface.

**Note:** In this article, I am going to discuss the LINQ Range Method and the rest two methods are going to discuss in our upcoming articles.

##### ****Linq Range Method in C#:****

This method is used to Generates a sequence of integral (integer) numbers within a specified range. The following is the signature of this method.



This method has 2 integer parameters. The first parameter (i.e. **int start**) specifies the value of the first integer in the sequence. The second parameter (i.e. **int count**) specifies the number of sequential integers to be generated. The return type of this method is **IEnumerable<int>** which going to contain a range of sequential integer numbers.

**Note:** When the count value is less than 0 or when the start + count – 1 value larger than the MaxValue then it will through ArgumentOutOfRangeException.

##### ****Example1:****

The following example generates a sequence of integers from 1 to 10 using the Linq Range method.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

IEnumerable**<int>** numberSequence = Enumerable.Range**(**1, 10**)**;

**foreach** **(int** i in numberSequence**)**

**{**

Console.WriteLine**(**i**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Once you run the application, it will print the values from 1 to 10 as expected.

##### ****Example2: Range method with Filter.****

The following print all the even numbers which are present between 10 and 30. Here, first we need to generate the sequence using range method and then we need to filter the data using the Where extension method.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

IEnumerable**<int>** EvenNumbers = Enumerable.Range**(**10, 30**)**.Where**(**x =**>** x%2 == 0**)**;

**foreach** **(int** num in EvenNumbers**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

##### Range method with Filter

**Example3:**

The following example shows how to generate a sequence of integers from 1 to 5 and then return the square of each number.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

IEnumerable**<int>** EvenNumbers = Enumerable.Range**(**1, 5**)**.Select**(**x =**>** x\*x**)**;

**foreach** **(int** num in EvenNumbers**)**

**{**

Console.Write**(**$"{num} "**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output: 1 4 9 16 25**

##### ****Example4: Range with string****

In the following example first, we generate a sequence and then we pass the sequence to our CustomLogic method and then merge the sequence with the return value from the CustomLogic method and return the result as **IEnumerable<string>**.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

IEnumerable**<**string**>** rangewithString = Enumerable.Range**(**1, 5**)**.Select**(**x =**>** **(**x \* x**)** + " " + CustomLogic**(**x**))**.ToArray**()**;

**foreach** **(**var item in rangewithString**)**

**{**

Console.WriteLine**(**item**)**;

**}**

Console.ReadKey**()**;

**}**

**private** **static** string CustomLogic**(int** x**)**

**{**

string result = string.Empty;

**switch** **(**x**)**

**{**

**case** 1:

result = "1st";

**break**;

**case** 2:

result = "2nd";

**break**;

**case** 3:

result = "3rd";

**break**;

**case** 4:

result = "4th";

**break**;

**case** 5:

result = "5th";

**break**;

**}**

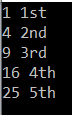
**return** result;

**}**

**}**

**}**

**Output:**



**Note:** This method is implemented by using the deferred execution. The immediate return value is an object that stores all the information that is required to perform the action. The query represented by this method is not executed until the object is enumerated either by calling its GetEnumerator method directly or by using a for each loop.

# Linq Repeat Method in C#

## ****Linq Repeat Method in C# with Examples****

In this article, I am going to discuss the **LINQ Repeat Method in C#** with an example. The Repeat method belongs to the Generation Operator category. Please read our previous article before proceeding to this article where we discussed the [**LINQ Range Method**](https://dotnettutorials.net/lesson/linq-range-method/) with some examples.

##### ****Linq Repeat Method in C#:****

The LINQ Repeat Method is used to generate a sequence or collection that with a specified number of elements and each element contains the same value. The following is the signature of this method.



The Repeat method has 2 integer parameters. The first parameter (i.e. **TResult element**) specifies the value to be repeated. The second parameter (i.e. **int count**) specifies the number of times to repeat the value. The return type of this method is **IEnumerable<int>** which is going to contain the repeated values. Here **TResult** specifies the data type of the value that is going to be repeated in the result sequence.

**Note:** When the count value is less than 0 then it will through **ArgumentOutOfRangeException**.

##### ****Example:****

The following example shows how to use the Repeat method to generate a sequence of a repeated value.

**using** *System.Linq;*

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

IEnumerable**<**string**>** repeatStrings = Enumerable.Repeat**(**"Welcome to DOT NET Tutorials", 10**)**;

**foreach** **(**String str in repeatStrings**)**

**{**

Console.WriteLine**(**str**)**;

**}**

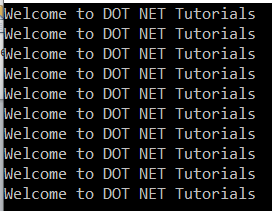
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Note:** The LINQ Repeat method is implemented using the deferred execution. So, the immediate return value is an object which stores all the required information to perform an action. The query represented by this method is not executed until the object is enumerated either by calling its GetEnumerator method directly or by using a for each loop.

# Linq Empty Method in C#

## ****Linq Empty Method in C# with Examples****

In this article, I am going to discuss the **LINQ Empty Method in C#** with some examples. Please read our previous article before proceeding to this article where we discussed the [**LINQ Repeat Method**](https://dotnettutorials.net/lesson/linq-repeat-method/) with an example.

##### ****Linq Empty Method in C#:****

The **Empty Method** belongs to the Generation Operator category. It is a static method included in the static Enumerable class. The **LINQ Empty Method** is used to return an empty collection (i.e. IEnumerable<T>) of a specified type. The following is the signature of this method.

Linq Empty Method in C# with Examples

Here **TResult** specifies the type parameter of the returned generic IEnumerable<TResult>. This method returns an empty IEnumerable<TResult> whose type argument is TResult.

**Note:** The main advantage of using the Empty method is “Even if you use an empty array or empty collection, then those are objects. As objects, they are going to be stored in the memory. Once they stored in memory, then it is the responsibility of Garbage Collector to take care of them”.

##### ****Example:****

In the following example, we have created two empty collections using the LINQ Empty Method. The first Empty method returns an empty collection of string while the second Empty method returns an empty collection of students.

**using** *System.Linq;*

**using** *System;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

var emptyCollection1 = Enumerable.Empty**<**string**>()**;

var emptyCollection2 = Enumerable.Empty**<**Student**>()**;

Console.WriteLine**(**"Count: {0} ", emptyCollection1.Count**())**;

Console.WriteLine**(**"Type: {0} ", emptyCollection1.GetType**()**.Name**)**;

Console.WriteLine**(**"Count: {0} ", emptyCollection2.Count**())**;

Console.WriteLine**(**"Type: {0} ", emptyCollection2.GetType**()**.Name**)**;

Console.ReadKey**()**;

**}**

**}**

**public** **class** Student

**{**

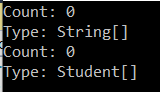
**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**}**

**}**

**Output:**



##### ****Why do we need the LINQ Empty Method?****

Let us understand the need for Empty Method with an example.

Consider one scenario, where our application calls a method which returns an **IEnumerable<int>**. There might be a situation where the method returns null. In the following example, the **GetData()** method is returning null. So, when we run the following program, it will throw NULL reference exception.

**using** *System.Collections.Generic;*

**using** *System;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

IEnumerable**<int>** integerSequence = GetData**()**;

**foreach** **(**var num in integerSequence**)**

**{**

Console.WriteLine**(**num**)**;

**}**

Console.ReadKey**()**;

**}**

**private** **static** IEnumerable**<int>** GetData**()**

**{**

**return** **null**;

**}**

**}**

**}**

The above problem can be fixed in two ways.

##### ****Solution1:****

We need to check for **NULL** before looping through the items in the collection as shown in the below example.

**using** *System.Collections.Generic;*

**using** *System;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

IEnumerable**<int>** integerSequence = GetData**()**;

**if(**integerSequence != **null)**

**{**

**foreach** **(**var num in integerSequence**)**

**{**

Console.WriteLine**(**num**)**;

**}**

**}**

Console.ReadKey**()**;

**}**

**private** **static** IEnumerable**<int>** GetData**()**

**{**

**return** **null**;

**}**

**}**

**}**

##### ****Solution2:****

We can solve the problem by using the LINQ Empty Method as shown in the below example. Here, we are using the **NULL-COALESCING** operator which checks if the **GetData()** method returns **NULL**, then initialized the **integerSequence** variable with an empty **IEnumerable<int>**.

**using** *System.Linq;*

**using** *System.Collections.Generic;*

**using** *System;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

IEnumerable**<int>** integerSequence = GetData**()** ?? Enumerable.Empty**<int>()**; ;

**foreach** **(**var num in integerSequence**)**

**{**

Console.WriteLine**(**num**)**;

**}**

Console.ReadKey**()**;

**}**

**private** **static** IEnumerable**<int>** GetData**()**

**{**

**return** **null**;

**}**

**}**

**}**

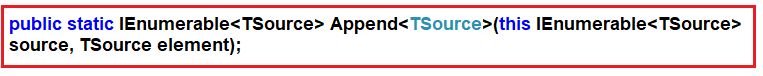
# Linq Append Method in C#

## ****Linq Append Method in C# with Example****

In this article, I am going to discuss the **LINQ Append Method in C#** with an example. Please read our previous article before proceeding to this article where we discussed the [**LINQ Empty Method**](https://dotnettutorials.net/lesson/linq-empty-method/) with some examples.

##### ****Linq Append Method in C#:****

The LINQ Append Method is used to append a value to the end of the sequence. This Append method does not modify the elements of the sequence. Instead, it creates a copy of the sequence with the new element. Following is the signature of the Append method.



**Type Parameters**

1. **TSource:** The data type of the elements contained in the sequence.

**Parameters:**

1. **IEnumerable<TSource> source:** A sequence of values.
2. **TSource element:** The value to append at the end of the sequence.

**Returns:**

1. **IEnumerable<TSource>:** A new sequence that ends with the element.

**Exceptions:** When the source is null, it will throw ArgumentNullException.

**Note:** This method is support from Framework 4.7.1 or later.

##### ****Example:****

The following example shows how to use the Append Method to append a value to the end of the sequence. The following example is self-explained. So, please go through the comment lines.

**using** *System.Linq;*

**using** *System.Collections.Generic;*

**using** *System;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

// Creating a list of integer

List**<int>** intSequence = new List**<int>** **{** 10, 20, 30, 40 **}**;

// Trying to append 5 at the end of the intSequence

intSequence.Append**(**5**)**;

//It doesn't work because the original list has not been changed

Console.WriteLine**(**string.Join**(**", ", intSequence**))**;

// It works now because we are using a changed copy of the original sequence

Console.WriteLine**(**string.Join**(**", ", intSequence.Append**(**5**)))**;

// Creating a new sequence explicitly

List**<int>** newintSequence = intSequence.Append**(**5**)**.ToList**()**;

// Printing the new sequence in the console

Console.WriteLine**(**string.Join**(**", ", newintSequence**))**;

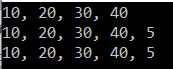
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



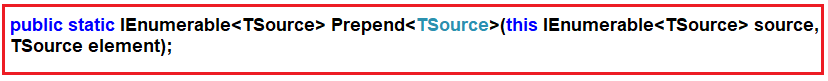
# Linq Prepend Method in C#

## ****Linq Prepend Method in C# With an Example****

In this article, I am going to discuss the **LINQ Prepend Method in C#** with an example. Please read our previous article before proceeding to this article where we discussed the [**LINQ Append Method**](https://dotnettutorials.net/lesson/linq-append-method/)with an example.

##### ****Linq Prepend Method in C#:****

The Linq Prepend Method is used to add one value to the beginning of a sequence. This Prepend method like the Append method does not modify the elements of the sequence. Instead, it creates a copy of the sequence with the new element. The signature of this is given below.



##### ****Type Parameters****

1. **TSource:** The data type of the elements contained in the sequence.

**Parameters:**

1. **IEnumerable<TSource> source:** A sequence of values.
2. **TSource element:** The value to prepend at the beginning of the sequence.

**Returns:**

1. **IEnumerable<TSource>:** A new sequence that begins with the element.

**Exceptions:** When the source is null, it will throw ArgumentNullException.

**Note:** This method is support from Framework 4.7.1 or later.

##### ****Example:****

The following example shows how to prepend a value to the beginning of the sequence using the Prepend method. The following example code is self-explained. So, please go through the comment lines.

**using** *System.Linq;*

**using** *System.Collections.Generic;*

**using** *System;*

**namespace** *LinqDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

// Creating a list of numbers

List**<int>** numberSequence = new List**<int>** **{** 10, 20, 30, 40 **}**;

// Trying to prepend 50

numberSequence.Prepend**(**50**)**;

// It will not work because the original sequence has not been changed

Console.WriteLine**(**string.Join**(**", ", numberSequence**))**;

// It works now because we are using a changed copy of the original list

Console.WriteLine**(**string.Join**(**", ", numberSequence.Prepend**(**50**)))**;

// If you prefer, you can create a new list explicitly

List**<int>** newnumberSequence = numberSequence.Prepend**(**50**)**.ToList**()**;

// And then write to the console output

Console.WriteLine**(**string.Join**(**", ", newnumberSequence**))**;

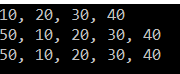
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



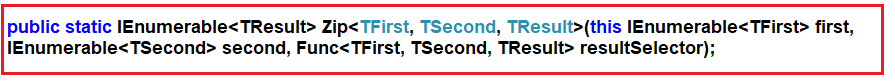
# Linq Zip Method in C#

## ****Linq Zip in C# with Example****

In this article, I am going to discuss the **LINQ Zip in C#** with examples. Please read our previous article before proceeding to this article where we discussed the [**LINQ Prepend Method**](https://dotnettutorials.net/lesson/linq-prepend-method/) with an example. At the end of this article, you will understand what is Linq Zip method and the need and use of C# Linq Zip Method with examples.

##### ****Linq Zip Method:****

The Linq Zip Method in C# is used to apply a specified function to the corresponding elements of two sequences and producing a sequence of the results. The signature of this method is given below.



##### ****Type Parameters:****

1. **TFirst:** The type of the elements of the first input sequence.
2. **TSecond:** The type of the elements of the second input sequence.
3. **TResult:** The type of the elements of the result sequence.

##### ****Parameters:****

1. **IEnumerable<TFirst> first:** The first sequence to merge.
2. **IEnumerable<TSecond> second:** The second sequence to merge.
3. **Func<TFirst,TSecond,TResult> resultSelector:** A function that specifies how to merge the elements from the two sequences.

**Returns:** This method is going to return an **IEnumerable<T>** that contains merged elements of two input sequences.

**Exceptions:** This method is going to throw **ArgumentNullException** when either the first or the second input sequence is null.

**Note:** The Zip method merges each element of the first sequence with an element in the second sequence that has the same index position. If both the sequences do not have the same number of elements, then the Zip method merges sequences until it reaches the end of the sequence which contains fewer elements. For example, if one sequence has five elements and the other sequence has four elements, then the result sequence will have only four elements.

##### ****Example:****

The following example shows how to merge two sequences using the Linq Zip method.

**using** *System;*

**using** *System.Linq;*

**namespace** *ThreadingDemo*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**int[]** numbersSequence = **{** 10, 20, 30, 40, 50 **}**;

string**[]** wordsSequence = **{** "Ten", "Twenty", "Thirty", "Fourty" **}**;

var resultSequence = numbersSequence.Zip**(**wordsSequence, **(**first, second**)** =**>** first + " - " + second**)**;

**foreach** **(**var item in resultSequence**)**

**{**

Console.WriteLine**(**item**)**;

**}**

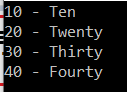
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



The first sequence contains 5 elements whereas the second sequence contains 4 elements. So, for the fifth element of the first sequence, there is no corresponding fifth element in the second sequence. As a result, the Zip method merges the four elements and that’s what we have seen in the output.

##### ****Note:****

The Zip method is implemented by using the deferred execution. So, the immediate return value of this method is going to be an object which stores all the required information which is required to perform the action. The query represented by this method is not executed until the object is enumerated either by calling its GetEnumerator method directly or by using for each loop.

# Deferred Execution vs Immediate Execution in LINQ

## ****Deferred Execution vs Immediate Execution in LINQ****

In this article, I am going to discuss the difference between **Deferred Execution vs Immediate Execution in LINQ** with some examples. Please read our previous article where we discussed the [**LINQ Zip Method**](https://dotnettutorials.net/lesson/linq-zip-method/) with an example. The LINQ queries are executed in two different ways as follows.

1. **Deferred execution**
2. **Immediate execution**

Based on the above two types of execution, the LINQ operators are divided into 2 categories. They are as follows:

1. **Deferred or Lazy Operators:**These query operators are used for deferred execution. For example – select, SelectMany, where, Take, Skip, etc. are belongs to Deferred or Lazy Operators category.
2. **Immediate or Greedy Operators:**These query operators are used for immediate execution. For Example – count, average, min, max, First, Last, ToArray, ToList, etc. are belongs to the Immediate or Greedy Operators category.

##### ****LINQ Deferred Execution:****

In this case, the LINQ Query is not executed at the point of its declaration. That means, when we write a LINQ query, it doesn’t execute by itself. It executes only when we access the query results. So, here the execution of the query is deferred until the query variable is iterated over using for each loop.

Let us understand this with an example. The below example is self-explained. So, please go through the comment lines.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

List**<**Employee**>** listEmployees = new List**<**Employee**>**

**{**

new Employee **{** ID= 1001, Name = "Priyanka", Salary = 80000 **}**,

new Employee **{** ID= 1002, Name = "Anurag", Salary = 90000 **}**,

new Employee **{** ID= 1003, Name = "Preety", Salary = 80000 **}**

**}**;

// In the below statement the LINQ Query is only defined and not executed

// If the query is executed here, then the result should not display Santosh

IEnumerable**<**Employee**>** result = **from** emp in listEmployees

**where** emp.Salary == 80000

**select** emp;

// Adding a new employee with Salary = 80000 to the collection listEmployees

listEmployees.Add**(**new Employee **{** ID = 1004, Name = "Santosh", Salary = 80000 **})**;

// The LINQ query is actually executed when we iterate thru using a for each loop

// This is proved because Santosh is also included in the result

**foreach** **(**Employee emp in result**)**

**{**

Console.WriteLine**(**$" {emp.ID} {emp.Name} {emp.Salary}"**)**;

**}**

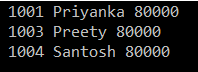
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Advantages of Deferred Execution:****

We will get the following advantages

1. It avoids unnecessary query execution which improves the performance of the application.
2. The Query creation and the Query execution are decoupled which provide us the flexibility to create the query in several steps.
3. A Linq deferred execution query is always re-evaluated when we re-enumerate. As a result, we always get the updated data.

##### ****Immediate Execution****

In the case of Immediate Execution, the LINQ query is executed at the point of its declaration. So, it forces the query to execute and gets the result immediately. Let us see an example for a better understanding. The following example is self-explained. So, please go through the comment lines.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

List**<**Employee**>** listEmployees = new List**<**Employee**>**

**{**

new Employee **{** ID= 1001, Name = "Priyanka", Salary = 80000 **}**,

new Employee **{** ID= 1002, Name = "Anurag", Salary = 90000 **}**,

new Employee **{** ID= 1003, Name = "Preety", Salary = 80000 **}**

**}**;

// In the following statement, the LINQ Query is executed immediately as we are

// Using the ToList() method which is a greedy operator which forces the query

// to be executed immediately

IEnumerable**<**Employee**>** result = **(from** emp in listEmployees

**where** emp.Salary == 80000

**select** emp**)**.ToList**()**;

// Adding a new employee with Salary = 80000 to the collection listEmployees

// will not have any effect on the result as the query is already executed

listEmployees.Add**(**new Employee **{** ID = 1004, Name = "Santosh", Salary = 80000 **})**;

// The above LINQ query is executed at the time of its creation.

// This is proved because Santosh is not included in the result

**foreach** **(**Employee emp in result**)**

**{**

Console.WriteLine**(**$" {emp.ID} {emp.Name} {emp.Salary}"**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

LINQ Immediate Execution in C#

##### ****Example:****

In the following Immediate Execution example, we are using the greedy operator which returns a single value.

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**namespace** *LINQDemo*

**{**

**public** **class** Employee

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **int** Salary **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

List**<**Employee**>** listEmployees = new List**<**Employee**>**

**{**

new Employee **{** ID= 1001, Name = "Priyanka", Salary = 80000 **}**,

new Employee **{** ID= 1002, Name = "Anurag", Salary = 90000 **}**,

new Employee **{** ID= 1003, Name = "Preety", Salary = 80000 **}**

**}**;

// In the following statement, the LINQ Query is executed immediately as we are

// Using the Count() method which is a greedy operator which forces the query

// to be executed immediately

var result = **(from** emp in listEmployees

**where** emp.Salary == 80000

**select** emp**)**.Count**()**;

// Adding a new employee with Salary = 80000 to the collection listEmployees

// will not have any effect on the result as the query is already executed

listEmployees.Add**(**new Employee **{** ID = 1004, Name = "Santosh", Salary = 80000 **})**;

// The LINQ query is executed at the time of its creation.

// This is proved because Santosh is not included in the count

Console.WriteLine**(**$" Employees with Salary 80000 : {result}"**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Deferred Execution vs Immediate Execution in LINQ

# ToList and ToArray Methods

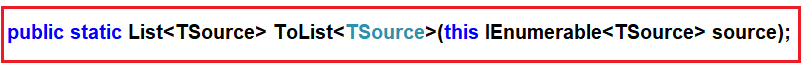
## ****ToList and ToArray Methods in C# with Examples****

In this article, I am going to discuss the **Linq ToList and ToArray Methods** **in C#** with some examples. Please read our previous article where we discussed the difference between [**Deferred Execution vs Immediate Execution in LINQ**](https://dotnettutorials.net/lesson/deferred-execution-vs-immediate-execution-in-linq/) with some examples. The ToList and ToArray methods belong to the conversion operator category.

##### ****ToList Method in LINQ:****

The **ToList** Method is used to create a **System.Collections.Generic.List<T>** collection from a **System.Collections.Generic.IEnumerable<T>**. This method causes the query to be executed immediately.

The signature of the **ToList** method is given below.

**Parameters:**  
**source:** The data type of source is **System.Collections.Generic.IEnumerable<T>**.

**Type parameters:**

**TSource:** The type of the elements contained in the source sequence.

**Returns:** It returns **System.Collections.Generic.List<T>** which contains elements from the source sequence.

**Exceptions:** This method throws **System.ArgumentNullException** when the source sequence is null.

##### ****Example: Convert int array to List<int> using the ToList method****

In the following example, first, we create an integer array and then we convert that array into a list by using the ToList method.

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**using** *System;*

**namespace** *ConversionOperators*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

//Creating Integer Array

**int[]** numbersArray = **{** 10, 22, 30, 40, 50, 60 **}**;

//Converting Integer Array to List using ToList method

List**<int>** numbersList = numbersArray.ToList**()**;

**foreach** **(**var num in numbersList**)**

**{**

Console.WriteLine**(**num**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

##### ****ToArray Method in Linq:****

The ToArray Method is used to copies the elements of **System.Collections.Generic.List<T>** to a new array. This method causes the query to be executed immediately. The signature of this method is given below.

Linq ToArray Method

Here T is the type of the array and this method converts a list into an array and returns that array containing copies of the elements of the **System.Collections.Generic.List<T>**.

##### ****Example: Convert List<int> to integer array.****

In the following example, first, we create a list of integers and then convert that list into an integer array by using the ToArray method.

**using** *System.Collections.Generic;*

**using** *System;*

**namespace** *ConversionOperators*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

//Create a List

List**<int>** numbersList = new List**<int>()**

**{**

10, 22, 30, 40, 50, 60

**}**;

//Converting List to Array

**int[]** numbersArray = numbersList.ToArray**()**;

**foreach** **(**var num in numbersArray**)**

**{**

Console.WriteLine**(**num**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

# ToDictionary Method

## ****ToDictionary Method in C#****

In this article, I am going to discuss the **Linq ToDictionary Method** in C# with some examples. Please read our previous article before proceeding to this article where we discussed the [**ToList and ToArray methods**](https://dotnettutorials.net/lesson/tolist-and-toarray-methods-linq/) with some examples. Like **ToList** and **ToArray**, the **ToDictionary** method also belongs to the conversion operator category.

##### ****ToDictionary Method in C#:****

The ToDictionary method in C# is used to creates a **System.Collections.Generic.Dictionary<TKey,TValue>** from an **System.Collections.Generic.IEnumerable<T>** according to a specified key selector. This method causes the query to be executed immediately.

There are four overloaded versions available for this method. Let us start the discussion with the following overloaded version.

##### ToDictionary Method in C#

##### ****Parameters:****

This method takes two parameters. They are as follows:

1. **source**: It is the **Collections.Generic.IEnumerable<T>** collection from which we need to create a **System.Collections.Generic.Dictionary<TKey,TValue>** collection.
2. **keySelector**: It is a function that is basically used to extract a key from each element.

##### ****Type parameters:****

1. **TSource**: The type of the elements of the source sequence.
2. **TKey**: The type of the key returned by keySelector.

**Returns:** It returns a **System.Collections.Generic.Dictionary<TKey,TValue>** collection that contains keys and values.

**Exceptions:** This method throws the following two exceptions.

1. It throws **ArgumentNullException** when the source or **keySelector** is **null** or the **keySelector** function produces a **key** that is **null**.
2. Throws **ArgumentException** when the **keySelector** produces duplicate keys for two elements.

##### ****Example: Converting List to a Dictionary.****

Here, in the following example, the product ID is the key and the product is its value.

**using** *System.Linq;*

**using** *System.Collections.Generic;*

**using** *System;*

**namespace** *ConversionOperators*

**{**

**public** **class** Product

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **double** Price **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

List**<**Product**>** listProducts = new List**<**Product**>**

**{**

new Product **{** ID= 1001, Name = "Mobile", Price = 800 **}**,

new Product **{** ID= 1002, Name = "Laptop", Price = 900 **}**,

new Product **{** ID= 1003, Name = "Desktop", Price = 800 **}**

**}**;

Dictionary**<int**, Product**>** productsDictionary = listProducts.ToDictionary**(**x =**>** x.ID**)**;

**foreach** **(**KeyValuePair**<int**, Product**>** kvp in productsDictionary**)**

**{**

Console.WriteLine**(**kvp.Key + " Name : " + kvp.Value.Name + ", Price: " + kvp.Value.Price**)**;

**}**

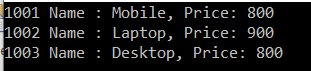
Console.ReadKey**()**;

**}**

**}**

**}**

When you run the above application, it will give the output as expected as shown below.



##### ****Another Overloaded version:****

The following **ToDictionary** method Creates a **System.Collections.Generic.Dictionary<TKey,TValue>** collection from an **System.Collections.Generic.IEnumerable<T>** according to the specified **key selector** and **element selector**.

##### Linq ToDictionary Method in C#

##### ****Parameters:****

1. **source:** It is the source System.Collections.Generic.IEnumerable<T> collection from where we need to create a System.Collections.Generic.Dictionary<TKey,TValue>.
2. **keySelector:** A function to extract a key from each element.
3. **elementSelector**: A transform function to produce a result element value from each element.

##### ****Type Parameters:****

1. **TSource**: The type of the elements of source.
2. **TKey**: The type of the key returned by keySelector.
3. **TElement**: The type of the value returned by the element selector.

**Returns:** It returns a **System.Collections.Generic.Dictionary<TKey,TValue>** that contains values of type **TElement** selected from the input sequence.

**Exceptions:** It throws **System.ArgumentNullException** when the **source** or **key selector** is null or the **key selector** function produces a key that is null. It also throws **System.ArgumentException** when the **key selector** produces duplicate keys for two elements.

##### ****Example:****

In the following example, we are converting **List<Product>** to a **Dictionary**. Here, the product ID is the key and the product name is its value.

**using** *System.Linq;*

**using** *System.Collections.Generic;*

**using** *System;*

**namespace** *ConversionOperators*

**{**

**public** **class** Product

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **double** Price **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

List**<**Product**>** listProducts = new List**<**Product**>**

**{**

new Product **{** ID= 1001, Name = "Mobile", Price = 800 **}**,

new Product **{** ID= 1002, Name = "Laptop", Price = 900 **}**,

new Product **{** ID= 1003, Name = "Desktop", Price = 800 **}**

**}**;

Dictionary**<int**, string**>** productsDictionary = listProducts.ToDictionary**(**x =**>** x.ID, x =**>** x.Name**)**;

**foreach** **(**KeyValuePair**<int**, string**>** kvp in productsDictionary**)**

**{**

Console.WriteLine**(**"Key : " + kvp.Key + " Value : " + kvp.Value**)**;

**}**

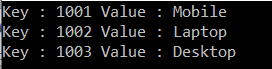
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Example:****

In the following example, it throws a **System.ArgumentException** as there are two products with the same ID (i.e. products with id 1001) and we are using ID as the key for the dictionary.

**namespace** *ConversionOperators*

**{**

**public** **class** Product

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **double** Price **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

List**<**Product**>** listProducts = new List**<**Product**>**

**{**

new Product **{** ID= 1001, Name = "Mobile", Price = 800 **}**,

new Product **{** ID= 1001, Name = "Laptop", Price = 900 **}**,

new Product **{** ID= 1003, Name = "Desktop", Price = 800 **}**

**}**;

Dictionary**<int**, string**>** productsDictionary = listProducts.ToDictionary**(**x =**>** x.ID, x =**>** x.Name**)**;

**foreach** **(**KeyValuePair**<int**, string**>** kvp in productsDictionary**)**

**{**

Console.WriteLine**(**"Key : " + kvp.Key + " Value : " + kvp.Value**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

##### ****Example:****

In the following example, it will throw **System.ArgumentNullException** as the source (i.e listProducts) is null.

**namespace** *ConversionOperators*

**{**

**public** **class** Product

**{**

**public** **int** ID **{** **get**; **set**; **}**

**public** string Name **{** **get**; **set**; **}**

**public** **double** Price **{** **get**; **set**; **}**

**}**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

List**<**Product**>** listProducts = **null**;

Dictionary**<int**, string**>** productsDictionary = listProducts.ToDictionary**(**x =**>** x.ID, x =**>** x.Name**)**;

**foreach** **(**KeyValuePair**<int**, string**>** kvp in productsDictionary**)**

**{**

Console.WriteLine**(**"Key : " + kvp.Key + " Value : " + kvp.Value**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

# Cast Operator

## ****Cast Operator in C# with Examples:****

In this article, I am going to discuss the Cast Operator in C# with some examples. Please read our previous article before proceeding to this article where we discussed the [**ToDictionary Method in C#**](https://dotnettutorials.net/lesson/todictionary-method/) with some examples. Like **ToList, ToArray**, **ToDictionary**, the **Cast Operator** also belongs to the conversion operators category.

##### ****Cast Operator in C#:****

The **Cast Operator in C#** is used to casts all the elements of a collection (**System.Collections.IEnumerable**) to a specified type and then return a new **System.Collections.Generic.IEnumerable<T>** collection which contains all the elements of the source sequence cast to the specified type. This method uses deferred execution.

The signature of this method is given below.

##### Cast Operator in C#

##### ****Parameters:****

1. **source:** The **System.Collections.IEnumerable** that contains the elements to be cast to type **TResult**.

##### ****Type parameters:****

1. **TResult:** The type to cast the elements of source.

**Exceptions:** It will throw **System.ArgumentNullException** when the source is null.  It also throws **System.InvalidCastException** when an element in the source sequence cannot be cast to the specified type TResult.

##### ****Example:****

In the following example, the source sequence contains 3 elements and all these three elements are of type integer. So, all these elements are cast to integers.

**using** *System.Linq;*

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Collections;*

**namespace** *ConversionOperators*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

ArrayList list = new ArrayList

**{**

10,

20,

30

**}**;

IEnumerable**<int>** result = list.Cast**<int>()**;

**foreach** **(int** i in result**)**

**{**

Console.WriteLine**(**i**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:** 10 20 30

##### ****Example2:****

In the following example, the line “**list.Add(“40”);**” will throw an exception. This is because it is a string value and we are trying to convert it into an int. So, it will throw **System.InvalidCastException**.

**using** *System.Linq;*

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Collections;*

**namespace** *ConversionOperators*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

ArrayList list = new ArrayList

**{**

10,

20,

30,

**}**;

//The following statement throws System.InvalidCastException

list.Add**(**"40"**)**;

IEnumerable**<int>** result = list.Cast**<int>()**;

**foreach** **(int** i in result**)**

**{**

Console.WriteLine**(**i**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

##### ****Example3:****

In the following example, the source sequence is null. So, when we run the application, it will throw **System.ArgumentNullException**.

**using** *System.Linq;*

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Collections;*

**namespace** *ConversionOperators*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

ArrayList list = **null**;

//Throws System.ArgumentNullException

IEnumerable**<int>** result = list.Cast**<int>()**;

**foreach** **(int** i in result**)**

**{**

Console.WriteLine**(**i**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

# Difference between Cast and OfType Operators

## ****Difference between Cast and OfType Operators in C#****

In this article, I am going to discuss the **difference between Cast and OfType Operators** in C# with an example. This is one of the frequently asked interview questions. For a better understanding of this question, I strongly recommended you to read the following two articles before proceeding to this article.

1. [**Cast Operator in C#**](https://dotnettutorials.net/lesson/cast-operator/)
2. [**OfType Operator in C#**](https://dotnettutorials.net/lesson/oftype-operator-in-linq/)

##### ****Difference between Cast and OfType operators:****

The Cast operator in C# will try to cast all the elements present in the source sequence into a specified type. If any of the elements in the source sequence cannot be cast to the specified type then it will throw **InvalidCastException**.

On the other hand, the OfType operator in C# returns only the elements of the specified type and the rest of the elements in the source sequence will be ignored as well as excluded from the result.

##### ****Cast Operator Example:****

In the following example, the Cast operator throws **InvalidCastException** as it is unable the Cast the value **“50”** (it is in the form of a string) to an integer.

**using** *System.Linq;*

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Collections;*

**namespace** *ConversionOperators*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

ArrayList list = new ArrayList

**{**

10,

20,

30,

"50"

**}**;

IEnumerable**<int>** result = list.Cast**<int>()**;

**foreach** **(int** i in result**)**

**{**

Console.WriteLine**(**i**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

Now run the above application, it should give you the following exception.



##### ****OfType Operator Example:****

Let us rewrite the same example using the **OfType** operator as shown below.

**using** *System.Linq;*

**using** *System.Collections.Generic;*

**using** *System;*

**using** *System.Collections;*

**namespace** *ConversionOperators*

**{**

**class** Program

**{**

**public** **static** **void** Main**()**

**{**

ArrayList list = new ArrayList

**{**

10,

20,

30,

"50"

**}**;

IEnumerable**<int>** result = list.OfType**<int>()**;

**foreach** **(int** i in result**)**

**{**

Console.WriteLine**(**i**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:** 10 20 30

Now, when you run the application, you will not get any exception. The value “50” is ignored as well as excluded from the result.

##### ****When to use Cast over OfType and vice versa?****

We need to use the Cast Operator

1. When we want to cast all the elements in the collection
2. When we know for sure the collection contains only elements of the specified type

We need to use the OfType operator

1. When we want to filter the elements and return only the specified type of elements.