Language-Integrated Query (LINQ) is a powerful query language introduced with .Net 3.5 & Visual Studio 2008. LINQ can be For example, SQL is a Structured Query Language used to save and retrieve data from a database. In the same way, LINQ is a structured query syntax built in C# and VB.NET to retrieve data from different types of data sources such as collections, ADO.Net DataSet, XML Docs, web service and MS SQL Server and other databases.used with C# or Visual Basic to query different data sources.

## Advantages of LINQ

* **Familiar language:**Developers don’t have to learn a new query language for each type of data source or data format.
* **Less coding:**It reduces the amount of code to be written as compared with a more traditional approach.
* **Readable code:**LINQ makes the code more readable so other developers can easily understand and maintain it.
* **Standardized way of querying multiple data sources:**The same LINQ syntax can be used to query multiple data sources.
* **Compile time safety of queries:**It provides type checking of objects at compile time.
* **IntelliSense Support:**LINQ provides IntelliSense for generic collections.
* **Shaping data:**You can retrieve data in different shapes.

Example: LINQ Query to Array

// Data source

string[] names = {"Bill", "Steve", "James", "Mohan" };

// LINQ Query

var myLinqQuery = from name in names

where name.Contains('a')

select name;

// Query execution

foreach(var name in myLinqQuery)

Console.Write(name + " ");

Why LINQ?

To understand why we should use LINQ, let's look at some examples. Suppose you want to find list of teenage students from an array of Student objects.

Before C# 2.0, we had to use a 'foreach' or a 'for' loop to traverse the collection to find a particular object. For example, we had to write the following code to find all Student objects from an array of Students where the age is between 12 and 20 (for teenage 13 to 19

Example: Use for loop to find elements from the collection in C# 1.0

class Student

{

public int StudentID { get; set; }

public String StudentName { get; set; }

public int Age { get; set; }

}

class Program

{

static void Main(string[] args)

{

Student[] studentArray = {

new Student() { StudentID = 1, StudentName = "John", Age = 18 },

new Student() { StudentID = 2, StudentName = "Steve", Age = 21 },

new Student() { StudentID = 3, StudentName = "Bill", Age = 25 },

new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 },

new Student() { StudentID = 5, StudentName = "Ron" , Age = 31 },

new Student() { StudentID = 6, StudentName = "Chris", Age = 17 },

new Student() { StudentID = 7, StudentName = "Rob",Age = 19 },

};

Student[] students = new Student[10];

int i = 0;

foreach (Student std in studentArray)

{

if (std.Age > 12 && std.Age < 20)

{

students[i] = std;

i++;

}

}

}

}

Example: Use Delegates to Find Elements from the Collection in C# 2.0

delegate bool FindStudent(Student std);

class StudentExtension

{

public static Student[] where(Student[] stdArray, FindStudent del)

{

int i=0;

Student[] result = new Student[10];

foreach (Student std in stdArray)

if (del(std))

{

result[i] = std;

i++;

}

return result;

}

}

class Program

{

static void Main(string[] args)

{

Student[] studentArray = {

new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,

new Student() { StudentID = 2, StudentName = "Steve", Age = 21 } ,

new Student() { StudentID = 3, StudentName = "Bill", Age = 25 } ,

new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,

new Student() { StudentID = 5, StudentName = "Ron" , Age = 31 } ,

new Student() { StudentID = 6, StudentName = "Chris", Age = 17 } ,

new Student() { StudentID = 7, StudentName = "Rob",Age = 19 } ,

};

Student[] students = StudentExtension.where(studentArray, delegate(Student std){

return std.Age > 12 && std.Age < 20;

});

}

}

}

So, with C# 2.0, you got the advantage of **delegate** in finding students with any criteria. You don't have to use a for loop to find students using different criteria. For example, you can use the same delegate function to find a student whose StudentId is 5 or whose name is Bill, as below:

Student[] students = StudentExtension.where(studentArray, delegate(Student std) {

return std.StudentID == 5;

});

//Also, use another criteria using same delegate

Student[] students = StudentExtension.where(studentArray, delegate(Student std) {

return std.StudentName == "Bill";

});

he C# team felt that they still needed to make the code even more compact and readable. So they introduced the extension method, lambda expression, expression tree, anonymous type and query expression in [C# 3.0](https://www.tutorialsteacher.com/csharp/csharp-version-history). You can use these features of C# 3.0, which are building blocks of LINQ to query to the different types of collection and get the resulted element(s) in a single statement.

The example below shows how you can use LINQ query with lambda expression to find a particular student(s) from the student collection.

Example: LINQ

class Program

{

static void Main(string[] args)

{

Student[] studentArray = {

new Student() { StudentID = 1, StudentName = "John", age = 18 } ,

new Student() { StudentID = 2, StudentName = "Steve", age = 21 } ,

new Student() { StudentID = 3, StudentName = "Bill", age = 25 } ,

new Student() { StudentID = 4, StudentName = "Ram" , age = 20 } ,

new Student() { StudentID = 5, StudentName = "Ron" , age = 31 } ,

new Student() { StudentID = 6, StudentName = "Chris", age = 17 } ,

new Student() { StudentID = 7, StudentName = "Rob",age = 19 } ,

};

// Use LINQ to find teenager students

Student[] teenAgerStudents = studentArray.Where(s => s.age > 12 && s.age < 20).ToArray();

// Use LINQ to find first student whose name is Bill

Student bill = studentArray.Where(s => s.StudentName == "Bill").FirstOrDefault();

// Use LINQ to find student whose StudentID is 5

Student student5 = studentArray.Where(s => s.StudentID == 5).FirstOrDefault();

}

}

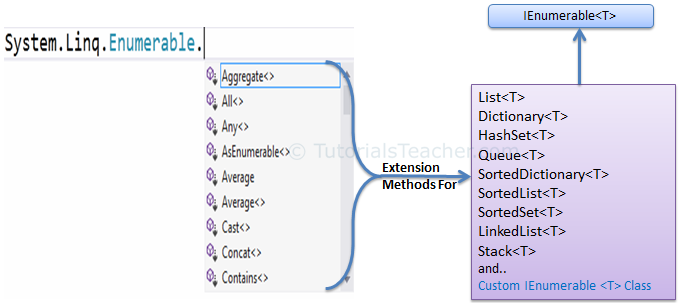
# LINQ API

We can write LINQ queries for the classes that implement [IEnumerable<T>](https://msdn.microsoft.com/en-us/library/9eekhta0(v=vs.110).aspx) or [IQueryable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.linq.iqueryable-1) interface. The [*System.Linq*](https://msdn.microsoft.com/en-us/library/system.linq(v=vs.110).aspx) namespace includes the following classes and interfaces require for LINQ queries.

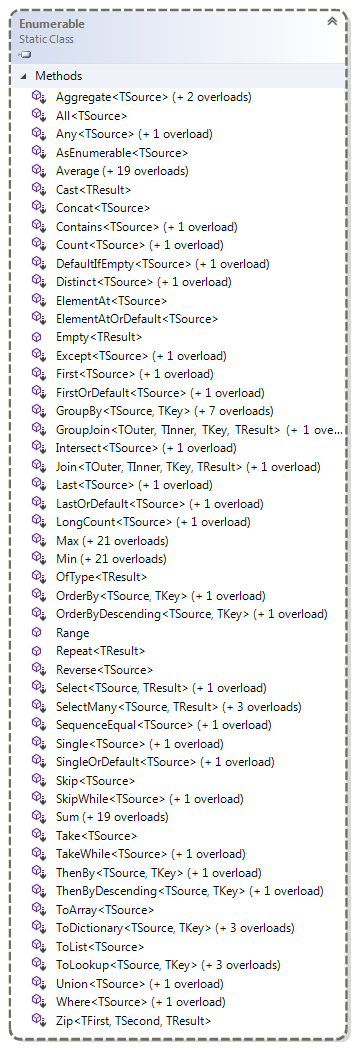
## Enumerable

The [Enumerable](https://msdn.microsoft.com/en-us/library/system.linq.enumerable(v=vs.110).aspx)) class includes extension methods for the classes that implement IEnumerable<T> interface, for example all the built-in collection classes implement IEnumerable<T> interface and so we can write LINQ queries to retrieve data from the built-in collections.

The following figure shows the extension methods included in Enumerable class that can be used with the generic collections in C# or VB.Net.

[](https://www.tutorialsteacher.com/Content/images/linq/Enumerable-extension-methods.png)

The following figure shows all the extension methods available in Enumerable class.

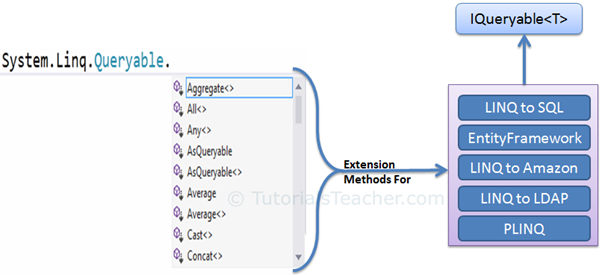
[](https://www.tutorialsteacher.com/Content/images/linq/Enumerable.png)Enumerable Class

## Queryable

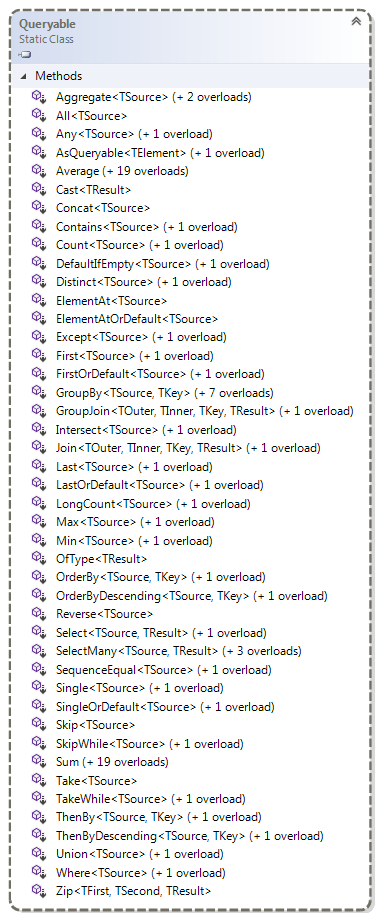
The [Queryable](https://msdn.microsoft.com/en-us/library/system.linq.queryable(v=vs.110).aspx) class includes extension methods for classes that implement [IQueryable<t>](https://msdn.microsoft.com/en-us/library/vstudio/bb351562(v=vs.100).aspx) interface. The IQueryable<T> interface is used to provide querying capabilities against a specific data source where the type of the data is known. For example, Entity Framework api implements IQueryable<T> interface to support LINQ queries with underlaying databases such as MS SQL Server.

Also, there are APIs available to access third party data; for example, LINQ to Amazon provides the ability to use LINQ with Amazon web services to search for books and other items. This can be achieved by implementing the IQueryable interface for Amazon.

The following figure shows the extension methods available in the Queryable class can be used with various native or third party data providers.

[](https://www.tutorialsteacher.com/Content/images/linq/Queryable-extension-methods.png)

The following figure shows the extension methods available in the Queryable class.

[](https://www.tutorialsteacher.com/Content/images/linq/queryable.png)Queryable class

LINQ Query Syntax

There are two basic ways to write a LINQ query to IEnumerable collection or IQueryable data sources.

1. Query Syntax or Query Expression Syntax
2. Method Syntax or Method Extension Syntax or Fluent

## Query Syntax

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

LINQ Query Syntax:

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

<Standard Query Operators> *<lambda expression>*

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

The LINQ query syntax starts with from keyword and ends with select keyword. The following is a sample LINQ query that returns a collection of strings which contains a word "Tutorials".

Example: LINQ Query Syntax in C#

// string collection

IList<string> stringList = new List<string>() {

"C# Tutorials",

"VB.NET Tutorials",

"Learn C++",

"MVC Tutorials" ,

"Java"

};

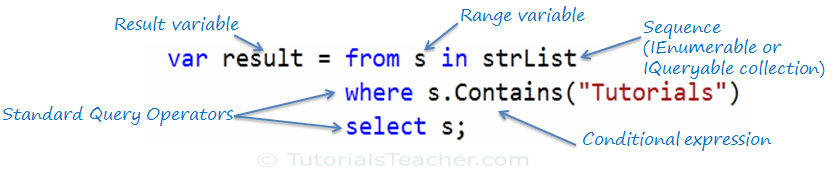
// LINQ Query Syntax

var result = from s in stringList

where s.Contains("Tutorials")

select s;

The following figure shows the structure of LINQ query syntax.

[](https://www.tutorialsteacher.com/Content/images/linq/linq-query-syntax.png)LINQ Query Syntax

Query syntax starts with a ***From*** clause followed by a ***Range*** variable. The ***From*** clause is structured like "**From** rangeV*ariableName* **in** *IEnumerablecollection*". In English, this means, from each object in the collection. It is similar to a foreach loop: foreach(Student s in studentList).

After the From clause, you can use different Standard Query Operators to filter, group, join elements of the collection. There are around 50 Standard Query Operators available in LINQ. In the above figure, we have used "where" operator (aka clause) followed by a condition. This condition is generally expressed using [lambda expression](https://www.tutorialsteacher.com/linq/linq-lambda-expression).

LINQ query syntax always ends with a Select or Group clause. The Select clause is used to shape the data. You can select the whole object as it is or only some properties of it. In the above example, we selected the each resulted string elements.

In the following example, we use LINQ query syntax to find out teenager students from the Student collection (sequence).

Example: LINQ Query Syntax in C#

// Student collection

IList<Student> studentList = new List<Student>() {

new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,

new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,

new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

};

// LINQ Query Syntax to find out teenager students

var teenAgerStudent = from s in studentList

where s.Age > 12 && s.Age < 20

select s;

LINQ Method Syntax

In the previous section, you have learned about LINQ Query Syntax. Here, you will learn about Method syntax.

https://www.tutorialsteacher.com/Content/images/tips.pngThe compiler converts query syntax into method syntax at compile time.

Method syntax (also known as fluent syntax) uses extension methods included in the [Enumerable](https://msdn.microsoft.com/en-us/library/system.linq.enumerable(v=vs.110).aspx)) or [Queryable](https://msdn.microsoft.com/en-us/library/system.linq.queryable(v=vs.110).aspx) static class, similar to how you would call the extension method of any class.

The following is a sample LINQ method syntax query that returns a collection of strings which contains a word "Tutorials".

Example: LINQ Method Syntax in C#

// string collection

IList<string> stringList = new List<string>() {

"C# Tutorials",

"VB.NET Tutorials",

"Learn C++",

"MVC Tutorials" ,

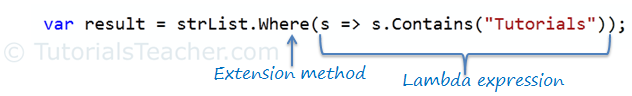
"Java"

};

// LINQ Query Syntax

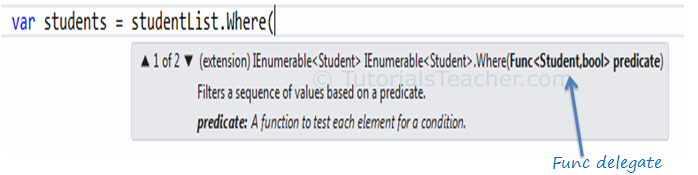
var result = stringList.Where(s => s.Contains("Tutorials"));

The following figure illustrates the structure of LINQ method syntax.

[](https://www.tutorialsteacher.com/Content/images/linq/linq-method-syntax.png)LINQ Method Syntax Structure

As you can see in the above figure, method syntax comprises of extension methods and Lambda expression. The extension method **Where()** is defined in the Enumerable class.

If you check the signature of the Where extension method, you will find the Where method accepts a [predicate](https://www.tutorialsteacher.com/csharp/csharp-predicate) delegate as Func<Student, bool>. This means you can pass any delegate function that accepts a Student object as an input parameter and returns a Boolean value as shown in the below figure. The lambda expression works as a delegate passed in the Where clause. Learn lambda expression in the next section.

[](https://www.tutorialsteacher.com/Content/images/linq/linq-where-extension-method.png)Func delegate in Where

The following example shows how to use LINQ method syntax query with the IEnumerable<T> collection.

Example: Method Syntax in C#

// Student collection

IList<Student> studentList = new List<Student>() {

new Student() { StudentID = 1, StudentName = "John", Age = 13} ,

new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,

new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,

new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,

new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }

};

// LINQ Method Syntax to find out teenager students

var teenAgerStudents = studentList.Where(s => s.Age > 12 && s.Age < 20)

.ToList<Student>();

Example: Method Syntax in VB.Net

// Student collection

Dim studentList = New List(Of Student) From {

New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},

New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},

New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},

New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},

New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}

}

// LINQ Method Syntax to find out teenager students

Dim teenAgerStudents As IList(Of Student) = studentList.Where(Function(s) s.Age > 12 And s.Age < 20)

.ToList()

# Anatomy of the Lambda Expression

C# 3.0(.NET 3.5) introduced the lambda expression along with LINQ. The lambda expression is a shorter way of representing [anonymous method](https://www.tutorialsteacher.com/csharp/csharp-anonymous-method) using some special syntax.

For example, following anonymous method checks if student is teenager or not:

Example: Anonymous Method in C#

delegate(Student s) { return s.Age > 12 && s.Age < 20; };

Example: Anonymous method in VB.Net

Dim isStudentTeenAger = Function(s As Student) As Boolean

Return s.Age > 12 And s.Age < 20

End Function

The above anonymous method can be represented using a Lambda Expression in C# and VB.Net as below:

Example: Lambda Expression in C#

s => s.Age > 12 && s.Age < 20

Example: Lambda Expression in VB.Net

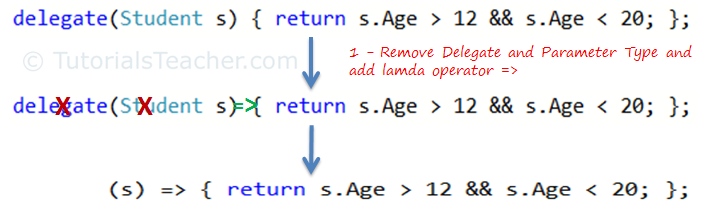
Function(s) s.Age > 12 And s.Age < 20

Let's see how the lambda expression evolved from the following anonymous method.

Example: Anonymous method in C#

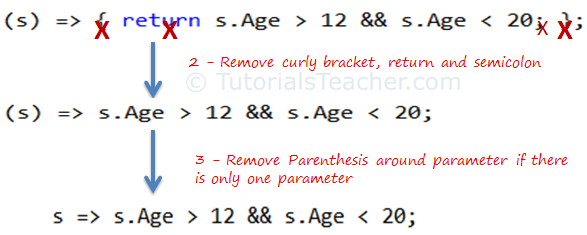
delegate(Student s) { return s.Age > 12 && s.Age < 20; };

The Lambda expression evolves from anonymous method by first removing the delegate keyword and parameter type and adding a lambda operator =>.

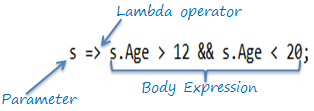
[](https://www.tutorialsteacher.com/Content/images/linq/lambda-expression-1.png)Lambda Expression from Anonymous Method

The above lambda expression is absolutely valid, but we don't need the curly braces, return and semicolon if we have only one statement that returns a value. So we can eliminate it.

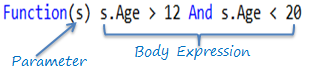
Also, we can remove parenthesis (), if we have only one parameter.

[](https://www.tutorialsteacher.com/Content/images/linq/lambda-expression-2.png)Lambda Expression from Anonymous Method

Thus, we got the lambda expression: s => s.Age > 12 && s.Age < 20 where **s** is a parameter, **=>** is the lambda operator and **s.Age > 12 && s.Age < 20** is the body expression:

[](https://www.tutorialsteacher.com/Content/images/linq/lambda-expression-structure.png)Lambda Expression Structure in C#

Same way we got lambda expression in VB.Net can be written as below:

[](https://www.tutorialsteacher.com/Content/images/linq/lambda-expression-vb.png)Lambda Expression Structure in VB.Net

The lambda expression can be invoked same way as delegate using ().

 Note:

VB.Net doesn't support lambda operator =>

## Lambda Expression with Multiple Parameters

You can wrap the parameters in parenthesis if you need to pass more than one parameter, as below:

Example: Specify Multiple Parameters in Lambda Expression C#

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

You can also give type of each parameters if parameters are confusing:

Example: Specify Parameter Type

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

Example: Specify Multiple Parameters in Lambda Expression VB.Net

Function(s, youngAge) s.Age >= youngAge

## Lambda Expression without Parameter

It is not necessary to have atleast one parameter in a lambda expression. The lambda expression can be specify without any parameter also.

Example: Lambda Expression without Parameter

**()** => Console.WriteLine("Parameter less lambda expression")

## Multiple Statements in Lambda Expression Body

You can wrap expressions in curly braces if you want to have more than one statement in the body:

Example: Multi Statements Lambda expression C#

(s, youngAge) =>

**{**

Console.WriteLine("Lambda expression with multiple statements in the body");

Return s.Age >= youngAge;

**}**

Example: Multi Statements Lambda Expression VB.Net

Function(s , youngAge)

Console.WriteLine("Lambda expression with multiple statements in the body")

Return s.Age >= youngAge

End Function

## Declare Local Variable in Lambda Expression Body

You can declare a variable in the expression body to use it anywhere in the expression body, as below:

Example: Local Variable in Lambda expression C#

s =>

{

**int youngAge = 18;**

Console.WriteLine("Lambda expression with multiple statements in the body");

return s.Age >= youngAge;

}

Example: Local Variable in Lambda Expression VB.Net

Function(s)

**Dim youngAge As Integer = 18**

Console.WriteLine("Lambda expression with multiple statements in the body")

Return s.Age >= youngAge

End Function

Lambda expression can also be assigned to built-in delegates such as [Func](https://www.tutorialsteacher.com/csharp/csharp-func-delegate), [Action](https://www.tutorialsteacher.com/csharp/csharp-action-delegate) and [Predicate](https://www.tutorialsteacher.com/csharp/csharp-predicate).

## Assign Lambda Expression to Delegate

The lambda expression can be assigned to Func<in T, out TResult> type delegate. The last parameter type in a Func delegate is the return type and rest are input parameters. Visit [Func delegate](https://www.tutorialsteacher.com/csharp/csharp-func-delegate) section of C# tutorials to know more about it.

Consider the following lambda expression to find out whether a student is a teenager or not.

Example: Lambda Expression Assigned to Func Delegate C#

Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;

Student std = new Student() { age = 21 };

bool isTeen = isStudentTeenAger(std);// returns false

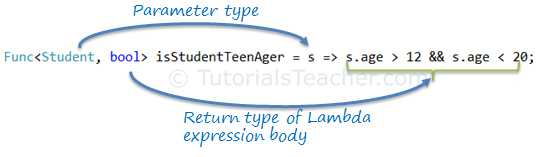
Example: Lamda Expression Assigned to Func Delegate VB.Net

Dim isStudentTeenAger As Func(Of Student, Boolean) = Function(s) s.Age > 12 And s.Age < 20

Dim stud As New Student With {.Age = 21}

Dim isTeen As Boolean = isStudentTeenAger(stud) // returns false

In the above example, the Func delegate expects the first input parameter to be of Student type and the return type to be boolean. The lambda expression s => s.age > 12 && s.age < 20 satisfies the Func<Student, bool> delegate requirement, as shown below:

[](https://www.tutorialsteacher.com/Content/images/linq/func-with-lambda-expression.png)Func delegate with Lambda Expression

The Func<> delegate shown above, would turn out to be a function as shown below.

bool isStudentTeenAger(Student s)

{

return s.Age > 12 && s.Age < 20;

}

## Action Delegate

Unlike the Func delegate, an Action delegate can only have input parameters. Use the [Action delegate](https://www.tutorialsteacher.com/csharp/csharp-action-delegate) type when you don't need to return any value from lambda expression.

Example: Lamda Expression Assigned to Action Delegate C#

Action<Student> PrintStudentDetail = s => Console.WriteLine("Name: {0}, Age: {1} ", s.StudentName, s.Age);

Student std = new Student(){ StudentName = "Bill", Age=21};

PrintStudentDetail(std);//output: Name: Bill, Age: 21

Example: Lamda Expression Assigned to Action Delegate VB.Net

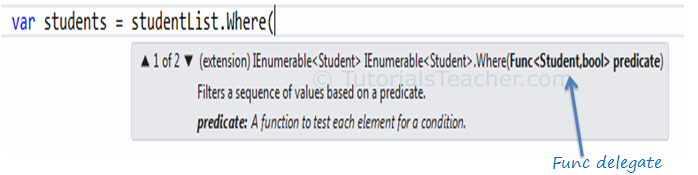
Dim printStudentDetail As Action(Of Student) = Sub(s) Console.WriteLine("Name: {0}, Age: {1} ", s.StudentName, s.Age)

Dim stud As New Student With {.StudentName = "Bill", .Age = 21}

printStudentDetail(stud)//output: Name: Bill, Age: 21

## Lambda Expression in LINQ Query

Usually lambda expression is used with LINQ query. Enumerable static class includes Where extension method for IEnumerable<T> that accepts Func<TSource,bool>. So, the Where() extension method for IEnumerable<Student> collection is required to pass Func<Student,bool>, as shown below:

[](https://www.tutorialsteacher.com/Content/images/linq/linq-where-extension-method.png)Func delegate parameter in Where extension method

So now, you can pass the lambda expression assigned to the Func delegate to the Where() extension method in the method syntax as shown below:

Example: Func Delegate in LINQ Method Syntax

IList<Student> studentList = new List<Student>(){...};

Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;

var teenStudents = studentList.Where(isStudentTeenAger).ToList<Student>();

Example: Func Delegate in LINQ Query Syntax

IList<Student> studentList = new List<Student>(){...};

Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;

var teenStudents = from s in studentList

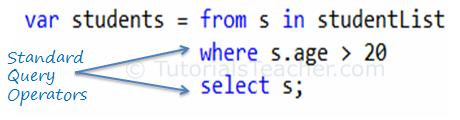
where isStudentTeenAger(s)

select s;

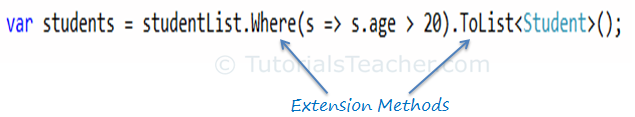
# Standard Query Operators

Standard Query Operators in LINQ are actually extension methods for the IEnumerable<T> and IQueryable<T> types. They are defined in the System.Linq.Enumerable and System.Linq.Queryable classes. There are over 50 standard query operators available in LINQ that provide different functionalities like filtering, sorting, grouping, aggregation, concatenation, etc.

## Standard Query Operators in Query Syntax

[](https://www.tutorialsteacher.com/Content/images/linq/standard-query-operators-linq-query-syntax.png)Standard Query Operators in Query Syntax

## Standard Query Operators in Method Syntax

[](https://www.tutorialsteacher.com/Content/images/linq/standard-query-operators-linq-method-syntax.png)Standard Query Operators in Method Syntax

Standard query operators in query syntax is converted into extension methods at compile time. So both are same.

Standard Query Operators can be classified based on the functionality they provide. The following table lists all the classification of Standard Query Operators:

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

##### **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.

##### **What is Linq All Operator in C#?**

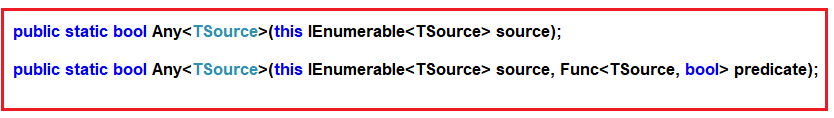
The **Linq All Operator in C#** is used to check whether all the elements of a data source satisfy a given condition or not. If all the elements satisfy the condition, then it returns true else return false. There is no overloaded version is available for the All method. The definition is given below.

* **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();**
* **}**
* **}**
* **}**
* **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; **}**
* **}**
* **}**
* 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();
* }
* }
* }

##### **What is Linq Any Operator in C#?**

The **Linq Any Operator in C#** 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 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 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 Any Operator in C#

##### **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:**

Any Operator 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

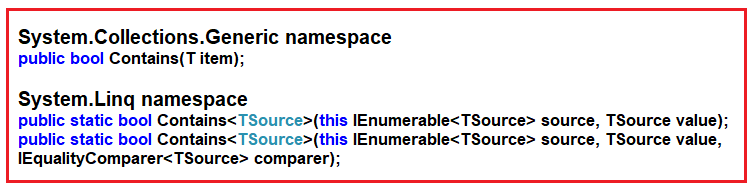
##### **Example5:**

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**()**;
* **}**
* **}**
* **}**

##### **What is Linq Contains Method 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**

##### **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:**

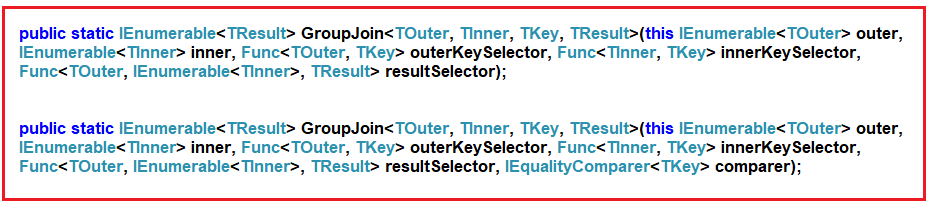
The following example uses the ToLookup Method to organize the students into groups based on Branch as the key. Here, the key will be branch and the collection will be the student belongs to that particular branch.

* **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**()**;
* **}**
* **}**
* **}**

##### **What is Linq Group Join?**

In Linq, we can apply the Group Join on two or more data sources based on a common key (the key must exist in both the data sources) and then it produces the result set in the form of groups. In simple words, we can say that Linq Group Join is used to group the result sets based on a common key.

So, the Group Join is basically used to produces hierarchical data structures. Each item from the first data source is paired with a set of correlated items from the second data source. There are two overloaded versions of this method is available as shown below.



The difference between these two methods is that the second overloaded versions take an additional IEqualityComparer. So, when working with Group Join we need to understand the following things.

1. Outer Data Source
2. Inner Data Source
3. Outer Key Selector
4. Inner Key Selector
5. Result Selector

##### **Creating Models and Data Sources:**

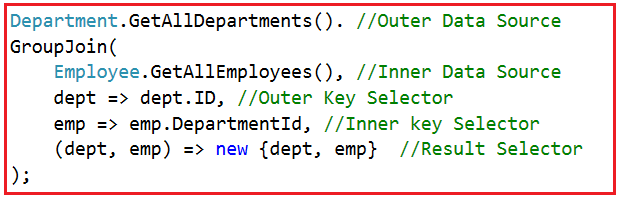
Let us understand Linq Group Join with an example. We are going to use the following Employee and Department data sources in this demo. Please create a class file and then copy and paste the following code in it.

* **using** System.Collections.Generic;
* **namespace** LINQJoin
* **{**
* **public** class Employee
* **{**
* **public** int ID **{** get; set; **}**
* **public** string Name **{** get; set; **}**
* **public** int DepartmentId **{** get; set; **}**
* **public** static List<Employee> GetAllEmployees**()**
* **{**
* **return** **new** List<Employee>**()**
* **{**
* **new** Employee **{** ID = 1, Name = "Preety", DepartmentId = 10**}**,
* **new** Employee **{** ID = 2, Name = "Priyanka", DepartmentId =20**}**,
* **new** Employee **{** ID = 3, Name = "Anurag", DepartmentId = 30**}**,
* **new** Employee **{** ID = 4, Name = "Pranaya", DepartmentId = 30**}**,
* **new** Employee **{** ID = 5, Name = "Hina", DepartmentId = 20**}**,
* **new** Employee **{** ID = 6, Name = "Sambit", DepartmentId = 10**}**,
* **new** Employee **{** ID = 7, Name = "Happy", DepartmentId = 10**}**,
* **new** Employee **{** ID = 8, Name = "Tarun", DepartmentId = 0**}**,
* **new** Employee **{** ID = 9, Name = "Santosh", DepartmentId = 10**}**,
* **new** Employee **{** ID = 10, Name = "Raja", DepartmentId = 20**}**,
* **new** Employee **{** ID = 11, Name = "Ramesh", DepartmentId = 30**}**
* **}**;
* **}**
* **}**
* **public** class Department
* **{**
* **public** int ID **{** get; set; **}**
* **public** string Name **{** get; set; **}**
* **public** static List<Department> GetAllDepartments**()**
* **{**
* **return** **new** List<Department>**()**
* **{**
* **new** Department **{** ID = 10, Name = "IT"**}**,
* **new** Department **{** ID = 20, Name = "HR"**}**,
* **new** Department **{** ID = 30, Name = "Sales" **}**,
* **}**;
* **}**
* **}**
* **}**

As you can see we created the above Employee and Department classes with some simple properties. The common property is Department ID i.e. the ID property in Department class and DepartmentID property in Employee class. Then we create two simple methods that are going to return the respective data sources. Further, if you notice the employee with ID 8 does not have a department.

##### **Example1: Using Method Syntax**

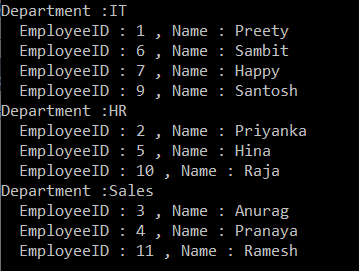
Here we need to group the employees by department. So, the outer data source is going to be the department data source and the inner data source is going to be the employee data source. Please have a look at the following image.



**The complete example is given below.**

* **using** System.Linq;
* **using** System;
* **namespace** LINQJoin
* **{**
* class Program
* **{**
* static **void** Main**(**string**[]** args**)**
* **{**
* //Group Join using Method Syntax
* var GroupJoinMS = Department.GetAllDepartments**()**.
* GroupJoin**(**
* Employee.GetAllEmployees**()**,
* dept => dept.ID,
* emp => emp.DepartmentId,
* **(**dept, emp**)** => **new** **{**dept, emp**}**
* **)**;
* //Printing the Result set
* //Outer Foreach is for all department
* **foreach(**var item in GroupJoinMS**)**
* **{**
* Console.WriteLine**(**"Department :" + item.dept.Name**)**;
* //Inner Foreach loop for each employee of a department
* **foreach(**var employee in item.emp**)**
* **{**
* Console.WriteLine**(**" EmployeeID : " + employee.ID + " , Name : " + employee.Name**)**;
* **}**
* **}**
* Console.ReadLine**()**;
* **}**
* **}**
* **}**

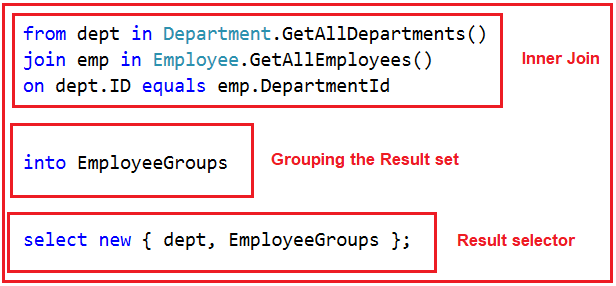
**Output:**



As you can see the employee with id 8 does not display here. This is because the employee with id 8 does not belong to any department.

##### **Example2: Using Query Syntax**

In query syntax there is no such Group Join operator is available. Here we need to use inner join along with the “into” operator. Let’s have a look at the following image.



**The complete code is given below.**

* **using** System.Linq;
* **using** System;
* **namespace** LINQJoin
* **{**
* class Program
* **{**
* static **void** Main**(**string**[]** args**)**
* **{**
* var GroupJoinQS = from dept in Department.GetAllDepartments**()**
* join emp in Employee.GetAllEmployees**()**
* on dept.ID equals emp.DepartmentId
* into EmployeeGroups
* select **new** **{** dept, EmployeeGroups **}**;
* //Outer Foreach is for all department
* **foreach(**var item in GroupJoinQS**)**
* **{**
* Console.WriteLine**(**"Department :" + item.dept.Name**)**;
* //Inner Foreach loop for each employee of a department
* **foreach(**var employee in item.EmployeeGroups**)**
* **{**
* Console.WriteLine**(**" EmployeeID : " + employee.ID + " , Name : " + employee.Name**)**;
* **}**
* **}**
* Console.ReadLine**()**;
* **}**
* **}**
* **}**

It will print the same output as the previous example.

##### **Example3: Specifying user-defined names in ResultSet**

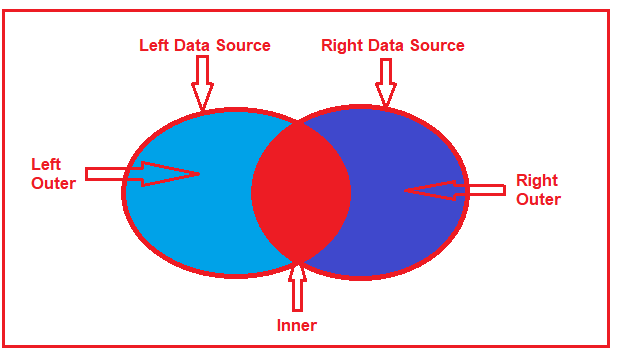
It is also possible to specify user-defined names as shown in the below example.

* **using** System.Linq;
* **using** System;
* **namespace** LINQJoin
* **{**
* class Program
* **{**
* static **void** Main**(**string**[]** args**)**
* **{**
* //Using Method Syntax
* var GroupJoinMS = Department.GetAllDepartments**()**.
* GroupJoin**(**
* Employee.GetAllEmployees**()**,
* dept => dept.ID,
* emp => emp.DepartmentId,
* //User Defined names in Result Selector
* **(**dept, emp**)** => **new{**
* Departments = dept,
* Employees = emp
* **}**
* **)**;
* //Using Query Syntax
* var GroupJoinQS = from dept in Department.GetAllDepartments**()**
* join emp in Employee.GetAllEmployees**()**
* on dept.ID equals emp.DepartmentId
* into EmployeeGroups
* //User Defined names in Result Selector
* select **new** **{**
* Departments = dept,
* Employees = EmployeeGroups
* **}**;
* **foreach(**var item in GroupJoinQS**)**
* **{**
* Console.WriteLine**(**"Department :" + item.Departments.Name**)**;
* **foreach(**var employee in item.Employees**)**
* **{**
* Console.WriteLine**(**" EmployeeID : " + employee.ID + " , Name : " + employee.Name**)**;
* **}**
* **}**
* Console.ReadLine**()**;
* **}**
* **}**
* **}**

In the next article, I am going to discuss the **[Left Outer Join in Linq](https://dotnettutorials.net/lesson/left-outer-join-in-linq/)** with some examples. In this article, I try to explain the **Linq Group Join using both Method and Query Syntax** example in C#.

##### **What is Left Outer 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 Left Outer Join in Linq, 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"**}**,
* **}**;
* **}**
* **}**
* **}**

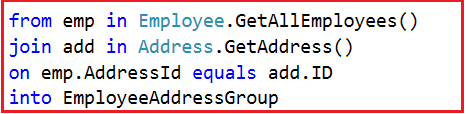
As you can see, here we also created two methods which will be going to return the respective Employees and Addresses which are going to be our data source. Here we hard-coded the data sources but in real-time you will get the data from a database. If you further notice we have two employees with address 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.

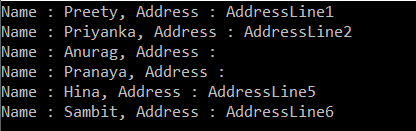
DefaultIfEmplty in Outer Join

**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.

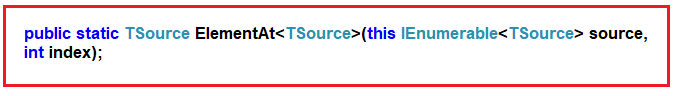
* **using** System.Linq;
* **using** System;
* **namespace** LINQJoin
* **{**
* class Program
* **{**
* static **void** Main**(**string**[]** args**)**
* **{**
* //Using Method Syntax
* var MSOuterJOIN = Employee.GetAllEmployees**()**
* .GroupJoin**(**
* Address.GetAddress**()**,
* emp => emp.AddressId,
* 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.

In the next article, I am going to discuss the **[Linq Cross Join](https://dotnettutorials.net/lesson/linq-cross-join/)** with some examples. In this article, I try to explain how to implement **Left Outer Join in Linq** using Method syntax and Query Syntax.

##### **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**()**;
* **}**
* **}**
* **}**

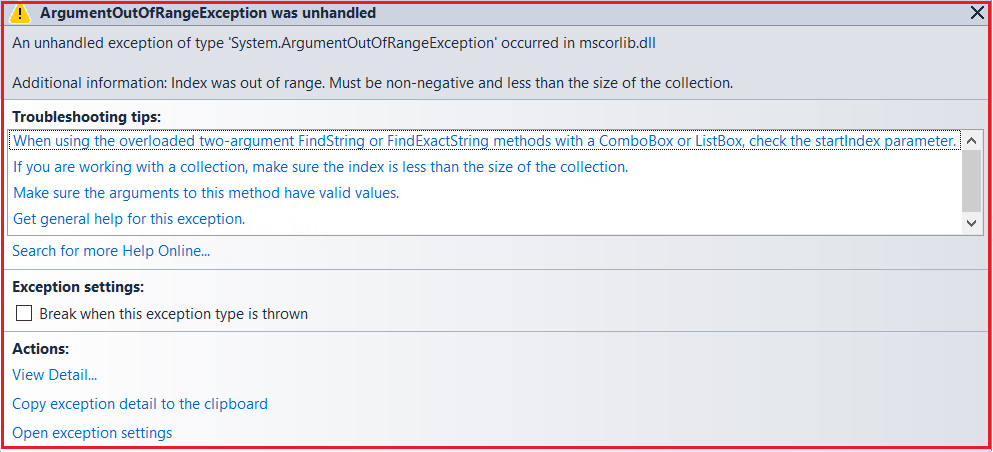
Run the application and it should print 2 in the output window as in index position 1 the value 2 is there.

##### **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**()**;
* **}**
* **}**
* **}**

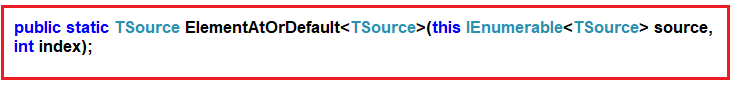
Again you will get the same exception as the previous example.

**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.

In the next article, I am going to discuss the **[First and FirstOrDefault method in Linq](https://dotnettutorials.net/lesson/first-and-firstordefault-methods-in-linq/)** with some examples. In this article, I try to explain the **Linq ElementAt and ElementAtOrDefault method in C#** with some examples. I hope you understood the need and use of these two methods in Linq.