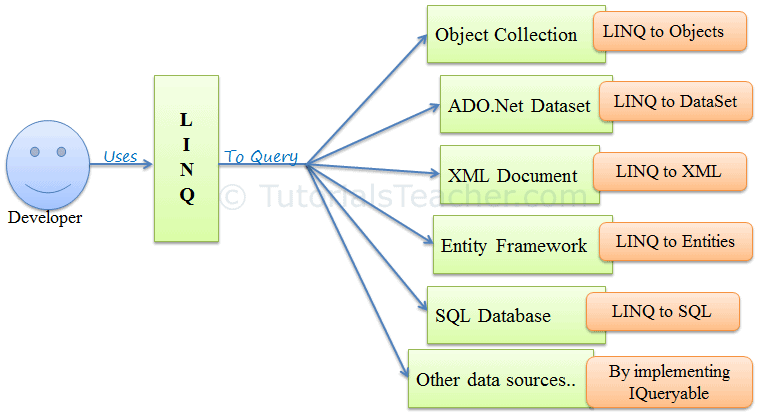
LINQ Tutorial

Language-Integrated Query (LINQ) is a powerful query language introduced with .Net 3.5 & Visual Studio 2008. LINQ can be used with C# or Visual Basic to query different data sources.

What is LINQ?

LINQ (Language Integrated Query) is **uniform query syntax** in C# and VB.NET to retrieve data from different sources and formats. It is integrated in C# or VB, thereby eliminating the mismatch between programming languages and databases, as well as providing a single querying interface for different types of data sources.

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 of data sources such as collections, ADO.Net DataSet VB.NET to retrieve data from different types, XML Docs, web service and MS SQL Server and other databases.

[](https://www.tutorialsteacher.com/Content/images/linq/linq-usage.PNG)LINQ Usage

LINQ queries return results as objects. It enables you to uses object-oriented approach on the result set and not to worry about transforming different formats of results into objects.

[](https://www.tutorialsteacher.com/Content/images/linq/linq-execution.PNG)

The following example demonstrates a simple LINQ query that gets all strings from an array which contains 'a'.

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 + " ");

In the above example, string array names is a data source. The following is a LINQ query which is assigned to a variable myLinqQuery.

from name in names

where name.Contains('a')

select name;

The above query uses query syntax of LINQ. You will learn more about it in the [Query Syntax](https://www.tutorialsteacher.com/linq/linq-query-syntax) chapter.

You will not get the result of a LINQ query until you execute it. LINQ query can be executed in multiple ways, here we used foreach loop to execute our query stored in myLinqQuery. The foreach loop executes the query on the data source and get the result and then iterates over the result set.

Thus, every LINQ query must query to some kind of data sources whether it can be array, collections, XML or other databases. After writing LINQ query, it must be executed to get the result.

# 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++;

Console.WriteLine(std.StudentID + " " + std.StudentName + " " + std.Age)

}

}

}

}

Use of for loop is cumbersome, not maintainable and readable. C# 2.0 introduced [delegate](https://www.tutorialsteacher.com/csharp/csharp-delegates), which can be used to handle this kind of a scenario, as shown below.

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";

});

ADVERTISEMENT

The 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();

}

}

As you can see in the above example, we specify different criteria using LINQ operator and lambda expression in a single statement. Thus, LINQ makes code more compact and readable and it can also be used to query different data sources. For example, if you have a student table in a database instead of an array of student objects as above, you can still use the same query to find students using the [Entity Framework](http://www.entityframeworktutorial.net/).

## **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.

# 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;

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-AFTZoc)

The following figure shows the structure of LINQ query syntax.

[Graphical user interface, text, application

Description automatically generated](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;

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-RCgTqH)

Example: LINQ Query 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 Query Syntax to find out teenager students

Dim teenAgerStudents As IList(Of Student) = (From s In studentList \_

Where s.Age > 12 And s.Age < 20 \_

Select s).ToList()

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-Vh8aF0)

 Points to Remember :

1. As name suggest, **Query Syntax** is same like SQL (Structure Query Language) syntax.
2. Query Syntax starts with *from* clause and can be end with *Select* or *GroupBy* clause.
3. Use various other opertors like filtering, joining, grouping, sorting operators to construct the desired result.
4. [Implicitly typed variable - var](https://www.tutorialsteacher.com/csharp/csharp-var-implicit-typed-local-variable) can be used to hold the result of the LINQ query.

LINQ Method Syntax

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

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" \t "_blank) static class, similar to how you would call the extension method of any class.

The compiler converts query syntax into method syntax at compile time.

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.

[Diagram

Description automatically generated with low confidence](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.

[Graphical user interface, text, application

Description automatically generated](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() Sonic { 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>();

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-bG7zlJ)

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()

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-EEasTZ)

 Points to Remember :

1. As name suggest, **Method Syntax** is like calling extension method.
2. LINQ **Method Syntax** aka Fluent syntax because it allows series of extension methods call.
3. Implicitly typed variable - var can be used to hold the result of the LINQ query.

# 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; };

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-HabxnO)

Example: Anonymous method in VB.Net

Dim isStudentTeenAger = Function(s As Student) As Boolean

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

End Function

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-77xAMs)

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

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-rqSYnO)

Example: Lambda Expression in VB.Net

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

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-R13yq1)

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 =>.

[A picture containing text

Description automatically generated](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.

[Text

Description automatically generated](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:

[Diagram

Description automatically generated with low confidence](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:

[Graphical user interface, text

Description automatically generated](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 =>

ADVERTISEMENT

## **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;

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-PWGyET)

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

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-52ROJw)

## **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

**///**

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-1JQVyo)

## **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;

**}**

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-oGZEjZ)

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

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-HA4Mqu)

## **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;

}

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-S1K7xU)

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

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-p76aBt)

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).

ADVERTISEMENT

## **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" \t "_blank) 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

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-MfwAY6)

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

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-dh5cUY)

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:

[Graphical user interface, text

Description automatically generated](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

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-TilB9f)

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

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-nCWoaG)

## **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:

[Graphical user interface, text, application

Description automatically generated](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>();

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-tCgBt7)

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;

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-VmWuAS)

You can follow the same method in VB.Net to pass Func delegate.

 Points to Remember :

1. Lambda Expression is a shorter way of representing anonymous method.
2. Lambda Expression syntax: *parameters => body expression*
3. Lambda Expression can have zero parameter.
4. Lambda Expression can have multiple parameters in parenthesis ().
5. Lambda Expression can have multiple statements in body expression in curly brackets {}.
6. Lambda Expression can be assigned to Func, Action or Predicate delegate.
7. Lambda Expression can be invoked in a similar way to delegate.

# 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**

[Graphical user interface, text

Description automatically generated](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**

[Graphical user interface, text

Description automatically generated](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 |

# Filtering Operator - Where

Filtering operators in LINQ filter the sequence (collection) based on some given criteria.

The following table lists all the filtering operators available in LINQ.

| Filtering Operators | Description |
| --- | --- |
| [Where](https://www.tutorialsteacher.com/linq/linq-filtering-operators-where#where) | Returns values from the collection based on a predicate function. |
| [OfType](https://www.tutorialsteacher.com/linq/linq-filtering-operators-oftype) | Returns values from the collection based on a specified type. However, it will depend on their ability to cast to a specified type. |

## **Where**

The Where operator (Linq extension method) filters the collection based on a given criteria expression and returns a new collection. The criteria can be specified as lambda expression or Func delegate type.

The **Where** extension method has following two overloads. Both overload methods accepts a [Func delegate](https://www.tutorialsteacher.com/csharp/csharp-func-delegate) type parameter. One overload required Func<TSource,bool> input parameter and second overload method required Func<TSource, int, bool> input parameter where int is for index:

Where method Overloads:

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,

Func<TSource, bool> predicate);

public static IEnumerable<TSource> Where<TSource>(this IEnumerable<TSource> source,

Func<TSource, int, bool> predicate);

### **Where clause in Query Syntax**

The following query sample uses a Where operator to filter the students who is teen ager from the given collection (sequence). It uses a lambda expression as a predicate function.

Example: Where clause - LINQ query syntax C#

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 }

};

var filteredResult = from s in studentList

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

select s.StudentName;

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-RCgTqH)

Example: Where clause - LINQ query syntax in VB.Net

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}

}

Dim filteredResult = From s In studentList

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

Select s.StudentName

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-Vh8aF0)

In the above example, filteredResult will include following students after query execution.

John  
Bill  
Ron

In the above sample query, the lambda expression body **s.Age > 12 && s.Age < 20** is passed as a predicate function **Func<TSource, bool>** that evaluates every student in the collection.

Alternatively, you can also use a Func type delegate with an anonymous method to pass as a predicate function as below (output would be the same):

Example: Where clause

Func<Student,bool> isTeenAger = delegate(Student s) {

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

};

var filteredResult = from s in studentList

where isTeenAger(s)

select s;

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-26pUZE)

You can also call any method that matches with Func parameter with one of Where() method overloads.

Example: Where clause

public static void Main()

{

var filteredResult = from s in studentList

where isTeenAger(s)

select s;

}

public static bool IsTeenAger(Student stud)

{

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

}

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-Vh8aF0)

ADVERTISEMENT

### **Where extension method in Method Syntax**

Unlike the query syntax, you need to pass whole lambda expression as a predicate function instead of just body expression in LINQ method syntax.

Example: Where in method syntax in C#

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

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-bG7zlJ)

Example: Where in method syntax in VB.Net

Dim filteredResult = studentList.Where(Function(s) s.Age > 12 And s.Age < 20 )

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-EEasTZ)

As mentioned above, the **Where** extension method also have second overload that includes index of current element in the collection. You can use that index in your logic if you need.

The following example uses the Where clause to filter out odd elements in the collection and return only even elements. Please remember that index starts from zero.

Example: Linq - Where extension method in C#

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

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

new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,

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

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

new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

};

var filteredResult = studentList.Where((s, i) => {

if(i % 2 == 0) // if it is even element

return true;

return false;

});

foreach (var std in filteredResult)

Console.WriteLine(std.StudentName);

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-W5ezzZ)

Output:

John  
Bill  
Ron

### **Multiple Where clause**

You can call the Where() extension method more than one time in a single LINQ query.

Example: Multiple where clause in Query Syntax C#

var filteredResult = from s in studentList

where s.Age > 12

where s.Age < 20

select s;

Example: Multiple where clause in Method Syntax C#

var filteredResult = studentList.Where(s => s.Age > 12).Where(s => s.Age < 20);

 Points to Remember :

1. **Where** is used for filtering the collection based on given criteria.
2. Where extension method has two overload methods. Use a second overload method to know the index of current element in the collection.
3. Method Syntax requires the whole lambda expression in Where extension method whereas Query syntax requires only expression body.
4. Multiple **Where** extension methods are valid in a single LINQ query.

# Filtering Operator - OfType

The OfType operator filters the collection based on the ability to cast an element in a collection to a specified type.

### **OfType in Query Syntax**

Use OfType operator to filter the above collection based on each element's type

Example: OfType operator in C#

IList mixedList = new ArrayList();

mixedList.Add(0);

mixedList.Add("One");

mixedList.Add("Two");

mixedList.Add(3);

mixedList.Add(new Student() { StudentID = 1, StudentName = "Bill" });

var stringResult = from s in mixedList.OfType<string>()

select s;

var intResult = from s in mixedList.OfType<int>()

select s;

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-VVUIZs)

Example: OfType operator in VB.Net:

Dim stringResult = From s In mixedList.OfType(Of String)()

The above sample queries will return items whose type is string in the mixedList. stringResult contains following elements after execution:

One  
Two  
0  
3  
Bill

### **OfType in Method Syntax**

You can use OfType<TResult>() extension method in linq method syntax as shown below.

Example: OfType in C#

var stringResult = mixedList.OfType<string>();

Example: OfType in VB.Net

Dim stringResult = mixedList.OfType(Of String)

stringResult would contain following elements.

One  
Two

Visit MSDN for more information on [Filtering operators](https://msdn.microsoft.com/en-us/library/bb546161.aspx).

 Points to Remember :

1. The **Where** operator filters the collection based on a predicate function.
2. The **OfType** operator filters the collection based on a given type
3. **Where** and **OfType** extension methods can be called multiple times in a single LINQ query.

# Sorting Operators: OrderBy & OrderByDescending

A sorting operator arranges the elements of the collection in ascending or descending order. LINQ includes following sorting operators.

| Sorting Operator | Description |
| --- | --- |
| [OrderBy](https://www.tutorialsteacher.com/linq/linq-sorting-operators-orderby-orderbydescending#orderby) | Sorts the elements in the collection based on specified fields in ascending or decending order. |
| [OrderByDescending](https://www.tutorialsteacher.com/linq/linq-sorting-operators-orderby-orderbydescending#orderbydescending) | Sorts the collection based on specified fields in descending order. Only valid in method syntax. |
| [ThenBy](https://www.tutorialsteacher.com/linq/linq-sorting-operators-thenby-thenbydescending) | Only valid in method syntax. Used for second level sorting in ascending order. |
| [ThenByDescending](https://www.tutorialsteacher.com/linq/linq-sorting-operators-thenby-thenbydescending) | Only valid in method syntax. Used for second level sorting in descending order. |
| Reverse | Only valid in method syntax. Sorts the collection in reverse order. |

## **OrderBy**

OrderBy sorts the values of a collection in ascending or descending order. It sorts the collection in ascending order by default because ascending keyword is optional here. Use descending keyword to sort collection in descending order.

Example: OrderBy in Query Syntax C#

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

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

new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,

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

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

new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

};

var orderByResult = from s in studentList

orderby s.StudentName

select s;

var orderByDescendingResult = from s in studentList

orderby s.StudentName descending

select s;

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-f5ywFg)

Example: OrderBy in Query Syntax VB.Net

Dim orderByResult = From s In studentList

Order By s.StudentName

Select s

Dim orderByDescendingResult = From s In studentList

Order By s.StudentName Descending

Select s

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-Zef7G3)

orderByResult in the above example would contain following elements after execution:

Bill  
John  
Ram  
Ron  
Steve

orderByDescendingResult in the above example would contain following elements after execution:

Steve  
Ron  
Ram  
John  
Bill

## **OrderBy in Method Syntax**

OrderBy extension method has two overloads. First overload of OrderBy extension method accepts the Func delegate type parameter. So you need to pass the lambda expression for the field based on which you want to sort the collection.

The second overload method of OrderBy accepts object of IComparer along with Func delegate type to use custom comparison for sorting.

OrderBy Overload Methods:

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector);

public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(this IEnumerable<TSource> source,

Func<TSource, TKey> keySelector,

IComparer<TKey> comparer);

The following example sorts the studentList collection in ascending order of StudentName using OrderBy extension method.

Example: OrderBy in Method Syntax C#

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

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

new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,

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

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

new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

};

var studentsInAscOrder = studentList.OrderBy(s => s.StudentName);

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-zJ5SmO)

Example: OrderBy in Method Syntax VB.Net

Dim studentsInAscOrder = studentList.OrderBy(Function(s) s.StudentName)

 Note:

Method syntax does not allow the decending keyword to sorts the collection in decending order. Use OrderByDecending() method for it.

ADVERTISEMENT

## **OrderByDescending**

OrderByDescending sorts the collection in descending order.

OrderByDescending is valid only with the Method syntax. It is not valid in query syntax because the query syntax uses ascending and descending attributes as shown above.

Example: OrderByDescending C#

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

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

new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,

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

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

new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }

};

var studentsInDescOrder = studentList.OrderByDescending(s => s.StudentName);

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-zJ5SmO)

Example: OrderByDescending VB.Net

Dim studentsInDescOrder = studentList.OrderByDescending(Function(s) s.StudentName)

A result in the above example would contain following elements after execution.

Steve  
Ron  
Ram  
John  
Bill

Please note that OrderByDescending is not supported in query syntax. Use the decending keyword instead.

## **Multiple Sorting**

You can sort the collection on multiple fields seperated by comma. The given collection would be first sorted based on the first field and then if value of first field would be the same for two elements then it would use second field for sorting and so on.

Example: Multiple sorting in Query syntax C#

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

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

new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,

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

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

new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 },

new Student() { StudentID = 6, StudentName = "Ram" , Age = 18 }

};

var orderByResult = from s in studentList

orderby s.StudentName, s.Age

select new { s.StudentName, s.Age };

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-0KqpNz)

In the above example, studentList collection includes two identical StudentNames, Ram. So now, studentList would be first sorted based on StudentName and then by Age in ascending order. So, orderByResult would contain following elements after execution

StudentName: Bill, Age: 25  
StudentName: John, Age: 18  
StudentName: Ram, Age: 18  
StudentName: Ram, Age: 20  
StudentName: Ron, Age: 19  
StudentName: Steve, Age: 15

 Note:

Multiple sorting in method syntax works differently. Use ThenBy or ThenByDecending extension methods for secondary sorting.

 Points to Remember :

1. LINQ includes five sorting operators: OrderBy, OrderByDescending, ThenBy, ThenByDescending and Reverse
2. LINQ query syntax does not support OrderByDescending, ThenBy, ThenByDescending and Reverse. It only supports 'Order By' clause with 'ascending' and 'descending' sorting direction.
3. LINQ query syntax supports multiple sorting fields seperated by comma whereas you have to use ThenBy & ThenByDescending methods for secondary sorting.

Sorting Operators: ThenBy & ThenByDescending

The ThenBy and ThenByDescending extension methods are used for sorting on multiple fields.

The OrderBy() method sorts the collection in ascending order based on specified field. Use ThenBy() method after OrderBy to sort the collection on another field in ascending order. Linq will first sort the collection based on primary field which is specified by OrderBy method and then sort the resulted collection in ascending order again based on secondary field specified by ThenBy method.

The same way, use ThenByDescending method to apply secondary sorting in descending order.

The following example shows how to use ThenBy and ThenByDescending method for second level sorting:

Example: ThenBy & ThenByDescending

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

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

new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,

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

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

new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 },

new Student() { StudentID = 6, StudentName = "Ram" , Age = 18 }

};

var thenByResult = studentList.OrderBy(s => s.StudentName).ThenBy(s => s.Age);

var thenByDescResult = studentList.OrderBy(s => s.StudentName).ThenByDescending(s => s.Age);

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-tEbcy8)

As you can see in the above example, we first sort a studentList collection by StudentName and then by Age. So now, thenByResult would contain following elements after sorting:

StudentName: Bill, Age: 25  
StudentName: John, Age: 18  
StudentName: Ram, Age: 18  
StudentName: Ram, Age: 20  
StudentName: Ron, Age: 19  
StudentName: Steve, Age: 15

thenByDescResult would contain following elements. Please notice that Ram with age 20 comes before Ram with age 18 because it has used ThenByDescending.

StudentName: Bill, Age: 25  
StudentName: John, Age: 18  
StudentName: Ram, Age: 20  
StudentName: Ram, Age: 18  
StudentName: Ron, Age: 19  
StudentName: Steve, Age: 15

You can use ThenBy and ThenByDescending method same way in VB.Net as below:

Example: ThenBy & ThenByDescending VB.Net

Dim sortedResult = studentList.OrderBy(Function(s) s.StudentName)

.ThenBy(Function(s) s.Age)

Dim sortedResult = studentList.OrderBy(Function(s) s.StudentName)

.ThenByDescending(Function(s) s.Age)

 Points to Remember :

1. OrderBy and ThenBy sorts collections in ascending order by default.
2. ThenBy or ThenByDescending is used for second level sorting in method syntax.
3. ThenByDescending method sorts the collection in decending order on another field.
4. ThenBy or ThenByDescending is NOT applicable in Query syntax.
5. Apply secondary sorting in query syntax by separating fields using comma.

# Grouping Operators: GroupBy & ToLookup

The grouping operators do the same thing as the GroupBy clause of SQL query. The grouping operators create a group of elements based on the given key. This group is contained in a special type of collection that implements an IGrouping<TKey,TSource> interface where TKey is a key value, on which the group has been formed and TSource is the collection of elements that matches with the grouping key value.

| Grouping Operators | Description |
| --- | --- |
| [GroupBy](https://www.tutorialsteacher.com/linq/linq-grouping-operator-groupby-tolookup#groupby) | The GroupBy operator returns groups of elements based on some key value. Each group is represented by IGrouping<TKey, TElement> object. |
| [ToLookup](https://www.tutorialsteacher.com/linq/linq-grouping-operator-groupby-tolookup#tolookup) | ToLookup is the same as GroupBy; the only difference is the execution of GroupBy is deferred whereas ToLookup execution is immediate. |

## **GroupBy**

The GroupBy operator returns a group of elements from the given collection based on some key value. Each group is represented by IGrouping<TKey, TElement> object. Also, the GroupBy method has eight overload methods, so you can use appropriate extension method based on your requirement in method syntax.

 Note:

A LINQ query can end with a GroupBy or Select clause.

The result of GroupBy operators is a collection of groups. For example, GroupBy returns IEnumerable<IGrouping<TKey,Student>> from the Student collection:

[Graphical user interface, text, application

Description automatically generated](https://www.tutorialsteacher.com/Content/images/linq/linq-groupby.png)Return type of GroupBy()

### **GroupBy in Query Syntax**

The following example creates a groups of students who have same age. Students of the same age will be in the same collection and each grouped collection will have a key and inner collection, where the key will be the age and the inner collection will include students whose age is matched with a key.

Example: GroupBy in Query syntax C#

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

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

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

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

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

new Student() { StudentID = 5, StudentName = "Abram" , Age = 21 }

};

var groupedResult = from s in studentList

group s by s.Age;

//iterate each group

foreach (var ageGroup in groupedResult)

{

Console.WriteLine("Age Group: {0}", ageGroup .Key); //Each group has a key

foreach(Student s in ageGroup) // Each group has inner collection

Console.WriteLine("Student Name: {0}", s.StudentName);

}

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-BZnxR2)

Output:

AgeGroup: 18  
StudentName: John  
StudentName: Bill  
AgeGroup: 21  
StudentName: Steve  
StudentName: Abram  
AgeGroup: 20  
StudentName: Ram

As you can see in the above example, you can iterate the group using a 'foreach' loop, where each group contains a key and inner collection. The following figure shows the result in debug view.

[Graphical user interface, text, application, chat or text message

Description automatically generated](https://www.tutorialsteacher.com/Content/images/linq/linq-groupby-2.png)Grouped collection with key and inner collection

Use "Into Group" with the 'Group By' clause in VB.Net as shown below.

Example: GroupBy clause in VB.Net

Dim groupQuery = From s In studentList

Group By s.Age Into Group

For Each group In groupQuery

Console.WriteLine("Age Group: {0}", group.Age) // Each group has key property name

For Each student In group.Group // Each group has inner collection

Console.WriteLine("Student Name: {0}", student.StudentName)

Next

Next

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-cCMaRM)

Notice that each group will have a property name on which group is performed. In the above example, we have used Age to form a group so each group will have "Age" property name instead of "Key" as a property name.

Output:

AgeGroup: 18  
StudentName: John  
StudentName: Bill  
AgeGroup: 21  
StudentName: Steve  
StudentName: Abram  
AgeGroup: 20  
StudentName: Ram

ADVERTISEMENT

## **GroupBy in Method Syntax**

The GroupBy() extension method works the same way in the method syntax. Specify the lambda expression for key selector field name in GroupBy extension method.

Example: GroupBy in method syntax C#

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

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

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

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

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

new Student() { StudentID = 5, StudentName = "Abram" , Age = 21 }

};

var groupedResult = studentList.GroupBy(s => s.Age);

foreach (var ageGroup in groupedResult)

{

Console.WriteLine("Age Group: {0}", ageGroup.Key); //Each group has a key

foreach(Student s in ageGroup) //Each group has a inner collection

Console.WriteLine("Student Name: {0}", s.StudentName);

}

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-9yg2ep)

Example: GroupBy in method syntax VB.Net

Dim groupQuery = studentList.GroupBy(Function(s) s.Age)

For Each ageGroup In groupQuery

Console.WriteLine("Age Group: {0}", ageGroup.Key) //Each group has a key

For Each student In ageGroup.AsEnumerable() //Each group has a inner collection

Console.WriteLine("Student Name: {0}", student.StudentName)

Next

Next

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-lNVo31)

Output:

AgeGroup: 18  
StudentName: John  
StudentName: Bill  
AgeGroup: 21  
StudentName: Steve  
StudentName: Abram  
AgeGroup: 20  
StudentName: Ram

## **ToLookup**

ToLookup is the same as GroupBy; the only difference is GroupBy execution is deferred, whereas ToLookup execution is immediate. Also, ToLookup is only applicable in Method syntax. **ToLookup is not supported in the query syntax.**

Example: ToLookup in method syntax C#

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

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

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

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

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

new Student() { StudentID = 5, StudentName = "Abram" , Age = 21 }

};

var lookupResult = studentList.ToLookup(s => s.age);

foreach (var group in lookupResult)

{

Console.WriteLine("Age Group: {0}", group.Key); //Each group has a key

foreach(Student s in group) //Each group has a inner collection

Console.WriteLine("Student Name: {0}", s.StudentName);

}

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-ZqU8cf)

Example: ToLookup in method syntax VB.Net

Dim loopupResult = studentList.ToLookup(Function(s) s.Age)

 Points to Remember :

1. GroupBy & ToLookup return a collection that has a key and an inner collection based on a key field value.
2. The execution of GroupBy is deferred whereas that of ToLookup is immediate.
3. A LINQ query syntax can be end with the GroupBy or Select clause.

# Joining Operator: Join

The joining operators joins the two sequences (collections) and produce a result.

| Joining Operators | Usage |
| --- | --- |
| [Join](https://www.tutorialsteacher.com/linq/linq-joining-operator-join#join) | The Join operator joins two sequences (collections) based on a key and returns a resulted sequence. |
| [GroupJoin](https://www.tutorialsteacher.com/linq/linq-joining-operator-groupjoin) | The GroupJoin operator joins two sequences based on keys and returns groups of sequences. It is like Left Outer Join of SQL. |

## **Join**

The Join operator operates on two collections, inner collection & outer collection. It returns a new collection that contains elements from both the collections which satisfies specified expression. It is the same as **inner join** of SQL.

### **Join in Method Syntax**

The Join extension method has two overloads as shown below.

Join Overload Methods:

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer,

IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector,

Func<TOuter, TInner, TResult> resultSelector);

public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer,

IEnumerable<TInner> inner,

Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector,

Func<TOuter, TInner, TResult> resultSelector,

IEqualityComparer<TKey> comparer);

As you can see in the first overload method takes five input parameters (except the first 'this' parameter): 1) outer 2) inner 3) outerKeySelector 4) innerKeySelector 5) resultSelector.

Let's take a simple example. The following example joins two string collection and return new collection that includes matching strings in both the collection.

Example: Join operator C#

IList<string> strList1 = new List<string>() {

"One",

"Two",

"Three",

"Four"

};

IList<string> strList2 = new List<string>() {

"One",

"Two",

"Five",

"Six"

};

var innerJoin = strList1.Join(strList2,

str1 => str1,

str2 => str2,

(str1, str2) => str1);

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-DoAglF)

One  
Two

Now, let's understand join metohod using following Student and Standard class where Student class includes StandardID that matches with StandardID of Standard class.

Example Classes

public class Student{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int StandardID { get; set; }

}

public class Standard{

public int StandardID { get; set; }

public string StandardName { get; set; }

}

The following example demonstrates LINQ Join query.

Example: Join Query C#

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

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

new Student() { StudentID = 2, StudentName = "Moin", StandardID =1 },

new Student() { StudentID = 3, StudentName = "Bill", StandardID =2 },

new Student() { StudentID = 4, StudentName = "Ram" , StandardID =2 },

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

};

IList<Standard> standardList = new List<Standard>() {

new Standard(){ StandardID = 1, StandardName="Standard 1"},

new Standard(){ StandardID = 2, StandardName="Standard 2"},

new Standard(){ StandardID = 3, StandardName="Standard 3"}

};

var innerJoin = studentList.Join(// outer sequence

standardList, // inner sequence

student => student.StandardID, // outerKeySelector

standard => standard.StandardID, // innerKeySelector

(student, standard) => new // result selector

{

StudentName = student.StudentName,

StandardName = standard.StandardName

});

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-8nUDoV)

The following image illustrate the parts of Join operator in the above example.

[Text

Description automatically generated](https://www.tutorialsteacher.com/Content/images/linq/linq-join-operator.png)join operator

In the above example of join query, studentList is outer sequence because query starts from it. First parameter in Join method is used to specify the inner sequence which is standardList in the above example. Second and third parameter of Join method is used to specify a field whose value should be match using lambda expression in order to include element in the result. The key selector for the outer sequence student => student.StandardID indicates that take StandardID field of each elements of studentList should be match with the key of inner sequence standard => standard.StandardID. If value of both the key field is matched then include that element into result.

The last parameter in Join method is an expression to formulate the result. In the above example, result selector includes StudentName and StandardName property of both the sequence.

ADVERTISEMENT

StandardID Key of both the sequences (collections) must match otherwise the item will not be included in the result. For example, Ron is not associated with any standard so Ron is not included in the result collection. innerJoinResult in the above example would contain following elements after execution:

John - Standard 1  
Moin - Standard 1  
Bill - Standard 2  
Ram - Standard 2

The following example demonstrates the Join operator in method syntax in VB.Net.

Example: Join operator VB.Net

Dim innerJoin = studentList.Join(standardList,

Function(s) s.StandardID,

Function(std) std.StandardID,

Function(s, std) New With

{

.StudentName = s.StudentName,

.StandardName = std.StandardName

});

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-4hlooi)

### **Join in Query Syntax**

Join operator in query syntax works slightly different than method syntax. It requires outer sequence, inner sequence, key selector and result selector. 'on' keyword is used for key selector where left side of 'equals' operator is outerKeySelector and right side of 'equals' is innerKeySelector.

Syntax: Join in query syntax

from ... in outerSequence

join ... in innerSequence

on outerKey equals innerKey

select ...

The following example of Join operator in query syntax returns a collection of elements from studentList and standardList if their Student.StandardID and Standard.StandardID is match.

Example: Join operator in query syntax C#

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

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

new Student() { StudentID = 2, StudentName = "Moin", Age = 21, StandardID =1 },

new Student() { StudentID = 3, StudentName = "Bill", Age = 18, StandardID =2 },

new Student() { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID =2 },

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

};

IList<Standard> standardList = new List<Standard>() {

new Standard(){ StandardID = 1, StandardName="Standard 1"},

new Standard(){ StandardID = 2, StandardName="Standard 2"},

new Standard(){ StandardID = 3, StandardName="Standard 3"}

};

var innerJoin = from s in studentList // outer sequence

join st in standardList //inner sequence

on s.StandardID equals st.StandardID // key selector

select new { // result selector

StudentName = s.StudentName,

StandardName = st.StandardName

};

[Try it](https://www.tutorialsteacher.com/codeeditor?cid=cs-4arutq)

Example: Join operator in query syntax VB.Net

Dim innerJoin = From s In studentList ' outer sequence

Join std In standardList ' inner sequence

On s.StandardID Equals std.StandardID ' key selector

Select \_ ' result selector

StudentName = s.StudentName,

StandardName = std.StandardName

Output:

John - Standard 1  
Moin - Standard 1  
Bill - Standard 2  
Ram - Standard 2

 Note:

Use the **equals** operator to match key selector in query syntax. == is not valid.

 Points to Remember :

1. **Join** and **GroupJoin** are joining operators.
2. **Join** is like inner join of SQL. It returns a new collection that contains common elements from two collections whosh keys matches.
3. **Join** operates on two sequences inner sequence and outer sequence and produces a result sequence.
4. **Join** query syntax:
5. from... in outerSequence
6. join... in innerSequence
7. on outerKey equals innerKey

select ...