LINQ Tutorials

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.

LINQ tutorials will help you to learn the LINQ language using topics which go from basic to advanced. These tutorials are broken down into series of related topics, so that you start from a topic which must be understand first, and then gradually learn other features of LINQ sequentially. LINQ tutorials are packed with easy to understand explanations, real-world examples, useful tips, informative notes and points to remember.

For Whom?

These tutorials are designed for beginners and professionals who want to learn LINQ step-by-step.

Prerequisites:

Basic knowledge of .Net Framework3.5/4.5, C#, Visual Studio is required.

Quiz

Test your LINQ knowledge - [Start LINQ Test](http://www.tutorialsteacher.com/online-test/linq-test)

So let's get started by clicking Next.

What is LINQ?

LINQ (Language Integrated Query) is uniform query syntax in C# and VB.NET used to save and retrieve data from different sources. 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 VB.NET used to save and retrieve data from different types of data sources like an Object Collection, SQL server database, XML, web service etc.

LINQ always works with objects so you can use the same basic coding patterns to query and transform data in XML documents, SQL databases, ADO.NET Datasets, .NET collections, and any other format for which a LINQ provider is available.

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

LINQ Usage

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++;

}

}

}

}

Use of for loop is cumbersome, not maintainable and readable. C# 2.0 introduced **delegate**, 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";

});

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](http://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.

C# 3.0 onwards:

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 API

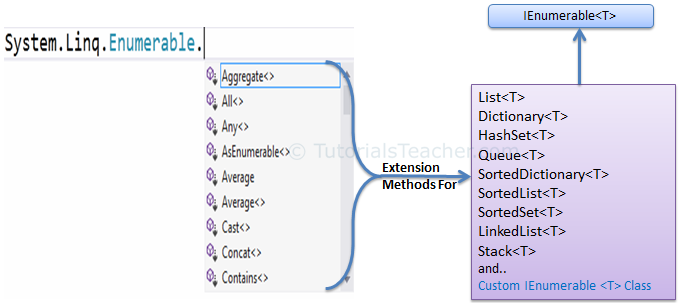
LINQ is nothing but the collection of extension methods for classes that implements IEnumerable and IQueryable interface. [*System.Linq*](http://msdn.microsoft.com/en-us/library/system.linq(v=vs.110).aspx) namespace includes the necessary classes & interfaces for LINQ. [Enumerable](http://msdn.microsoft.com/en-us/library/system.linq.enumerable(v=vs.110).aspx)) and [Queryable](http://msdn.microsoft.com/en-us/library/system.linq.queryable(v=vs.110).aspx) are two main static classes of LINQ API that contain extension methods.

*System.Linq* namespace is included by default when you add a new class in Visual Studio, so that you can use LINQ by default.

## Enumerable:

**Enumerable** class includes extension methods for the classes that implement [IEnumerable<T>](http://msdn.microsoft.com/en-us/library/9eekhta0(v=vs.110).aspx) interface, this include all the collection types in [System.Collections.Generic](http://www.tutorialsteacher.com/csharp/csharp-generic-collections) namespaces such as List<T>, Dictionary<T>, SortedList<T>, Queue<T>, HashSet<T>, LinkedList<T> etc.

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

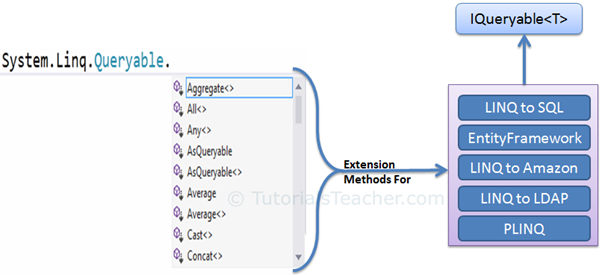
[](http://www.tutorialsteacher.com/Content/images/linq/Enumerable-extension-methods.png)IEnumerable<T> extension methods in Enumerable class

## Queryable:

The**Queryable** class includes extension methods for classes that implement [IQueryable<t>](http://msdn.microsoft.com/en-us/library/vstudio/bb351562(v=vs.100).aspx) interface. IQueryable<T> 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 database like 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 by implementing IQueryable interface.

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

[](http://www.tutorialsteacher.com/Content/images/linq/Queryable-extension-methods.png)IQueryable extension methods in Queryable class

Visit MSDN to know all the extension methods of [Enumerable](https://msdn.microsoft.com/en-us/library/system.linq.enumerable(v=vs.110).aspx) and [Queryable](https://msdn.microsoft.com/en-us/library/system.linq.queryable(v=vs.110).aspx) class.

### Points to Remember :

1. Use **System.Linq** namespace to use LINQ.
2. LINQ api includes two main static class Enumerable & Queryable.
3. The static **Enumerable** class includes extension methods for classes that implements IEnumerable<T> interface.
4. IEnumerable<T> type of collections are in-memory collection like List, Dictionary, SortedList, Queue, HashSet, LinkedList
5. The static **Queryable** class includes extension methods for classes that implements IQueryable<T> interface
6. Remote query provider implements IQueryable<T>. eg. Linq-to-SQL, LINQ-to-Amazon etc.

# 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"

};

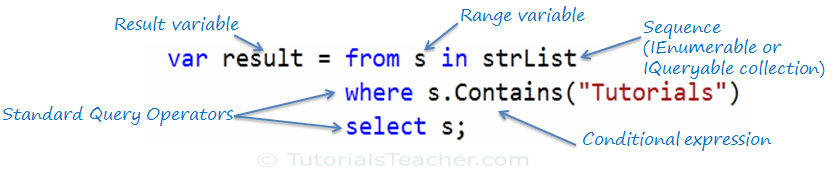
// LINQ Query Syntax

var result = from s in stringList

where s.Contains("Tutorials")

select s;

The following figure shows the structure of LINQ query syntax.

[](http://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](http://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;

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()

Click Next to learn the method syntax.

**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](http://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.

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

Method syntax (also known as fluent syntax) uses extension methods included in the **Enumerable** or **Queryable** 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" ,

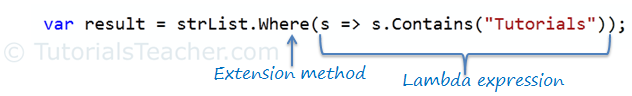
"Java"

};

// LINQ Query Syntax

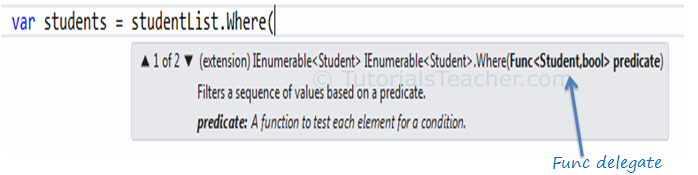
var result = stringList.Where(s => s.Contains("Tutorials"));

The following figure illustrates the structure of LINQ method syntax.

[](http://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](http://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.

[](http://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()

**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](http://www.tutorialsteacher.com/csharp/csharp-anonymous-method) using some special syntax.

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

Anonymous method in C#:

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

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:

Lambda Expression in C#:

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

Lambda Expression in VB.Net:

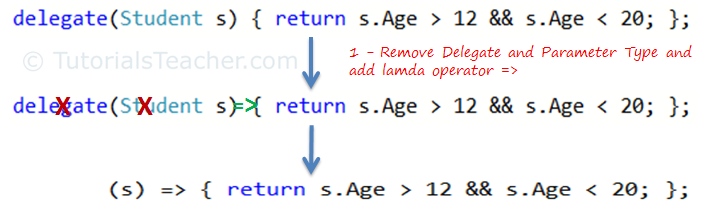
Function(s) s.Age > 12 And s.Age < 20

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

Anonymous method in C#:

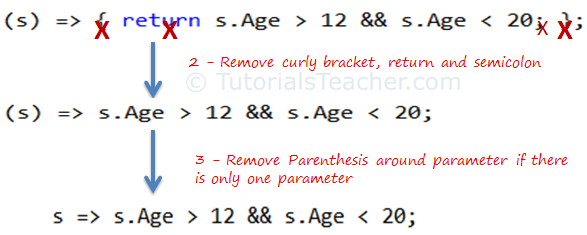
delegate(Student s) { return s.Age > 12 && s.Age < 20; };

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

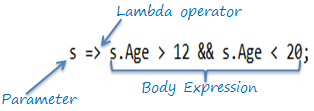
[](http://www.tutorialsteacher.com/Content/images/linq/lambda-expression-1.png)Lambda Expression from Anonymous Method

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.

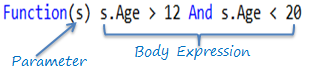
Also, we can remove parenthesis (), if we have only one parameter.

[](http://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:

[](http://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:

[](http://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 in lambda expression C#

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

Example: Specify multiple parameters in lambda expression VB.Net

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

## Lambd expression without any 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 with zero parameter.

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

## Multiple statements in body expression:

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

Example: Lambda expression C#

(s, youngAge) =>

**{**

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

Return s.Age >= youngAge;

**}**

Example: Lambda expression VB.Net

Function(s , youngAge)

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

Return s.Age >= youngAge

End Function

## 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: Lambda expression C#

s =>

{

**int youngAge = 18;**

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

return s.Age >= youngAge;

}

Example: 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](http://www.tutorialsteacher.com/csharp/csharp-func-delegate), [Action](http://www.tutorialsteacher.com/csharp/csharp-action-delegate) and [Predicate](http://www.tutorialsteacher.com/csharp/csharp-predicate).

## Func Delegate:

Use the Func<> delegate when you want to return something from a lambda expression. The last parameter type in a Func<> delegate is the return type and rest are input parameters. Visit [Func delegate](http://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 in C#

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

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

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

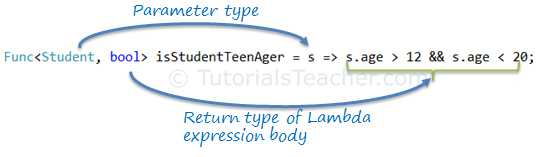
Example: Lamda expression assigned to Func delegate in VB.Net

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 expressions => s.age > 12 && s.age < 20 satisfies the Func<Student, bool> delegate requirement, as shown below:

[](http://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.

C#:

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](http://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 in 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 in VB.Net

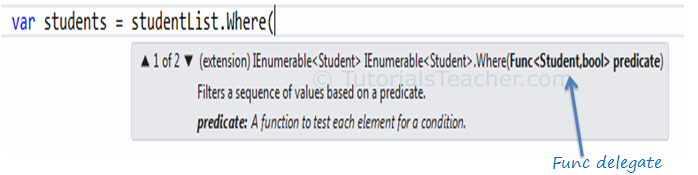
Dim printStudentDetail As Action(Of Student) = Sub(s) Console.WriteLine("Name: {0}, Age: {1} ", s.StudentName, s.Age)

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:

[](http://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;

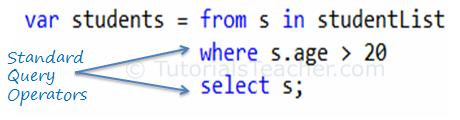
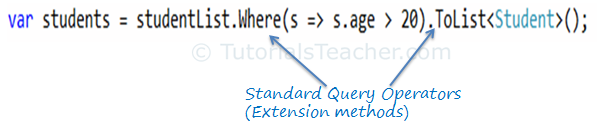
var teenStudents = from s in studentList

where isStudentTeenAger(s)

select s;

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.
8. Standard Query Operators:
9. 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.
10. Standard Query Operators in Query Syntax:
11. [](http://www.tutorialsteacher.com/Content/images/linq/standard-query-operators-linq-query-syntax.png)Standard Query Operators in Query Syntax
12. Standard Query Operators in Method Syntax:
13. [](http://www.tutorialsteacher.com/Content/images/linq/standard-query-operators-linq-method-syntax.png)Standard Query Operators in Method Syntax
14. Standard query operators in query syntax is converted into extension methods at compile time. So both are same.
15. 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 |

1. Learn each Standard Query Operators in the next sections.

# Filtering Operators - 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 | Returns values from the collection based on a predicate function |
| [OfType](http://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](http://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;

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

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):

Exapmle: 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;

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

Exapmle: 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;

}

**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);

Example: Where in method syntax in VB.Net

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

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);

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.

Learn another filtering operator - OfType in the next section.

# 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;

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 

### 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 | Sorts the elements in the collection based on specified fields in ascending or decending order. |
| OrderByDescending | Sorts the collection based on specified fields in descending order. Only valid in method syntax. |
| ThenBy | Only valid in method syntax. Used for second level sorting in ascending order. |
| 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;

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

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() method overloads:

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);

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.

## 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);

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 };

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

We have seen how to do sorting using multiple fields in query syntax in the previous section.

Multiple sorting in method syntax is supported by using ThenBy and ThenByDescending extension methods.

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);

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 follwoing 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.StudentName)

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

.ThenByDescending(Function(s) s.StudentName)

### 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.

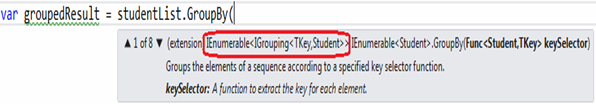
| **Grouping Operators** | **Description** |
| --- | --- |
| GroupBy | The GroupBy operator returns groups of elements based on some key value. Each group is represented by IGrouping<TKey, TElement> object. |
| ToLookup | ToLookup is the same as GroupBy; the only difference is the execution of GroupBy is deferred whereas ToLookup execution is immediate. |

## GroupBy:

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

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.

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

[](http://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

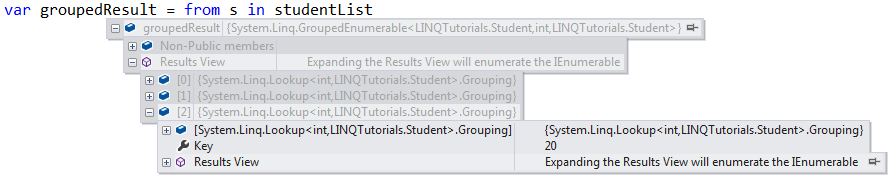
Console.WriteLine("Student Name: {0}", s.StudentName);

}

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.

[](http://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

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

**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);

}

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

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);

}

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 | The Join operator joins two sequences (collections) based on a key and returns a resulted sequence. |
| 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() method overloads:

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);

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

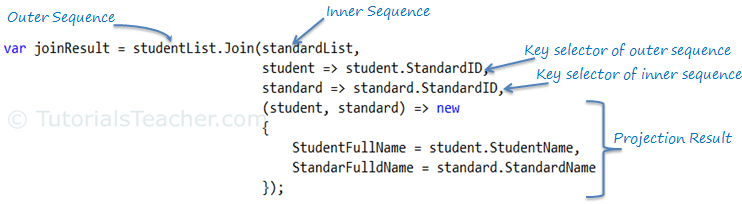
{

StudentName = student.StudentName,

StandardName = standard.StandardName

});

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

[](http://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.

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

});

**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

};

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,

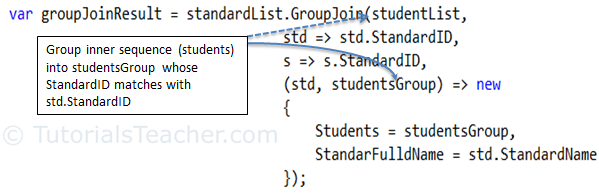
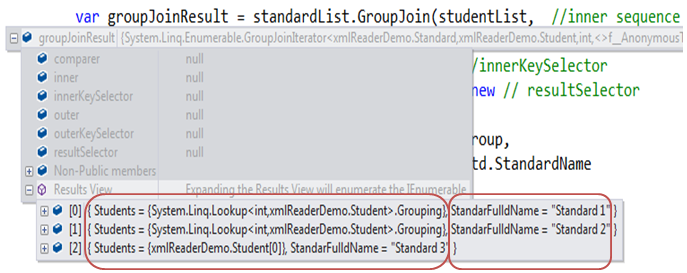
StandardName = std.StandardName

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:   
   from... in outerSequence  
   join... in innerSequence   
   on outerKey equals innerKey  
   select ...
5. Joining Operator: GroupJoin
6. We have seen the Join operator in the previous section. The GroupJoin operator performs the same task as Join operator except that GroupJoin returns a result in group based on specified group key. The GroupJoin operator joins two sequences based on key and groups the result by matching key and then returns the collection of grouped result and key.
7. GroupJoin in Method Syntax:
8. GroupJoin requires same parameters as Join. GroupJoin has following two overload methods:
9. Join() method overloads:
10. public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer, IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector, Func<TInner, TKey> innerKeySelector, Func<TOuter, IEnumerable<TInner>, TResult> resultSelector);
11. public static IEnumerable<TResult> GroupJoin<TOuter, TInner, TKey, TResult>(this IEnumerable<TOuter> outer, IEnumerable<TInner> inner, Func<TOuter, TKey> outerKeySelector, Func<TInner, TKey> innerKeySelector, Func<TOuter, IEnumerable<TInner>, TResult> resultSelector, IEqualityComparer<TKey> comparer);
12. 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. Please notice that resultSelector is of Func delegate type that has second input parameter as IEnumerable type for inner sequence.
13. Now, let's understand GroupJoin using following Student and Standard class where Student class includes StandardID that matches with StandardID of Standard class.
14. Example Classes
15. public class Student{
16. public int StudentID { get; set; }
17. public string StudentName { get; set; }
18. public int StandardID { get; set; }
19. }
20. public class Standard{
21. public int StandardID { get; set; }
22. public string StandardName { get; set; }
23. }
24. Consider the following GroupJoin query example.
25. Example: GroupJoin in Method syntax C#
26. IList<Student> studentList = new List<Student>() {
27. new Student() { StudentID = 1, StudentName = "John", StandardID =1 },
28. new Student() { StudentID = 2, StudentName = "Moin", StandardID =1 },
29. new Student() { StudentID = 3, StudentName = "Bill", StandardID =2 },
30. new Student() { StudentID = 4, StudentName = "Ram", StandardID =2 },
31. new Student() { StudentID = 5, StudentName = "Ron" }
32. };
33. IList<Standard> standardList = new List<Standard>() {
34. new Standard(){ StandardID = 1, StandardName="Standard 1"},
35. new Standard(){ StandardID = 2, StandardName="Standard 2"},
36. new Standard(){ StandardID = 3, StandardName="Standard 3"}
37. };
38. var groupJoin = standardList.GroupJoin(studentList, //inner sequence
39. std => std.StandardID, //outerKeySelector
40. s => s.StandardID, //innerKeySelector
41. (std, studentsGroup) => new // resultSelector
42. {
43. Students = studentsGroup,
44. StandarFulldName = std.StandardName
45. });
46. foreach (var item in groupJoin)
47. {
48. Console.WriteLine(item.StandarFulldName );
49. foreach(var stud in item.Students)
50. Console.WriteLine(stud.StudentName);
51. }
52. Output:  
    Standard 3:
53. In the above example of GroupJoin query, standardList is the outer sequence, because the query starts from it. The first parameter in GroupJoin method is to specify the inner sequence, which is studentList in the above example. The second and third parameters of the GroupJoin() method are to specify a field whose value should be matched using lambda expression, in order to include element in the result. The key selector for the outer sequence standard => standard.StandardID indicates that StandardID field of each elements in standardList should be match with the key of inner sequence studentList student => student.StandardID. If value of both the key field is matched then include that element into grouped collection studentsGroup where key would be StandardID.
54. The last parameter in Join method is an expression to formulate the result. In the above example, result selector includes grouped collection studentGroup and StandardName.
55. The following image illustrate that inner sequence grouped into studentsGroup collection for matching StandardID key and that grouped collection can be used to formulate the result.
56. [](http://www.tutorialsteacher.com/Content/images/linq/linq-groupjoin.png)Grouping Operator - GroupJoin
57. Resultset would include an anonymous objects that has the Students and StandardFullName properties. Students property will be a collection of Students whose StandardID matches with Standard.StandardID.
58. [](http://www.tutorialsteacher.com/Content/images/linq/groupJoin-result.png)GroupJoin Result in Debug View
59. You can access the result using a 'foreach' loop. Each element will have the StandardFullName & Students property, where Students will be a collection.
60. Example: Access GroupJoin Result in C#
61. foreach (var item in groupJoinResult)
62. {
63. Console.WriteLine(item.StandarFulldName );
64. foreach(var stud in item.Students)
65. Console.WriteLine(stud.StudentName);
66. }
67. Following is an example of GroupJoin in VB.Net:
68. Example: GroupJoin in Method syntax VB.Net
69. Dim groupJoin = standardList.GroupJoin( ' outer sequence
70. studentList, ' inner sequence
71. Function(s) s.StandardID, ' outerKeySelector
72. Function(stud) stud.StandardID, ' innerKeySelector
73. Function(s, studentGroup) New With { ' result selector
74. .students = studentGroup,
75. .standardName = s.StandardName
76. })
77. For Each item In groupJoin
79. Console.WriteLine(item.standardName)
81. For Each std In item.students
82. Console.WriteLine( std.StudentName)
83. Next
84. Next
85. Output:
86. Standard 1:   
    John,  
    Moin,  
    Standard 2:  
    Bill,  
    Ram,  
    Standard 3:
87. GroupJoin in Query Syntax:
88. GroupJoin operator in query syntax works slightly different than method syntax. It requires an outer sequence, inner sequence, key selector and result selector. 'on' keyword is used for key selector where the left side of 'equals' operator is the outerKeySelector and the right side of 'equals' is the innerKeySelector. Use the **into** keyword to create the grouped collection.
89. Syntax: GroupJoin in Query syntax
90. from ... in outerSequence
91. join ... in innerSequence
92. on outerKey equals innerKey
93. into groupedCollection
94. select ...
96. The following example demonstrates the GroupJoin in query syntax.
97. Example: GroupJoin Query Syntax C#
98. IList<Student> studentList = new List<Student>() {
99. new Student() { StudentID = 1, StudentName = "John", Age = 13, StandardID =1 },
100. new Student() { StudentID = 2, StudentName = "Moin", Age = 21, StandardID =1 },
101. new Student() { StudentID = 3, StudentName = "Bill", Age = 18, StandardID =2 },
102. new Student() { StudentID = 4, StudentName = "Ram" , Age = 20, StandardID =2 },
103. new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
104. };
105. IList<Standard> standardList = new List<Standard>() {
106. new Standard(){ StandardID = 1, StandardName="Standard 1"},
107. new Standard(){ StandardID = 2, StandardName="Standard 2"},
108. new Standard(){ StandardID = 3, StandardName="Standard 3"}
109. };
110. var groupJoin = from std in standardList
111. join s in studentList
112. on std.StandardID equals s.StandardID
113. into studentGroup
114. select new {
115. Students = studentGroup ,
116. StandardName = std.StandardName
117. };
118. foreach (var item in groupJoin)
119. {
120. Console.WriteLine(item.StandarFulldName );
121. foreach(var stud in item.Students)
122. Console.WriteLine(stud.StudentName);
123. }
124. Example: GroupJoin Query Syntax VB.Net
126. Dim groupJoin = From s In standardList
127. Group Join stud In studentList
128. On stud.StandardID Equals s.StandardID
129. Into Group \_
130. Select \_
131. StudentsGroup = Group,
132. StandardName = s.StandardName
133. Output:
134. Standard 1:   
     John,  
     Moin,  
     Standard 2:  
     Bill,  
     Ram,  
     Standard 3:
135. In the VB.Net, the **InTo** keyword will create a group of all students of same standard and assign it to the **Group** keyword. So, use Group in the projection result.
136. **Note :**Use of the **equals** operator to match key selector. == is not valid.

# Projection Operators: Select, SelectMany

There are two projection operators available in LINQ. 1) Select 2) SelectMany

## Select:

The Select operator always returns an IEnumerable collection which contains elements based on a transformation function. It is similar to the Select clause of SQL that produces a flat result set.

Now, let's understand Select query operator using the following Student class.

Example Classes

public class Student{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

}

### Select in Query Syntax:

LINQ query syntax must end with a **Select**or**GroupBy**clause. The following example demonstrates select operator that returns a string collection of StudentName.

Example: Select in query syntax C#

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

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

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

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

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

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

};

var selectResult = from s in studentList

select s.StudentName;

The select operator can be used to formulat the result as per our requirement. It can be used to return a collection of custom class or anonymous type which includes properties as per our need.

The following example of the select clause returns a collection of [anonymous type](http://www.tutorialsteacher.com/csharp/csharp-anonymous-type) containing the Name and Age property.

Example: Select operator in 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 }

};

// returns collection of anonymous objects with Name and Age property

var selectResult = from s in studentList

select new { Name = "Mr. " + s.StudentName, Age = s.Age };

// iterate selectResult

foreach (var item in selectResult)

Console.WriteLine("Student Name: {0}, Age: {1}", item.Name, item.Age);

Example: Select operator in query syntax VB.Net

Dim selectResult = From s In studentList

Select New With {.Name = s.StudentName, .Age = s.Age}

Output:

Student Name: Mr. John, Age: 13   
Student Name: Mr. Moin, Age: 21  
Student Name: Mr. Bill, Age: 18  
Student Name: Mr. Ram, Age: 20  
Student Name: Mr. Ron, Age: 15

### Select in Method Syntax:

The Select operator is optional in method syntax. However, you can use it to shape the data. In the following example, Select extension method returns a collection of anonymous object with the Name and Age property:

Example: Select in method syntax C#

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

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

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

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

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

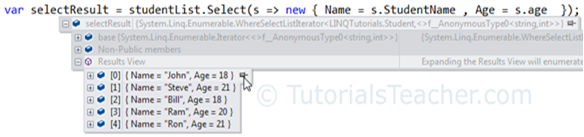
new Student() { StudentID = 5, StudentName = "Ron" , Age = 21 }

};

var selectResult = studentList.Select(s => new { Name = s.StudentName ,

Age = s.Age });

In the above example, selectResult would contain anonymous objects with Name and Age property as shown below in the debug view.

[](http://www.tutorialsteacher.com/Content/images/linq/lnq-select-result.png)Select clause returns an Anonymous objects

Example: Select in method syntax VB.Net

Dim selectResult = studentList.Select(Function(s) New With {.Name = s.StudentName,

.Age = s.Age})

## Select Many:

The SelectMany operator projects sequences of values that are based on a transform function and then flattens them into one sequence.

Visit MSDN for more information on [projection operators](http://msdn.microsoft.com/en-us/library/bb546168.aspx?cs-save-lang=1&cs-lang=csharp#code-snippet-1).

Quantifier Operators:

The quantifier operators evaluate elements of the sequence on some condition and return a boolean value to indicate that some or all elements satisfy the condition.

| **Operator** | **Description** |
| --- | --- |
| All | Checks if all the elements in a sequence satisfies the specified condition |
| Any | Checks if any of the elements in a sequence satisfies the specified condition |
| Contain | Checks if the sequence contains a specific element |

All:

The All operator evalutes each elements in the given collection on a specified condition and returns True if all the elements satisfy a condition.

Example: All operator 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 }

};

// checks whether all the students are teenagers

bool areAllStudentsTeenAger = studentList.All(s => s.Age > 12 && s.Age < 20);

Console.WriteLine(areAllStudentsTeenAger);

Example: All operator VB.Net

Dim areAllStudentsTeenAger = studentList.All(Function(s) s.Age > 12 And s.Age < 20)

Output:

Any:

Any checks whether any element satisfy given condition or not? In the following example, Any operation is used to check whether any student is teen ager or not.

Example: Any operator C#

bool isAnyStudentTeenAger = studentList.Any(s => s.age > 12 && s.age < 20);

Example: Any operator VB.Net

Dim isAnyStudentTeenAger = studentList.Any(Function(s) s.Age > 12 And s.Age < 20)

Output:

true

**Note :**Quantifier operators are **Not Supported** with C# query syntax.

# Quantifier Operator: Contains

The Contains operator checks whether a specified element exists in the collection or not and returns a boolean.

The Contains() extension method has following two overloads. The first overload method requires a value to check in the collection and the second overload method requires additional parameter of IEqualityComparer type for custom equalality comparison.

Contains() method overloads:

public static bool Contains<TSource>(this IEnumerable<TSource> source, TSource value);

public static bool Contains<TSource>(this IEnumerable<TSource> source,

TSource value,

IEqualityComparer<TSource> comparer);

As mentioned above, the Contains() extension method requires a value to check as a input parameter. Type of a value must be same as type of generic collection. The following example of Contains checks whether 10 exists in the collection or not. Please notice that int is a type of generic collection.

Example: Contains operator C#

IList<int> intList = new List<int>() { 1, 2, 3, 4, 5 };

bool result = intList.Contains(10); // returns false

Example: Contains operator VB.Net

Dim intList As IList(Of Integer) = New List(Of Integer) From {1, 2, 3, 4, 5}

Dim result = intList.Contains(10) ' returns false

The above example works well with primitive data types. However, it will not work with a custom class. Consider the following example:

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 }

};

Student std = new Student(){ StudentID =3, StudentName = "Bill"};

bool result = studentList.Contains(std); //returns false

As you can see in the above example, Contains returns false even if "Bill" exists in the studentList. This is because the Contains extension method only compares reference of an object but not the actual values of an object. So to compare values of the student object, you need to create a class by implementing IEqualityComparer interface, that compares values of two Student objects and returns boolean.

The following is a StudentComparer class that implements IEqualityComparer<Student> interface to compare values of two Students objects:

Example: IEqualityComperer

class StudentComparer : IEqualityComparer<Student>

{

public bool Equals(Student x, Student y)

{

if (x.StudentID == y.StudentID &&

x.StudentName.ToLower() == y.StudentName.ToLower())

return true;

return false;

}

public int GetHashCode(Student obj)

{

return obj.GetHashCode();

}

}

Now, you can use the above StudentComparer class in second overload method of Contains extension method that accepts second parameter of IEqualityComparer type, as below:

Example: Contains with Comparer class 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 }

};

Student std = new Student(){ StudentID =3, StudentName = "Bill"};

bool result = studentList.Contains(std, new StudentComparer()); //**returns true**

So thus, you have to use comparer class in order to get corrent result from Contains extension method for custom classes.

The following is a similar example in VB.Net:

Example: Contains with Comparer class VB.Net

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

}

Public Class StudentComparer Implements IEqualityComparer(Of Student)

Public Function Equals1(x As Student, y As Student) As Boolean Implements IEqualityComparer(Of Student).Equals

If (x.StudentID = y.StudentID And x.StudentName.ToLower() = y.StudentName.ToLower()) Then

Return True

End If

Return False

End Function

Public Function GetHashCode1(obj As Student) As Integer Implements IEqualityComparer(Of Student).GetHashCode

Return obj.GetHashCode()

End Function

End Class

Sub Main

Dim studentList = New List(Of Student) From {

New Student() With {.StudentID = 1, .StudentName = "John", .Age = 18},

New Student() With {.StudentID = 2, .StudentName = "Steve", .Age = 15},

New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 25},

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

New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 19}

}

Dim std As New Student With {.StudentID = 3, .StudentName = "Bill"}

Dim result = studentList.Contains(std, New StudentComparer()) ' returns true

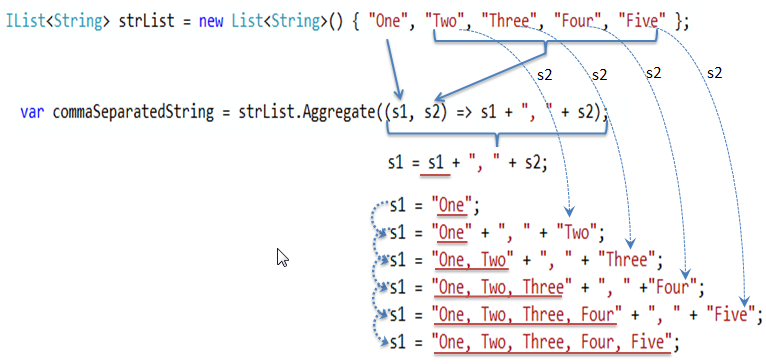
End Sub

**Note :**Quantifier operators are **Not Supported** with query syntax in C# or VB.Net.

### Points to Remember:

1. All, Any & Contains are quantifier operators in LINQ.
2. All checks if all the elements in a sequence satisfies the specified condition.
3. Any check if any of the elements in a sequence satisfies the specified condition
4. Contains operator checks whether specified element exists in the collection or not.
5. Use custom class that derives IEqualityOperator with Contains to check for the object in the collection.
6. All, Any & Contains are not supported in query syntax in C# or VB.Net.
7. Aggregation Operators: Aggregate
8. The aggregation operators perform mathematical operations like Average, Aggregate, Count, Max, Min and Sum, on the numeric property of the elements in the collection.

| **Method** | **Description** |
| --- | --- |
| Aggregate | Performs a custom aggregation operation on the values in the collection. |
| Average | calculates the average of the numeric items in the collection. |
| Count | Counts the elements in a collection. |
| LongCount | Counts the elements in a collection. |
| Max | Finds the largest value in the collection. |
| Min | Finds the smallest value in the collection. |
| Sum | Calculates sum of the values in the collection. |

1. Aggregate:
2. The Aggregate method performs an accumulate operation. Aggregate extension method has the following overload methods:
3. Aggregate() method overloads:
4. public static TSource Aggregate<TSource>(this IEnumerable<TSource> source,
5. Func<TSource, TSource, TSource> func);
6. public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source,
7. TAccumulate seed,
8. Func<TAccumulate, TSource, TAccumulate> func);
9. public static TResult Aggregate<TSource, TAccumulate, TResult>(this IEnumerable<TSource> source,
10. TAccumulate seed,
11. Func<TAccumulate, TSource, TAccumulate> func,
12. Func<TAccumulate, TResult> resultSelector);
13. The following example demonstrates Aggregate method that returns comma seperated elements of the string list.
14. Example: Aggregate operator in method syntax C#
15. IList<String> strList = new List<String>() { "One", "Two", "Three", "Four", "Five"};
16. var commaSeperatedString = strList.Aggregate((s1, s2) => s1 + ", " + s2);
17. Console.WriteLine(commaSeperatedString);
18. Output:
19. One, Two, Three, Four, Five
20. In the above example, Aggregate extension method returns comma separated strings from strList collection. The following image illustrates the whole aggregate operation performed in the above example.
21. [](http://www.tutorialsteacher.com/Content/images/linq/linq-aggregate-1.png)
22. Aggregate extension method
23. As per the above figure, first item of strList "One" will be pass as s1 and rest of the items will be passed as s2. The lambda expression (s1, s2) => s1 + ", " + s2 will be treated like s1 = s1 + ", " + s1 where s1 will be accumulated for each item in the collection. Thus, Aggregate method will return comma separated string.
24. Example: Aggregate operator in method syntax VB.Net
25. Dim strList As IList(Of String) = New List(Of String) From {
26. "One",
27. "Two",
28. "Three",
29. "Four",
30. "Five"
31. }
32. Dim commaSeparatedString = strList.Aggregate(Function(s1, s2) s1 + ", " + s2)
33. Aggregate method with seed value:
34. The second overload method of Aggregate requires first parameter for seed value to accumulate. Second parameter is Func type delegate:  
    TAccumulate Aggregate<TSource, TAccumulate>(**TAccumulate seed, Func<TAccumulate, TSource, TAccumulate> func**); .
35. The following example uses string as a seed value in the Aggregate extension method.
36. Example: Aggregate method with seed value in C#
38. // Student collection
39. IList<Student> studentList = new List<Student>>() {
40. new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
41. new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
42. new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
43. new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
44. new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
45. };
46. string commaSeparatedStudentNames = studentList.Aggregate<Student, string>(
47. "Student Names: ", // seed value
48. (str, s) => str += s.StudentName + "," );
49. Console.WriteLine(commaSeparatedStudentNames);
50. Example: Aggregate method with seed value in VB.Net
52. // Student collection
53. Dim studentList = New List(Of Student) From {
54. New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},
55. New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},
56. New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},
57. New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},
58. New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}
59. }
60. Dim commaSeparatedStudentNames = studentList.Aggregate(Of String)(
61. "Student Names: ",
62. Function(str, s) str + s.StudentName + ",")
63. Console.WriteLine(commaSeparatedStudentNames);
65. Output:
66. Student Names: John, Moin, Bill, Ram, Ron,
67. In the above example, the first parameter of the Aggregate method is the "Student Names: " string that will be accumulated with all student names. The comma in the lambda expression will be passed as a second parameter.
68. The following example use Aggregate operator to add the age of all the students.
69. Example: Aggregate method with seed value in C#
71. // Student collection
72. IList<Student> studentList = new List<Student>>() {
73. new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
74. new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
75. new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
76. new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
77. new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
78. };
79. int SumOfStudentsAge = studentList.Aggregate<Student, int>(0,
80. (totalAge, s) => totalAge += s.Age );
81. Aggregate method with result selector:
82. Now, let's see third overload method that required the third parameter of the Func delegate expression for result selector, so that you can formulate the result.
83. Consider the following example.
84. Example: Aggregate method C#
86. IList<Student> studentList = new List<Student>>() {
87. new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
88. new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
89. new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
90. new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
91. new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
92. };
93. string commaSeparatedStudentNames = studentList.Aggregate<Student, string,string>(
94. String.Empty, // seed value
95. (str, s) => str += s.StudentName + ",", // returns result using seed value, String.Empty goes to lambda expression as str
96. str => str.Substring(0,str.Length - 1 )); // result selector that removes last comma
97. Console.WriteLine(commaSeparatedStudentNames);
98. In the above example, we have specified a lambda expression str => str.Substring(0,str.Length - 1 ) which will remove the last comma in the string result. Below is the same example in VB.Net.
99. Example: Aggregate method VB.Net
101. // Student collection
102. Dim studentList = New List(Of Student) From {
103. New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},
104. New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},
105. New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},
106. New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},
107. New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}
108. }
109. Dim commaSeparatedStudentNames = studentList.Aggregate(Of String, String)(
110. String.Empty,
111. Function(str, s) str + s.StudentName + ",",
112. Function(str) str.Substring(0, str.Length - 1))
113. Console.WriteLine(commaSeparatedStudentNames);
114. Output:
115. John, Moin, Bill, Ram, Ron
116. **Note :**Aggregate operator is **Not Supported** with query syntax in C# or VB.Net.
117. Learn about another aggregate operator - Average in the next section.
118. Aggregation Operator: Average
119. Average extension method calculates the average of the numeric items in the collection. Average method returns nullable or non-nullable decimal, double or float value.
120. The following example demonstrate Agerage method that returns average value of all the integers in the collection.
121. Example: Average method - C#
122. IList<int> intList = new List<int>>() { 10, 20, 30 };
123. var avg = intList.Average();
124. Console.WriteLine("Average: {0}", avg);
125. You can specify an int, decimal, double or float property of a class as a lambda expression of which you want to get an average value. The following example demonstrates Average method on the complex type.
126. Example: Average operator in method syntax C#
127. IList<Student> studentList = new List<Student>>() {
128. new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
129. new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
130. new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
131. new Student() { StudentID = 4, StudentName = "Ram" , Age = 20} ,
132. new Student() { StudentID = 5, StudentName = "Ron" , Age = 15 }
133. };
134. var avgAge = studentList.Average(s => s.Age);
135. Console.WriteLine("Average Age of Student: {0}", avgAge);
136. Example: Average operator in method syntax VB.Net
137. Dim studentList = New List(Of Student) From {
138. New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},
139. New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},
140. New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},
141. New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},
142. New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}
143. }
144. Dim avgAge = studentList.Average(Function(s) s.Age)
145. Console.WriteLine("Average Age of Student: {0}", avgAge)
146. Output:
147. Average Age of Student: 17.4
148. The Average operator in query syntax is **Not Supported** in C#. However, it is supported in VB.Net as shown below.
149. Example: Average operator in query syntax VB.Net
150. Dim studentList = New List(Of Student) From {
151. New Student() With {.StudentID = 1, .StudentName = "John", .Age = 13},
152. New Student() With {.StudentID = 2, .StudentName = "Moin", .Age = 21},
153. New Student() With {.StudentID = 3, .StudentName = "Bill", .Age = 18},
154. New Student() With {.StudentID = 4, .StudentName = "Ram", .Age = 20},
155. New Student() With {.StudentID = 5, .StudentName = "Ron", .Age = 15}
156. }
157. Dim avgAge = Aggregate st In studentList Into Average(st.Age)
158. Console.WriteLine("Average Age of Student: {0}", avgAge)
159. Output:

# Average Age of Student: 17.4 Aggregation Operator: Count

The Count operator returns the number of elements in the collection or number of elements that have satisfied the given condition.

The Count() extension method has the following two overloads:

Count() method overloads in C#

int Count<TSource>();

int Count<TSource>(Func<TSource, bool> predicate);

The first overload method of Count returns the number of elements in the specified collection, whereas the second overload method returns the number of elements which have satisfied the specified condition given as lambda expression/predicate function.

The following example returns the number of elements in a collection using the Count() method in the method syntax.

Example: Count() in 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 = "Mathew" , Age = 15 }

};

var numOfStudents = studentList.Count();

Console.WriteLine("Number of Students: {0}", numOfStudents);

Example: Count() 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 numOfStudents = studentList.Count()

Console.WriteLine("Number of Students: {0}", numOfStudents);

Output:

Number of Students: 5

In the following example, we get the count of students whose age is 18 or more by specifying condition in the Count method:

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 = "Mathew" , Age = 15 }

};

var numOfStudents = studentList.Count(s => s.Age >= 18);

Console.WriteLine("Number of Students: {0}", numOfStudents);

Output:

Number of Students: 3

**Note :**Count() extension method with predicate parameter is **Not Supported** in VB.Net.

Count operator in query syntax:

Example: Count operator in query syntax 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}

}

Dim numOfStudents = Aggregate st In studentList

Into Count(st.Age >= 18)

Console.WriteLine("Number of Student: {0}", numOfStudents);

Output:

Number of Students: 3

C# Query Syntax doesn't support aggregation operators. However, you can wrap the query into brackets and use an aggregation functions as shown below.

Example: Count operator in query syntax C#

var totalAge = (from s in studentList

select s.age).Count();

Aggregation Operator: Max

The Max operator returns the largest numeric element from a collection.

The following example demonstrates Max() on primitive collection.

Example: Max method - C#

IList<int> intList = new List<int>() { 10, 21, 30, 45, 50, 87 };

var largest = intList.Max();

Console.WriteLine("Largest Element: {0}", largest);

var largestEvenElements = intList.Max(i => {

if(i%2 == 0)

return i;

return 0;

});

Console.WriteLine("Largest Even Element: {0}", largestEvenElements );

Output:

Largest Element: 87  
Largest Even Element: 50

The following example demonstrates Max() method on the complex type collection.

Example: Max operator in method 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 oldest = studentList.Max(s => s.Age);

Console.WriteLine("Oldest Student Age: {0}", oldest);

Example: Max operator in method syntax 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 oldest = studentList.Max(Function(s) s.Age)

Console.WriteLine("Oldest Student Age: {0}", oldest)

Output:

Oldest Student Ag: 21

Max returns a result of any data type. The following example shows how you can find a student with the longest name in the collection:

C#: LINQ Max()

public class Student : IComparable<Student>

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

public int StandardID { get; set; }

public int CompareTo(Student other)

{

if (this.StudentName.Length >= other.StudentName.Length)

return 1;

return 0;

}

}

class Program

{

static void Main(string[] args)

{

// 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 = "Steve" , Age = 15 }

};

var studentWithLongName = studentList.Max();

Console.WriteLine("Student ID: {0}, Student Name: {1}",

.StudentID, studentWithLongName.StudentName)

}

}

Output:

Student ID: 5, Student Name: Steve

**Note :**You can use Min extension method/operator the same way as Max.

As per the above example, to find the student with the longest name, you need to implement IComparable<T> interface and compare student names' length in CompareTo method. So now, you can use Max() method which will use CompareTo method in order to return appropriate result.

Max operator in Query Syntax:

Max operator is **Not Supported** in C# Query syntax. However, it is supported in VB.Net query syntax as shown below.

Example: Max operator in query syntax 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 maxAge = Aggregate st In studentList Into Max(st.Age)

Console.WriteLine("Maximum Age of Student: {0}", maxAge);

Output:

# MaximAggregation Operator: Sum

The Sum() method calculates the sum of numeric items in the collection.

The following example demonstrates Sum() on primitive collection.

Example: LINQ Sum() - C#

IList<int> intList = new List<int>() { 10, 21, 30, 45, 50, 87 };

var total = intList.Sum();

Console.WriteLine("Sum: {0}", total);

var sumOfEvenElements = intList.Sum(i => {

if(i%2 == 0)

return i;

return 0;

});

Console.WriteLine("Sum of Even Elements: {0}", sumOfEvenElements );

Output:

Sum: 243  
Sum of Even Elements: 90

The following example calculates sum of all student's age and also number of adult students in a student collection.

Example: LINQ Sum() - 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 sumOfAge = studentList.Sum(s => s.Age);

Console.WriteLine("Sum of all student's age: {0}", sumOfAge);

var numOfAdults = studentList.Sum(s => {

if(s.Age >= 18)

return 1;

else

return 0;

});

Console.WriteLine("Total Adult Students: {0}", numOfAdults);

Example: Sum in method syntax - 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}

}

Dim sumOfAge = studentList.Sum(Function(s) s.Age)

Console.WriteLine("Total Age of Student: {0}", sumOfAge)

Dim numOfAdults = studentList.Sum(Function(s)

if(s.Age >= 18)

return 1

else

return 0

end if

end function)

Console.WriteLine("Total Adult Students: {0}", numOfAdults)

Out

Sum operator in query syntax:

Sum operator is **Not Supported** in C# Query syntax.

Example: Sum operator in query syntax - 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}

}

Dim totalAge = Aggregate st In studentList Into Sum(st.Age)

Console.WriteLine("Sum of all student's age: {0}", totalAge);

Output:

Sum of all student's age: 87

# um Age of StElement Operators: ElementAt, ElementAtOrDefault

Element operators return a particular element from a sequence (collection).

The following table lists all the Element operators in LINQ.

| **Element Operators (Methods)** | **Description** |
| --- | --- |
| ElementAt | Returns the element at a specified index in a collection |
| ElementAtOrDefault | Returns the element at a specified index in a collection or a default value if the index is out of range. |
| First | Returns the first element of a collection, or the first element that satisfies a condition. |
| FirstOrDefault | Returns the first element of a collection, or the first element that satisfies a condition. Returns a default value if index is out of range. |
| Last | Returns the last element of a collection, or the last element that satisfies a condition |
| LastOrDefault | Returns the last element of a collection, or the last element that satisfies a condition. Returns a default value if no such element exists. |
| Single | Returns the only element of a collection, or the only element that satisfies a condition. |
| SingleOrDefault | Returns the only element of a collection, or the only element that satisfies a condition. Returns a default value if no such element exists or the collection does not contain exactly one element. |

The ElementAt() method returns an element from the specified index from a given collection. If the specified index is out of the range of a collection then it will throw an *Index out of range* exception. Please note that index is a zero based index.

The ElementAtOrDefault() method also returns an element from the specified index from a collaction and if the specified index is out of range of a collection then it will return a default value of the data type instead of throwing an error.

The following example demonstrates ElementAt and ElementAtOrDefault method on primitive collection.

Example: LINQ ElementAt() and ElementAtOrDefault() - C#

IList<int> intList = new List<int>() { 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { "One", "Two", null, "Four", "Five" };

Console.WriteLine("1st Element in intList: {0}", intList.ElementAt(0));

Console.WriteLine("1st Element in strList: {0}", strList.ElementAt(0));

Console.WriteLine("2nd Element in intList: {0}", intList.ElementAt(1));

Console.WriteLine("2nd Element in strList: {0}", strList.ElementAt(1));

Console.WriteLine("3rd Element in intList: {0}", intList.ElementAtOrDefault(2));

Console.WriteLine("3rd Element in strList: {0}", strList.ElementAtOrDefault(2));

Console.WriteLine("10th Element in intList: {0} - default int value",

intList.ElementAtOrDefault(9));

Console.WriteLine("10th Element in strList: {0} - default string value (null)",

strList.ElementAtOrDefault(9));

Console.WriteLine("intList.ElementAt(9) throws an exception: Index out of range");

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

Console.WriteLine(intList.ElementAt(9));

Output:

exception: Index was out of range....

As you can see in the above example, intList.ElementAtOrDefault(9) returns 0 (default value of int) because intList does not include 10th element. However intList.ElementAt(9) throws "Index out of range" exception.The same way, strList.ElementAtOrDefault(9) returns null which is default value of string type. (console display empty space because it cannot display null)

Thus, it is advisable to use the ElementAtOrDefault extension method to eliminate the possibility of a runtime exception.

Element Operators: First & FirstOrDefault

The First and FirstOrDefault method returns an element from the zeroth index in the collection i.e. the first element. Also, it returns an element that satisfies the specified condition.

| **Element Operators** | **Description** |
| --- | --- |
| First | Returns the first element of a collection, or the first element that satisfies a condition. |
| FirstOrDefault | Returns the first element of a collection, or the first element that satisfies a condition. Returns a default value if index is out of range. |

First and FirstOrDefault has two overload methods. The first overload method doesn't take any input parameter and returns the first element in the collection. The second overload method takes the lambda expression as predicate delegate to specify a condition and returns the first element that satisfies the specified condition.

Overload methods of First and FirstOrDefault - C#

public static TSource First<TSource>(this IEnumerable<TSource> source);

public static TSource First<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source);

public static TSource FirstOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

The First() method returns the first element of a collection, or the first element that satisfies the specified condition using lambda expression or Func delegate. If a given collection is empty or does not include any element that satisfied the condition then it will throw InvalidOperation exception.

The FirstOrDefault() method does the same thing as First() method. The only difference is that it returns default value of the data type of a collection if a collection is empty or doesn't find any element that satisfies the condition.

The following example demonstrates First() method.

Example: LINQ First() - C#

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

Console.WriteLine("1st Element in intList: {0}", intList.First());

Console.WriteLine("1st Even Element in intList: {0}", intList.First(i => i % 2 == 0));

Console.WriteLine("1st Element in strList: {0}", strList.First());

Console.WriteLine("emptyList.First() throws an InvalidOperationException");

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

Console.WriteLine(emptyList.First());

Output:

exception: Sequence contains no elements...

The following example demonstrates FirstOrDefault() method.

Example: LINQ FirstOrDefault() - C#

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

Console.WriteLine("1st Element in intList: {0}", intList.FirstOrDefault());

Console.WriteLine("1st Even Element in intList: {0}",

intList.FirstOrDefault(i => i % 2 == 0));

Console.WriteLine("1st Element in strList: {0}", strList.FirstOrDefault());

Console.WriteLine("1st Element in emptyList: {0}", emptyList.FirstOrDefault());

Output:

Element in emptyList:

Be careful while specifying condition in First() or FirstOrDefault(). First() will throw an exception if a collection does not include any element that satisfies the specified condition or includes null element.

If a collection includes null element then FirstOrDefault() throws an exception while evaluting the specified condition. The following example demonstrates this.

Example: LINQ First() & FirstOrDefault() - C#

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

Console.WriteLine("1st Element which is greater than 250 in intList: {0}",

intList.First( i > 250));

Console.WriteLine("1st Even Element in intList: {0}",

strList.FirstOrDefault(s => s.Contains("T")));

Output:

# Run-time exception: Sequence contains no matching elementElement Operators : Last & LastOrDefault

| **Element Operators** | **Description** |
| --- | --- |
| Last | Returns the last element from a collection, or the last element that satisfies a condition |
| LastOrDefault | Returns the last element from a collection, or the last element that satisfies a condition. Returns a default value if no such element exists. |

Last and LastOrDefault has two overload methods. One overload method doesn't take any input parameter and returns last element from the collection. Second overload method takes a lambda expression to specify a condition and returns last element that satisfies the specified condition.

Overload methods of Last and LastOrDefault - C#

public static TSource Last<TSource>(this IEnumerable<TSource> source);

public static TSource Last<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static TSource LastOrDefault<TSource>(this IEnumerable<TSource> source);

public static TSource LastOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

The Last() method returns the last element from a collection, or the last element that satisfies the specified condition using lambda expression or Func delegate. If a given collection is empty or does not include any element that satisfied the condition then it will throw InvalidOperation exception.

The LastOrDefault() method does the same thing as Last() method. The only difference is that it returns default value of the data type of a collection if a collection is empty or doesn't find any element that satisfies the condition.

The following example demonstrates Last() method.

Example: LINQ Last() - C#

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

Console.WriteLine("Last Element in intList: {0}", intList.Last());

Console.WriteLine("Last Even Element in intList: {0}", intList.Last(i => i % 2 == 0));

Console.WriteLine("Last Element in strList: {0}", strList.Last());

Console.WriteLine("emptyList.Last() throws an InvalidOperationException");

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

Console.WriteLine(emptyList.Last());

Output:

exception: Sequence contains no elements...

The following example demonstrates LastOrDefault() method.

Example: LINQ LastOrDefault() - C#

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

Console.WriteLine("Last Element in intList: {0}", intList.LastOrDefault());

Console.WriteLine("Last Even Element in intList: {0}",

intList.LastOrDefault(i => i % 2 == 0));

Console.WriteLine("Last Element in strList: {0}", strList.LastOrDefault());

Console.WriteLine("Last Element in emptyList: {0}", emptyList.LastOrDefault());

Output:

Element in emptyList:

Be careful while specifying condition in Last() or LastOrDefault(). Last() will throw an exception if a collection does not include any element that satisfies the specified condition or includes null element.

If a collection includes null element then LastOrDefault() throws an exception while evaluting the specified condition. The following example demonstrates this.

Example: LINQ Last() & LastOrDefault() - C#

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

Console.WriteLine("Last Element which is greater than 250 in intList: {0}",

intList.Last(i => i > 250));

Console.WriteLine("Last Even Element in intList: {0}",

strList.LastOrDefault(s => s.Contains("T")));

Output:

# Run-time eElement Operators: Single & SingleOrDefault

| **Element Operators** | **Description** |
| --- | --- |
| Single | Returns the only element from a collection, or the only element that satisfies a condition. If Single() found no elements or more than one elements in the collection then throws InvalidOperationException. |
| SingleOrDefault | The same as Single, except that it returns a default value of a specified generic type, instead of throwing an exception if no element found for the specified condition. However, it will thrown InvalidOperationException if it found more than one element for the specified condition in the collection. |

Single and SingleOrDefault have two overload methods. The first overload method doesn't take any input parameter and returns a single element in the collection. The second overload method takes the lambda expression as a predicate delegate that specifies the condition and returns a single element that satisfies the specified condition.

Overload methods of Single and SingleOrDefault - C#

public static TSource Single<TSource>(this IEnumerable<TSource> source);

public static TSource Single<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

public static TSource SingleOrDefault<TSource>(this IEnumerable<TSource> source);

public static TSource SingleOrDefault<TSource>(this IEnumerable<TSource> source, Func<TSource, bool> predicate);

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

The SingleOrDefault() method does the same thing as Single() method. The only difference is that it returns default value of the data type of a collection if a collection is empty, includes more than one element or finds no element or more than one element for the specified condition.

Example: Single in method syntax C#

IList<int> oneElementList = new List<int>() { 7 };

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

Console.WriteLine("The only element in oneElementList: {0}", oneElementList.Single());

Console.WriteLine("The only element in oneElementList: {0}",

oneElementList.SingleOrDefault());

Console.WriteLine("Element in emptyList: {0}", emptyList.SingleOrDefault());

Console.WriteLine("The only element which is less than 10 in intList: {0}",

intList.Single(i => i < 10));

//Followings throw an exception

//Console.WriteLine("The only Element in intList: {0}", intList.Single());

//Console.WriteLine("The only Element in intList: {0}", intList.SingleOrDefault());

//Console.WriteLine("The only Element in emptyList: {0}", emptyList.Single());

Output:

The only element in oneElementList: 7  
The only element in oneElementList: 7  
Element in emptyList: 0  
The only element which is less than 10 in intList: 7

The following example code throws an exception because Single() or SingleOrDefault() returns none or multiple elements for the specified condition.

C#: Single() and SingleOrDefault()

IList<int> oneElementList = new List<int>() { 7 };

IList<int> intList = new List<int>() { 7, 10, 21, 30, 45, 50, 87 };

IList<string> strList = new List<string>() { null, "Two", "Three", "Four", "Five" };

IList<string> emptyList = new List<string>();

//following throws error because list contains more than one element which is less than 100

Console.WriteLine("Element less than 100 in intList: {0}", intList.Single(i => i < 100));

//following throws error because list contains more than one element which is less than 100

Console.WriteLine("Element less than 100 in intList: {0}",

intList.SingleOrDefault(i => i < 100));

//following throws error because list contains more than one elements

Console.WriteLine("The only Element in intList: {0}", intList.Single());

//following throws error because list contains more than one elements

Console.WriteLine("The only Element in intList: {0}", intList.SingleOrDefault());

//following throws error because list does not contains any element

Console.WriteLine("The only Element in emptyList: {0}", emptyList.Single());

**Points to Remember :**

1. Single() expects one and only one element in the collection.
2. Single() throws an exception when it gets no element or more than one elements in the collection.
3. If specified a condition in Single() and result contains no element or more than one elements then it throws an exception.
4. SingleOrDefault() will return default value of a data type of generic collection if there is no elements in a colection or for the specified condition.
5. SingleOrDefault() will throw an exception if there is more than one elements in a colection or for the specified condition.

# Equality Operator: SequenceEqual

There is only one equality operator: SequenceEqual. The SequenceEqual method checks whether the number of elements, value of each element and order of elements in two collections are equal or not.

If the collection contains elements of primitive data types then it compares the values and number of elements, whereas collection with complex type elements, checks the references of the objects. So, if the objects have the same reference then they considered as equal otherwise they are considered not equal.

The following example demonstrates the SequenceEqual method with the collection of primitive data types.

Example: SequenceEqual in Method Syntax C#

IList<string> strList1 = new List<string>(){"One", "Two", "Three", "Four", "Three"};

IList<string> strList2 = new List<string>(){"One", "Two", "Three", "Four", "Three"};

bool isEqual = strList1.SequenceEqual(strList2); // returns true

Console.WriteLine(isEqual);

Output:

true

If the order of elements are not the same then SequenceEqual() method returns false.

Example: SequenceEqual in Method Syntax C#

IList<string> strList1 = new List<string>(){"One", "Two", "Three", "Four", "Three"};

IList<string> strList2 = new List<string>(){ "Two", "One", "Three", "Four", "Three"};

bool isEqual = strList1.SequenceEqual(strList2); // returns false

Console.WriteLine(isEqual);

Output:

false

The SequenceEqual extension method checks the references of two objects to determine whether two sequences are equal or not. This may give wrong result. Consider following example:

Example: SequenceEqual in C#

Student std = new Student() { StudentID = 1, StudentName = "Bill" };

IList<Student> studentList1 = new List<Student>(){ std };

IList<Student> studentList2 = new List<Student>(){ std };

bool isEqual = studentList1.SequenceEqual(studentList2); // returns true

Student std1 = new Student() { StudentID = 1, StudentName = "Bill" };

Student std2 = new Student() { StudentID = 1, StudentName = "Bill" };

IList<Student> studentList3 = new List<Student>(){ std1};

IList<Student> studentList4 = new List<Student>(){ std2 };

isEqual = studentList3.SequenceEqual(studentList4);// returns false

In the above example, studentList1 and studentList2 contains the same student object, std. So studentList1.SequenceEqual(studentList2) returns true. But, stdList1 and stdList2 contains two seperate student object, std1 and std2. So now, stdList1.SequenceEqual(stdList2) will return false even if std1 and std2 contain the same value.

To compare the values of two collection of complex type (reference type or object), you need to implement IEqualityComperar<T> interface as shown below.

Example: IEqualityComparer C#:

class StudentComparer : IEqualityComparer<Student>

{

public bool Equals(Student x, Student y)

{

if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower())

return true;

return false;

}

public int GetHashCode(Student obj)

{

return obj.GetHashCode();

}

}

Now, you can use above StudentComparer class in SequenceEqual extension method as a second parameter to compare the values:

Example: Compare object type elements using SequenceEqual C#

IList<Student> studentList1 = 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 }

};

IList<Student> studentList2 = 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 }

};

// following returns true

bool isEqual = studentList1.SequenceEqual(studentList2, new StudentComparer());

**Points to Remember :**

1. The SequenceEqual method compares the number of items and their values for primitive data types.
2. The SequenceEqual method compares the reference of objects for complex data types.
3. Use IEqualityComparer class to compare two colection of complex type using SequenceEqual method.
4. Concatenation Operator: Concat
5. The Concat() method appends two sequences of the same type and returns a new sequence (collection).
6. Example: Concat in C#
7. IList<string> collection1 = new List<string>() { "One", "Two", "Three" };
8. IList<string> collection2 = new List<string>() { "Five", "Six"};
9. var collection3 = collection1.Concat(collection2);
10. foreach (string str in collection3)
11. Console.WriteLine(str);
12. Output:

Six

1. Example: Concat in C#
2. IList<int> collection1 = new List<int>() { 1, 2, 3 };
3. IList<int> collection2 = new List<int>() { 4, 5, 6 };
4. var collection3 = collection1.Concat(collection2);
5. foreach (int i in collection3)
6. Console.WriteLine(i);
7. Output:
8. Concat operator is not supported in query syntax in C# or VB.Net.

Generation Operator: DefaultIfEmpty:

The DefaultIfEmpty() method returns a new collection with the default value if the given collection on which DefaultIfEmpty() is invoked is empty.

Another overload method of DefaultIfEmpty() takes a value parameter that should be replaced with default value.

Consider the following example.

Example: DefaultIfEmpty C#

IList<string> emptyList = new List<string>();

var newList1 = emptyList.DefaultIfEmpty();

var newList2 = emptyList.DefaultIfEmpty("None");

Console.WriteLine("Count: {0}" , newList1.Count());

Console.WriteLine("Value: {0}" , newList1.ElementAt(0));

Console.WriteLine("Count: {0}" , newList2.Count());

Console.WriteLine("Value: {0}" , newList2.ElementAt(0));

Output:

: None

In the above example, emptyList.DefaultIfEmpty() returns a new string collection with one element whose value is null because null is a default value of string. Another method emptyList.DefaultIfEmpty("None") returns a string collection with one element whose value is "None" instead of null.

The following example demonstrates calling DefaultIfEmpty on int collection.

Example: DefaultIfEmpty C#

IList<int> emptyList = new List<int>();

var newList1 = emptyList.DefaultIfEmpty();

var newList2 = emptyList.DefaultIfEmpty(100);

Console.WriteLine("Count: {0}" , newList1.Count());

Console.WriteLine("Value: {0}" , newList1.ElementAt(0));

Console.WriteLine("Count: {0}" , newList2.Count());

Console.WriteLine("Value: {0}" , newList2.ElementAt(0));

Output:

: 100

The following example demonstrates DefaultIfEmpty() method on complex type collection.

Example: DefaultIfEmpty C#:

IList<Student> emptyStudentList = new List<Student>();

var newStudentList1 = studentList.DefaultIfEmpty(new Student());

var newStudentList2 = studentList.DefaultIfEmpty(new Student(){

StudentID = 0,

StudentName = "" });

Console.WriteLine("Count: {0} ", newStudentList1.Count());

Console.WriteLine("Student ID: {0} ", newStudentList1.ElementAt(0));

Console.WriteLine("Count: {0} ", newStudentList2.Count());

Console.WriteLine("Student ID: {0} ", newStudentList2.ElementAt(0).StudentID);

Output:

# Generation Operators: Empty, Range, Repeat

LINQ includes generation operators DefaultIfEmpty, Empty, Range & Repeat. The Empty, Range & Repeat methods are not extension methods for IEnumerable or IQueryable but they are simply static methods defined in a static class Enumerable.

| **Method** | **Description** |
| --- | --- |
| Empty | Returns an empty collection |
| Range | Generates collection of IEnumerable<T> type with specified number of elements with sequential values, starting from first element. |
| Repeat | Generates a collection of IEnumerable<T> type with specified number of elements and each element contains same specified value. |

Empty:

The Empty() method is not an extension method of IEnumerable or IQueryable like other LINQ methods. It is a static method included in Enumerable static class. So, you can call it the same way as other static methods like Enumerable.Empty<TResult>(). The Empty() method returns an empty collection of a specified type as shown below.

Example: Enumerable.Empty()

var emptyCollection1 = Enumerable.Empty<string>();

var emptyCollection2 = Enumerable.Empty<Student>();

Console.WriteLine("Count: {0} ", emptyCollection1.Count());

Console.WriteLine("Type: {0} ", emptyCollection1.GetType().Name );

Console.WriteLine("Count: {0} ",emptyCollection2.Count());

Console.WriteLine("Type: {0} ", emptyCollection2.GetType().Name );

Output:

Type: String[]   
Range:

The Range() method returns a collection of IEnumerable<T> type with specified number of elements and sequential values starting from the first element.

Example: Enumerable.Range()

var intCollection = Enumerable.Range(10, 10);

Console.WriteLine("Total Count: {0} ", intCollection.Count());

for(int i = 0; i < intCollection.Count(); i++)

Console.WriteLine("Value at index {0} : {1}", i, intCollection.ElementAt(i));

Output:

9 : 19

In the above example, Enumerable.Range(10, 10) creates collection with 10 integer elements with the sequential values starting from 10. First parameter specifies the starting value of elements and second parameter specifies the number of elements to create.

Repeat:

The Repeat() method generates a collection of IEnumerable<T> type with specified number of elements and each element contains same specified value.

Example: Repeat

var intCollection = Enumerable.Repeat<int>(10, 10);

Console.WriteLine("Total Count: {0} ", intCollection.Count());

for(int i = 0; i < intCollection.Count(); i++)

Console.WriteLine("Value at index {0} : {1}", i, intCollection.ElementAt(i));

Output:

at index 9: 10

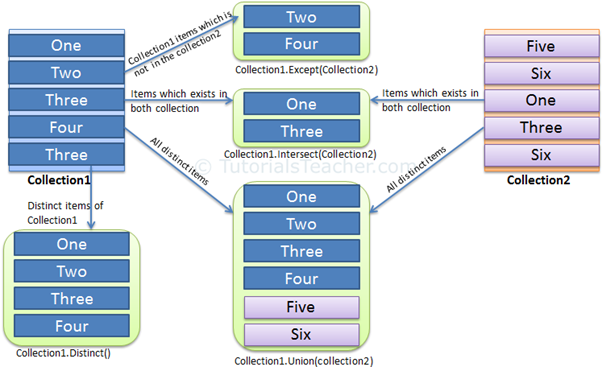
In the above example, Enumerable.Repeat<int>(10, 10) creates collection with 100 integer type elements with the repeated value of 10. First parameter specifies the values of all the elements and second parameter specifies the number of elements to create.

Set operator: Distinct

The following table lists all Set operators available in LINQ.

| **Set Operators** | **Usage** |
| --- | --- |
| Distinct | Returns distinct values from a collection. |
| Except | Returns the difference between two sequences, which means the elements of one collection that do not appear in the second collection. |
| Intersect | Returns the intersection of two sequences, which means elements that appear in both the collections. |
| Union | Returns unique elements from two sequences, which means unique elements that appear in either of the two sequences. |

The following figure shows how each set operators works on the collections:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-set-operators.png)LINQ Set operators

Distinct:

The Distinct extension method returns a new collection of unique elements from the given collection.

Example: Distinct C#

IList<string> strList = new List<string>(){ "One", "Two", "Three", "Two", "Three" };

IList<int> intList = new List<int>(){ 1, 2, 3, 2, 4, 4, 3, 5 };

var distinctList1 = strList.Distinct();

foreach(var str in distinctList1)

Console.WriteLine(str);

var distinctList2 = intList.Distinct();

foreach(var i in distinctList2)

Console.WriteLine(i);

Output:

The Distinct extension method doesn't compare values of complex type objects. You need to implement IEqualityComparer<T> interface in order to compare the values of complex types. In the following example, StudentComparer class implements IEqualityComparer<Student> to compare Student< objects.

Example: Implement IEqualityComparer in C#

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

}

class StudentComparer : IEqualityComparer<Student>

{

public bool Equals(Student x, Student y)

{

if (x.StudentID == y.StudentID

&& x.StudentName.ToLower() == y.StudentName.ToLower())

return true;

return false;

}

public int GetHashCode(Student obj)

{

return obj.StudentID.GetHashCode();

}

}

Now, you can pass an object of the above StudentComparer class in the Distinct() method as a parameter to compare the Student objects as shown below.

Example: Distinct 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 = 3, StudentName = "Bill", Age = 25 } ,

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

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

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

};

var distinctStudents = studentList.Distinct(new StudentComparer());

foreach(Student std in distinctStudents)

Console.WriteLine(std.StudentName);

Output:

John  
Distinct operator in Query Syntax:

The Distinct operator is **Not Supported** in C# Query syntax. However, you can use Distinct method of query variable or wrap whole query into brackets and then call Distinct().

Use the Distinct keyword in VB.Net query syntax:

Example: Distinct in query syntax VB.Net

Dim strList = New List(Of string) From {"One", "Three", "Two", "Two", "One" }

Dim distinctStr = From s In strList \_

Select s Distinct

Set operator: Except

The Except() method requires two collections. It returns a new collection with elements from the first collection which do not exist in the second collection (parameter collection).

Example: Except in method syntax C#

IList<string> strList1 = new List<string>(){"One", "Two", "Three", "Four", "Five" };

IList<string> strList2 = new List<string>(){"Four", "Five", "Six", "Seven", "Eight"};

var result = strList1.Except(strList2);

foreach(string str in result)

Console.WriteLine(str);

Output:

One   
Two  
Three

Except extension method doesn't return the correct result for the collection of complex types. You need to implement IEqualityComparer interface in order to get the correct result from Except method.

Implement IEqualityComparer interface for Student class as shown below:

Example: IEqualityComparer with Except method C#

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

}

class StudentComparer : IEqualityComparer<Student>

{

public bool Equals(Student x, Student y)

{

if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower())

return true;

return false;

}

public int GetHashCode(Student obj)

{

return obj.StudentID.GetHashCode();

}

}

Now, you can pass above StudentComparer class in Except extension method in order to get the correct result:

Example: Except() with object type C#

IList<Student> studentList1 = 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 = 5, StudentName = "Ron" , Age = 19 }

};

IList<Student> studentList2 = new List<Student>() {

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

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

};

var resultedCol = studentList1.Except(studentList2,new StudentComparer());

foreach(Student std in resultedCol)

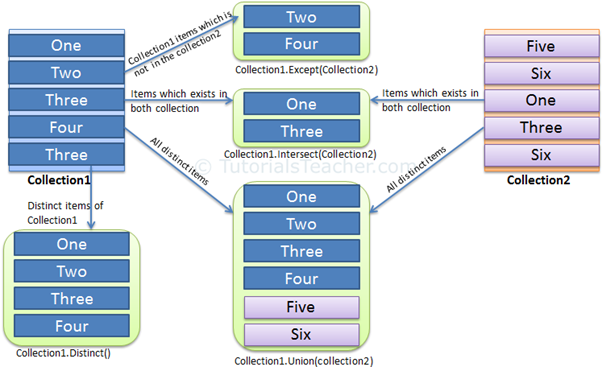
Console.WriteLine(std.StudentName);

Output:

John  
Steve

The Except operator is **Not Supported** in C# & VB.Net Query syntax. However, you can use Distinct method on query variable or wrap whole query into brackets and then call Except().

The following figure shows how each set operators works on the collections:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-set-operators.png)LINQ Set operators

Set operator: Intersect

The Intersect extension method requires two collections. It returns a new collection that includes common elements that exists in both the collection. Consider the following example.

Example: Intersect in method syntax C#

IList<string> strList1 = new List<string>() { "One", "Two", "Three", "Four", "Five" };

IList<string> strList2 = new List<string>() { "Four", "Five", "Six", "Seven", "Eight"};

var result = strList1.Intersect(strList2);

foreach(string str in result)

Console.WriteLine(str);

Output:

Four  
Five

The Intersect extension method doesn't return the correct result for the collection of complex types. You need to implement IEqualityComparer interface in order to get the correct result from Intersect method.

Implement IEqualityComparer interface for Student class as shown below:

Example: Use IEqualityComparer with Intersect in C#

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

}

class StudentComparer : IEqualityComparer<Student>

{

public bool Equals(Student x, Student y)

{

if (x.StudentID == y.StudentID &&

x.StudentName.ToLower() == y.StudentName.ToLower())

return true;

return false;

}

public int GetHashCode(Student obj)

{

return obj.StudentID.GetHashCode();

}

}

Now, you can pass above StudentComparer class in the Intersect extension method in order to get the correct result:

Example: Intersect operator C#

IList<Student> studentList1 = 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 = 5, StudentName = "Ron" , Age = 19 }

};

IList<Student> studentList2 = new List<Student>() {

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

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

};

var resultedCol = studentList1.Intersect(studentList2, new StudentComparer());

foreach(Student std in resultedCol)

Console.WriteLine(std.StudentName);

Output:

Bill  
Ron

The Intersect operator is **Not Supported** in C# & VB.Net Query syntax. However, you can use the Intersect method on a query variable or wrap whole query into brackets and then call Intersect().

The following figure shows how each set operators works on the collections:

# Set operator: Union

The Union extension method requires two collections and returns a new collection that includes distinct elements from both the collections. Consider the following example.

Example: Union() in C#

IList<string> strList1 = new List<string>() { "One", "Two", "three", "Four" };

IList<string> strList2 = new List<string>() { "Two", "THREE", "Four", "Five" };

var result = strList1.Union(strList2);

foreach(string str in result)

Console.WriteLine(str);

Output:

One   
Two   
three   
THREE   
Four   
Five

The Union extension method doesn't return the correct result for the collection of complex types. You need to implement IEqualityComparer interface in order to get the correct result from Union method.

Implement IEqualityComparer interface for Student class as below:

Example: Union operator with IEqualityComparer:

public class Student

{

public int StudentID { get; set; }

public string StudentName { get; set; }

public int Age { get; set; }

}

class StudentComparer : IEqualityComparer<Student>

{

public bool Equals(Student x, Student y)

{

if (x.StudentID == y.StudentID && x.StudentName.ToLower() == y.StudentName.ToLower())

return true;

return false;

}

public int GetHashCode(Student obj)

{

return obj.StudentID.GetHashCode();

}

}

Now, you can pass above StudentComparer class in the Union extension method to get the correct result:

Example: Union operator C#

IList<Student> studentList1 = 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 = 5, StudentName = "Ron" , Age = 19 }

};

IList<Student> studentList2 = new List<Student>() {

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

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

};

var resultedCol = studentList1.Union(studentList2, new StudentComparer());

foreach(Student std in resultedCol)

Console.WriteLine(std.StudentName);

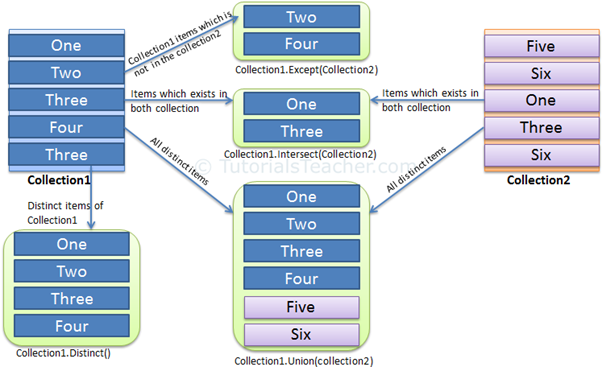
Output:

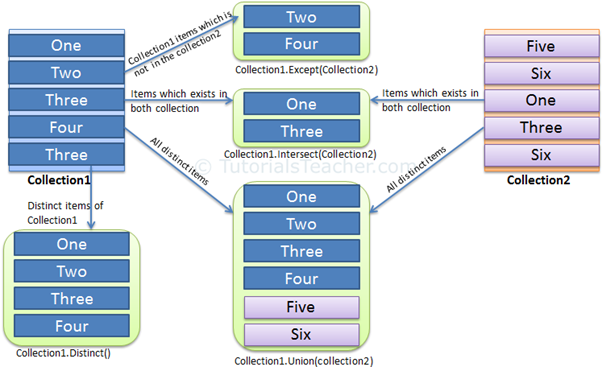
John   
Steve   
Bill  
Ron

### Query Syntax:

The Union operator is **Not Supported** in C# & VB.Net Query syntax. However, you can use Union method on query variable or wrap whole query into brackets and then call Union().

The following figure shows how each set operators works on the collections:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-set-operators.png)LINQ Set operators

[](http://www.tutorialsteacher.com/Content/images/linq/linq-set-operators.png)LINQ Set operators

# contains no mPartitioning Operators: Skip & SkipWhile

Partitioning operators split the sequence (collection) into two parts and return one of the parts.

| **Method** | **Description** |
| --- | --- |
| Skip | Skips elements up to a specified position starting from the first element in a sequence. |
| SkipWhile | Skips elements based on a condition until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition, it then skips 0 elements and returns all the elements in the sequence. |
| Take | Takes elements up to a specified position starting from the first element in a sequence. |
| TakeWhile | Returns elements from the first element until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition then returns an empty collection. |

Skip:

The Skip() method skips the specified number of element starting from first element and returns rest of the elements.

Example: Skip() - C#

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

var newList = strList.Skip(2);

foreach(var str in newList)

Console.WriteLine(str);

Output:

Three  
Skip operator in Query Syntax:

The Skip & SkipWhile operator is **Not Supported in C# query syntax**. However, you can use Skip/SkipWhile method on a query variable or wrap whole query into brackets and then call Skip/SkipWhile.

The following example demonstrates skip operator in query syntax - VB.NET

Example: Skip operator in VB.Net

Dim skipResult = From s In studentList

Skip 3

Select s

SkipWhile:

As the name suggests, the SkipWhile() extension method in LINQ skip elements in the collection till the specified condition is true. It returns a new collection that includes all the remaining elements once the specified condition becomes false for any element.

The SkipWhile() method has two overload methods. One method accepts the predicate of Func<TSource, bool> type and other overload method accepts the predicate Func<TSource, int, bool> type that pass the index of an element.

In the following example, SkipWhile() method skips all elements till it finds a string whose length is equal or more than 4 characters.

Example: SkipWhile in C#

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

"One",

"Two",

"Three",

"Four",

"Five",

"Six" };

var resultList = strList.SkipWhile(s => s.Length < 4);

foreach(string str in resultList)

Console.WriteLine(str);

Output:

Three   
Four  
Five  
Six

In the above example, SkipWhile() skips first two elements because their length is less than 3 and finds third element whose length is equal or more than 4. Once it finds any element whose length is equal or more than 4 characters then it will not skip any other elements even if they are less than 4 characters.

Now, consider the following example where SkipWhile() does not skip any elements because the specified condition is false for the first element.

Example: SkipWhile in C#

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

"Three",

"One",

"Two",

"Four",

"Five",

"Six" };

var resultList = strList.SkipWhile(s => s.Length < 4);

foreach(string str in resultList)

Console.WriteLine(str);

Output:

Three

The second overload of SkipWhile passes an index of each elements. Consider the following example.

Example: SkipWhile with index in C#

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

"One",

"Two",

"Three",

"Four",

"Five",

"Six" };

var result = strList.SkipWhile((s, i) => s.Length > i);

foreach(string str in result)

Console.WriteLine(str);

Output:

Five  
Six

In the above example, the lambda expression includes element and index of an elements as a parameter. It skips all the elements till the length of a string element is greater than it's index.

Skip/SkipWhile operator in Query Syntax:

Skip & SkipWhile operator is **NOT Supported in C# query syntax**. However, you can use Skip/SkipWhile method on a query variable or wrap whole query into brackets and then call Skip/SkipWhile().

Example: SkipWhile method in VB.Net

Dim strList = New List(Of string) From {

"One",

"Two",

"Three",

"Four",

"Five",

"Six" }

Dim skipWhileResult = From s In studentList

Skip While s.Length < 4

Select s

Output:

# Three  Four  Five  SixPartitioning Operators: Take & TakeWhile

Partitioning operators split the sequence (collection) into two parts and returns one of the parts.

| **Method** | **Description** |
| --- | --- |
| Skip | Skips elements up to a specified position starting from the first element in a sequence. |
| SkipWhile | Skips elements based on a condition until an element does not satisfy the condition. If the first element itself doesn't satisfy the condition, it then skips 0 elements and returns all the elements in the sequence. |
| Take | Takes elements up to a specified position starting from the first element in a sequence. |
| TakeWhile | Returns elements from the given collection until the specified condition is true. If the first element itself doesn't satisfy the condition then returns an empty collection. |

Take:

The Take() extension method returns the specified number of elements starting from the first element.

Example: Take() in C#

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

var newList = strList.Take(2);

foreach(var str in newList)

Console.WriteLine(str);

Output:

One  
Two

Take & TakeWhile operator is **Not Supported in C# query syntax**. However, you can use Take/TakeWhile method on query variable or wrap whole query into brackets and then call Take/TakeWhile().

Example: Take operator in query syntax VB.Net

Dim takeResult = From s In studentList

Take 3

Select s

TakeWhile:

The TakeShile() extension method returns elements from the given collection until the specified condition is true. If the first element itself doesn't satisfy the condition then returns an empty collection.

The TakeWhile method has two overload methods. One method accepts the predicate of Func<TSource, bool> type and the other overload method accepts the predicate Func<TSource, int, bool> type that passes the index of element.

In the following example, TakeWhile() method returns a new collection that includes all the elements till it finds a string whose length less than 4 characters.

Example: TakeWhile in C#

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

"Three",

"Four",

"Five",

"Hundred" };

var result = strList.TakeWhile(s => s.Length > 4);

foreach(string str in result)

Console.WriteLine(str);

Output:

Three

In the above example, TakeWhile() includes only first element because second string element does not satisfied the condition.

TakeWhile also passes an index of current element in predicate function. Following example of TakeWhile method takes elements till length of string element is greater than it's index

Example: TakeWhile in C#:

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

"One",

"Two",

"Three",

"Four",

"Five",

"Six" };

var resultList = strList.TakeWhile((s, i) => s.Length > i);

foreach(string str in resultList)

Console.WriteLine(str);

Output:

# One  Two  Three FourConversion Operators:

The Conversion operators in LINQ are useful in converting the type of the elements in a sequence (collection). There are three types of conversion operators: **As** operators (AsEnumerable and AsQueryable), **To** operators (ToArray, ToDictionary, ToList and ToLookup), and **Casting** operators (Cast and OfType).

The following table lists all the conversion operators.

| **Method** | **Description** |
| --- | --- |
| AsEnumerable | Returns the input sequence as IEnumerable<t> |
| AsQueryable | Converts IEnumerable to IQueryable, to simulate a remote query provider |
| Cast | Coverts a non-generic collection to a generic collection (IEnumerable to IEnumerable<T>) |
| [OfType](http://www.tutorialsteacher.com/linq/linq-filtering-operators-oftype) | Filters a collection based on a specified type |
| ToArray | Converts a collection to an array |
| ToDictionary | Puts elements into a Dictionary based on key selector function |
| ToList | Converts collection to List |
| [ToLookup](http://www.tutorialsteacher.com/linq/linq-grouping-operator-groupby-tolookup) | Groups elements into an Lookup<TKey,TElement> |

AsEnumerable & AsQueryable:

The AsEnumerable and AsQueryable methods cast or convert a source object to IEnumerable<T> or IQueryable<T> respectively.

Consider the following example: (courtesy: [Jon Skeet](http://stackoverflow.com/a/9063184/861716))

Example: AsEnumerable & AsQueryable operator in C#:

class Program

{

static void ReportTypeProperties<T>(T obj)

{

Console.WriteLine("Compile-time type: {0}", typeof(T).Name);

Console.WriteLine("Actual type: {0}", obj.GetType().Name);

}

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 } ,

};

ReportTypeProperties( studentArray);

ReportTypeProperties(studentArray.AsEnumerable());

ReportTypeProperties(studentArray.AsQueryable());

}

}

Output:

: EnumerableQuery`1

As you can see in the above example AsEnumerable and AsQueryable methods convert compile time type to IEnumerable and IQueryable respectively

Visit [stackoverflow](http://stackoverflow.com/questions/17968469/whats-the-differences-between-tolist-asenumerable-asqueryable) for detail information on AsEnumerable and AsQueryable method.

Cast:

Cast does the same thing as AsEnumerable<T>. It cast the source object into IEnumerable<T>.

Example: Cast operator in C#

class Program

{

static void ReportTypeProperties<T>(T obj)

{

Console.WriteLine("Compile-time type: {0}", typeof(T).Name);

Console.WriteLine("Actual type: {0}", obj.GetType().Name);

}

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 } ,

};

ReportTypeProperties( studentArray);

ReportTypeProperties(studentArray.Cast<Student>());

}

}

Output:

Student[]

studentArray.Cast<Student>() is the same as (IEnumerable<Student>)studentArray but Cast<Student>() is more readable.

To Operators: ToArray(), ToList(), ToDictionary():

As the name suggests, ToArray(), ToList(), ToDictionary() method converts a source object into an array, List or Dictionary respectively.

**To** operators force the execution of the query. It forces the remote query provider to execute a query and get the result from the underlying data source e.g. SQL Server database.

Example: ToArray & ToList in C#

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

"One",

"Two",

"Three",

"Four",

"Three"

};

string[] strArray = strList.ToArray<string>();// converts List to Array

IList<string> list = strArray.ToList<string>(); // converts array into list

ToDictionary - Converts a Generic list to a generic dictionary:

Example: ToDictionary in 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 = "Ron" , age = 21 }

};

//following converts list into dictionary where StudentId is a key

IDictionary<int, Student> studentDict =

studentList.ToDictionary<Student, int>(s => s.StudentID);

foreach(var key in studentDict.Keys)

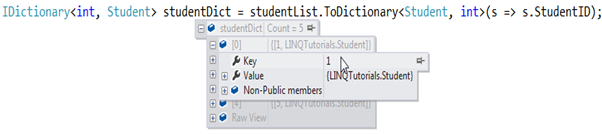
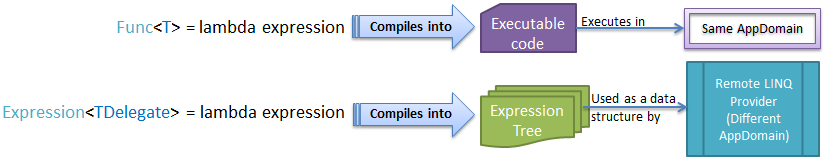
Console.WriteLine("Key: {0}, Value: {1}",

key, (studentDict[key] as Student).StudentName);

Output:

Key: 1, Value: John   
Key: 2, Value: Steve   
Key: 3, Value: Bill   
Key: 4, Value: Ram   
Key: 5, Value: Ron

The following figure shows how studentDict in the above example contains a key-value pair, where key is a StudentID and the value is Student object.

1. [](http://www.tutorialsteacher.com/Content/images/linq/linq-todictionary.png)LINQ-ToDictionary Operator
2. Expression in LINQ:
3. We have learned that the [lambda Expression](http://www.tutorialsteacher.com/linq/linq-lambda-expression) can be assigned to the Func or Action type delegates to process over in-memory collections. The .NET compiler converts the lambda expression assigned to Func or Action type delegate into executable code at compile time.
4. LINQ introduced the new type called [Expression](http://msdn.microsoft.com/en-us/library/bb335710(v=vs.110).aspx) that represents strongly typed lambda expression. It means lambda expression can also be assigned to Expression<TDelegate> type. The .NET compiler converts the lambda expression which is assigned to Expression<TDelegate> into an [Expression tree](http://www.tutorialsteacher.com/linq/expression-tree) instead of executable code. This expression tree is used by remote LINQ query providers as a data structure to build a runtime query out of it (such as LINQ-to-SQL, EntityFramework or any other LINQ query provider that implements IQueryable<T> interface).
5. The following figure illustrates differences when the lambda expression assigned to the Func or Action delegate and the Expression in LINQ.
6. [](http://www.tutorialsteacher.com/Content/images/linq/expression.png)Expression and Func
7. We will learn Expression tree in the next section but first, let's see how to define and invoke an Expression.
8. Define an Expression:
9. Take the reference of *System.Linq.Expressions* namespace and use an Expression<TDelegate> class to define an Expression. Expression<TDelegate> requires delegate type Func or Action.
10. For example, you can assign lambda expression to the isTeenAger variable of Func type delegate, as shown below:
11. Example: Define Func delegate for an expression in C#
13. public class Student
14. {
15. public int StudentID { get; set; }
16. public string StudentName { get; set; }
17. public int Age { get; set; }
18. }
19. Func<Student, bool> isTeenAger = s => s.Age > 12 && s.Age < 20;
20. Example: Define Func delegate for an expression in VB.Net
22. Dim isTeenAger As Func(Of Student, Boolean) = Function(s) s.Age > 12 And s.Age < 20
23. And now, you can convert the above Func type delegate into an Expression by wrapping Func delegate with Expresson, as below:
24. Example: Define Expression in C#
26. Expression<Func<Student, bool>> isTeenAgerExpr = s => s.age > 12 && s.age < 20;
27. Example: Define Expression in VB.Net
29. Dim isTeenAgerExpr As Expression(Func(Of Student, Boolean)) =
30. Function(s) s.Age > 12 And s.Age < 20
31. in the same way, you can also wrap an Action<t> type delegate with Expression if you don't return a value from the delegate.
32. Example: Define Expression in C#
34. Expression<Action<Student>> printStudentName = s => Console.WriteLine(s.StudentName);
35. Example: Define Expression in VB.Net
37. Dim printStudentName As Expression(Action(Of Student) =
38. Function(s) Console.WriteLine(s.StudentName);
39. Thus, you can define Expression<TDelegate> type. Now, let's see how to invoke delegate wrapped by an Expression<TDelegate>.
40. Invoke an Expression:
41. You can invoke the delegate wrapped by an Expression the same way as a delegate, but first you need to compile it using the Compile() method. Compile() returns delegateof **Func** or **Action** type so that you can invoke it like a delegate.
42. Example: Invoke Expression in C#
44. Expression<Func<Student, bool>> isTeenAgerExpr = s => s.age > 12 && s.age < 20;
45. //compile Expression using Compile method to invoke it as Delegate
46. Func<Student, bool> isTeenAger = isTeenAgerExpr.Compile();
48. //Invoke
49. bool result = isTeenAger(new Student(){ StudentID = 1, StudentName = "Steve", Age = 20});
50. Example: Invoke Expression in VB.Net
52. Dim isTeenAgerExpr As Expression(Of Func(Of Student, Boolean)) =
53. Function(s) s.Age > 12 And s.Age < 20
54. 'compile Expression using Compile method to invoke it as Delegate
55. Dim isTeenAger As Func(Of Student, Boolean) = isTeenAgerExpr.Compile()
56. Dim result = isTeenAger(New Student() With { .StudentID = 1, .StudentName = "Steve", .Age = 20})
57. Learn about the Expression tree in detail in the next section.

Expression Tree:

You have learned about the [Expression](http://www.tutorialsteacher.com/linq/linq-expression) in the previous section. Now, let's learn about the Expresion tree here.

Expression tree as name suggests is nothing but expressions arranged in a tree-like data structure. Each node in an expression tree is an expression. For example, an expression tree can be used to represent mathematical formula x < y where x, < and y will be represented as an expression and arranged in the tree like structure.

Expression tree is an in-memory representation of a lambda expression. It holds the actual elements of the query, not the result of the query.

The expression tree makes the structure of the lambda expression transparent and explicit. You can interact with the data in the expression tree just as you can with any other data structure.

For example, consider the following isTeenAgerExpr expression:

Example: Expression in C#

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

The compiler will translate the above expression into the following expression tree:

Example: Expression Tree in C#

Expression.Lambda<Func<Student, bool>>(

Expression.AndAlso(

Expression.GreaterThan(Expression.Property(pe, "Age"), Expression.Constant(12, typeof(int))),

Expression.LessThan(Expression.Property(pe, "Age"), Expression.Constant(20, typeof(int)))),

new[] { pe });

You can also build an expression tree manually. Let's see how to build an expression tree for the following simple lambda expression:

Example: Func delegate in C#:

Func<Student, bool> isAdult = s => s.age >= 18;

This Func type delegate will be treated like the following method:

C#:

public bool function(Student s)

{

return s.Age > 18;

}

To create the expression tree, first of all, create a parameter expression where Student is the type of the parameter and 's' is the name of the parameter as below:

Step 1: Create Parameter Expression in C#

ParameterExpression pe = Expression.Parameter(typeof(Student), "s");

Now, use Expression.Property() to create s.Age expression where s is the parameter and Age is the property name of Student. (**Expression** is an abstract class that contains static helper methods to create the Expression tree manually.)

Step 2: Create Property Expression in C#

MemberExpression me = Expression.Property(pe, "Age");

Now, create a constant expression for 18:

Step 3: Create Constant Expression in C#

ConstantExpression constant = Expression.Constant(18, typeof(int));

Till now, we have built expression trees for s.Age (member expression) and 18 (constant expression). We now need to check whether a member expression is greater than a constant expression or not. For that, use the Expression.GreaterThanOrEqual() method and pass the member expression and constant expression as parameters:

Step 4: Create Binary Expression in C#

BinaryExpression body = Expression.GreaterThanOrEqual(me, constant);

Thus, we have built an expression tree for a lambda expression body s.Age >= 18. We now need to join the parameter and body expressions. Use Expression.Lambda(body, parameters array) to join the body and parameter part of the lambda expression s => s.age >= 18:

Step 5: Create Lambda Expression in C#

var isAdultExprTree = Expression.Lambda<Func<Student, bool>>(body, new[] { pe });

This way you can build an expression tree for simple Func delegates with a lambda expression.

Example: Expression Tree in C#

ParameterExpression pe = Expression.Parameter(typeof(Student), "s");

MemberExpression me = Expression.Property(pe, "Age");

ConstantExpression constant = Expression.Constant(18, typeof(int));

BinaryExpression body = Expression.GreaterThanOrEqual(me, constant);

var ExpressionTree = Expression.Lambda<Func<Student, bool>>(body, new[] { pe });

Console.WriteLine("Expression Tree: {0}", ExpressionTree);

Console.WriteLine("Expression Tree Body: {0}", ExpressionTree.Body);

Console.WriteLine("Number of Parameters in Expression Tree: {0}",

ExpressionTree.Parameters.Count);

Console.WriteLine("Parameters in Expression Tree: {0}", ExpressionTree.Parameters[0]);

Example: Expression Tree in VB.Net

Dim pe As ParameterExpression = Expression.Parameter(GetType(Student), "s")

Dim mexp As MemberExpression = Expression.Property(pe, "Age")

Dim constant As ConstantExpression = Expression.Constant(18, GetType(Integer))

Dim body As BinaryExpression = Expression.GreaterThanOrEqual(mexp, constant)

Dim ExpressionTree As Expression(Of Func(Of Student, Boolean)) =

Expression.Lambda(Of Func(Of Student, Boolean))(body, New ParameterExpression() {pe})

Console.WriteLine("Expression Tree: {0}", ExpressionTree)

Console.WriteLine("Expression Tree Body: {0}", ExpressionTree.Body)

Console.WriteLine("Number of Parameters in Expression Tree: {0}",

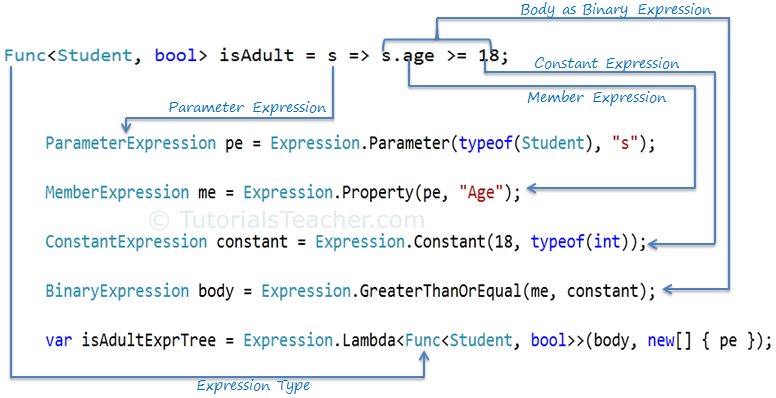
ExpressionTree.Parameters.Count)

Console.WriteLine("Parameters in Expression Tree: {0}", ExpressionTree.Parameters(0))

Output:

Expression Tree: s => (s.Age >= 18)   
Expression Tree Body: (s.Age >= 18)  
Number of Parameters in Expression Tree: 1  
Parameters in Expression Tree: s

The following image illustrates the whole process of creating an expression tree:

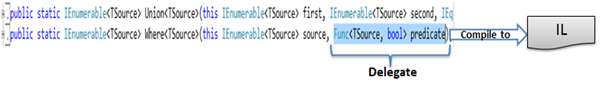
[](http://www.tutorialsteacher.com/Content/images/linq/linq-construct-expression-tree.png)Construct Expression Tree

Why Expression Tree?

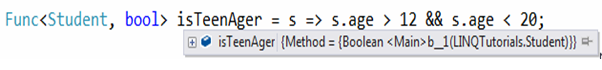
We have seen in the previous section that the lambda expression assigned to ***Func<T>*** compiles into executable code and the lambda expression assigned to ***Expression<TDelegate>*** type compiles into Expression tree.

Executable code excutes in the same application domain to process over in-memory collection. Enumerable static classes contain extension methods for in-memory collections that implements **IEnumerable<T>** interface e.g. List<T>, Dictionary<T>, etc. The Extension methods in an Enumerable class accept a predicate parameter of **Func** type delegate. For example, the **Where** extension method accepts **Func<TSource, bool> predicate**. It then compiles it into IL (Intermediate Language) to process over in-memory collections that are in the same AppDomain.

The following image shows Where extension method in Enumerable class includes Func delegate as a parameter:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-func-delegate-IEnumerable.png)Func delegate in Where

**Func** delegate is a raw executable code, so if you debug the code, you will find that the **Func** delegate will be represented as opaque code. You cannot see its parameters, return type and body:

[](http://www.tutorialsteacher.com/Content/images/linq/func-delegate-debug.png)Func delegate in debug mode

**Func** delegate is for in-memory collections because it will be processed in the same AppDomain, but what about remote LINQ query providers like LINQ-to-SQL, EntityFramework or other third party products that provides LINQ capabilities? How would they parse lambda expression that has been compiled into raw executable code to know about the parameters, return type of lambda expression and build runtime query to process further? The answer is **Expression tree**.

Expression<TDelegate> is compiled into a data structure called an expression tree.

If you debug the code, Expression delegate will be represented as shown below:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-expressiontree-debug.png)Expression Tree in debug mode

Now you can see the difference between a normal delegate and an Expression. An expression tree is transparent. You can retrieve a parameter, return type and body expression information from the expression, as below:

Example: Expression Tree in C#

Expression<Func<Student, bool>> isTeenAgerExpr = s => s.Age > 12 && s.Age < 20;

Console.WriteLine("Expression: {0}", isTeenAgerExpr );

Console.WriteLine("Expression Type: {0}", isTeenAgerExpr.NodeType);

var parameters = isTeenAgerExpr.Parameters;

foreach (var param in parameters)

{

Console.WriteLine("Parameter Name: {0}", param.Name);

Console.WriteLine("Parameter Type: {0}", param.Type.Name );

}

var bodyExpr = isTeenAgerExpr.Body as BinaryExpression;

Console.WriteLine("Left side of body expression: {0}", bodyExpr.Left);

Console.WriteLine("Binary Expression Type: {0}", bodyExpr.NodeType);

Console.WriteLine("Right side of body expression: {0}", bodyExpr.Right);

Console.WriteLine("Return Type: {0}", isTeenAgerExpr.ReturnType);

Output:

side of body expression: (s.Age < 20)Return Type: System.Boolean

LINQ query for LINQ-to-SQL or Entity Framework is not executed in the same app domain. For example, the following LINQ query for Entity Framework is never actually executed inside your program:

Example: LINQ query in C#

var query = from s in dbContext.Students

where s.Age >= 18

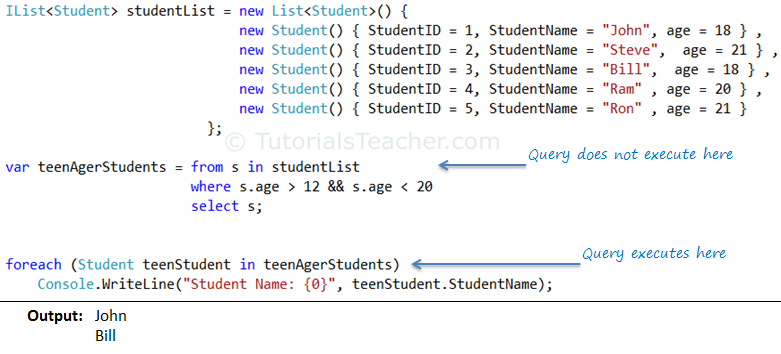
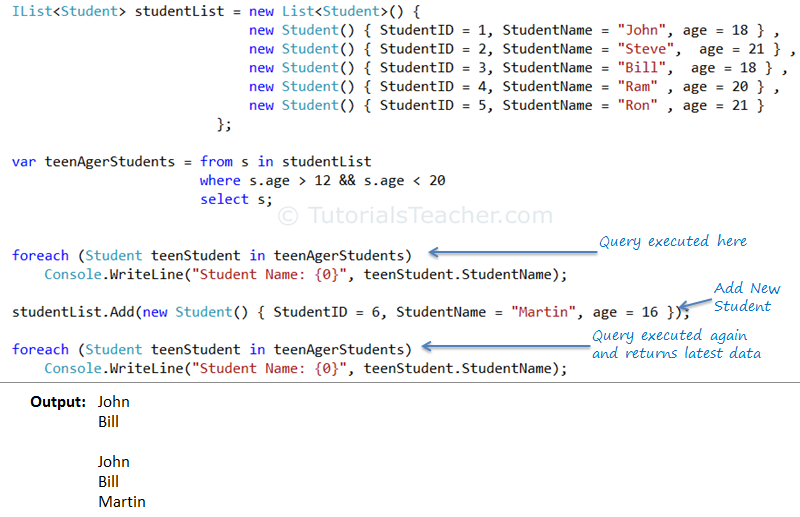
select s;

It is first translated into an SQL statement and then executed on the database server.

The code found in a query expression has to be translated into an SQL query that can be sent to another process as a string. For LINQ-to-SQL or Entity Frameworks, that process happens to be an SQL server database. It is obviously going to be much easier to translate a data structure such as an expression tree into SQL than it is to translate raw IL or executable code into SQL because, as you have seen, it is easy to retrieve information from an expression.

Expression trees were created for the task of converting code such as a query expression into a string that can be passed to some other process and executed there.

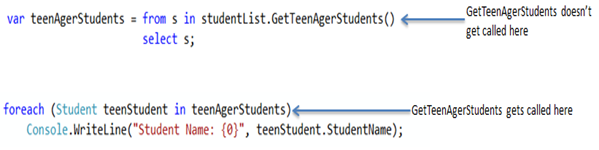
Queryable static class includes extension methods that accept a predicate parameter of Expression type. This predicate expression will be converted into an Expression Tree and then will be passed to the remote LINQ provider as a data structure so that the provider can build an appropriate query from the expression tree and execute the query.

1. [](http://www.tutorialsteacher.com/Content/images/linq/linq-expressiontree-process.png)Expression Tree Process
2. Deferred Execution of LINQ Query:
3. Deferred execution means that the evaluation of an expression is delayed until its realized value is actually required. It greatly improves performance by avoiding unnecessary execution.
4. Deferred execution is applicable on any in-memory collection as well as remote LINQ providers like LINQ-to-SQL, LINQ-to-Entities, LINQ-to-XML, etc.
5. Let's understand deferred execution using the following example:
6. [](http://www.tutorialsteacher.com/Content/images/linq/linq-deferred-execution-1.png)Deferred Execution
7. In the above example, you can see the query is materialized and executed when you iterate using the foreach loop. This is called deferred execution. LINQ processes the studentList collection when you actually access each object from the collection and do something with it.
8. **Deferred execution returns the latest data:**
9. To check whether deferred execution returns the latest data each time, add one more teen ager student after the foreach loop and check the teenager student list:
10. [](http://www.tutorialsteacher.com/Content/images/linq/linq-deferred-execution-2.png)Deferred Execution
11. As you can see, the second foreach loop executes the query again and returns the latest data. Deferred execution re-evaluates on each execution; this is called **lazy evaluation**. This is one of the major advantages of deferred execution: it always gives you the latest data.
12. Implementing deferred execution:
13. You can implement deferred execution for your custom extension methods for *IEnumerable* using the ***yield*** keyword of C#.
14. For example, you can implement custom extension method GetTeenAgerStudents for IEnumerable that returns a list of all students who are teenagers.
15. Example: Implimenting Deferred execution in C#
17. public static class EnumerableExtensionMethods
18. {
19. public static IEnumerable<Student> GetTeenAgerStudents(this IEnumerable<Student> source)
20. {
21. foreach (Student std in source)
22. {
23. Console.WriteLine("Accessing student {0}", std.StudentName);
24. if (std.age > 12 && std.age < 20)
25. yield return std;
26. }
27. }
28. }

31. Notice that we print the student name on the console whenever GetTeenAgerStudents() gets called.
32. You can now use this extension method as below:
33. C#:
35. IList<Student> studentList = new List<Student>() {
36. new Student() { StudentID = 1, StudentName = "John", age = 13 } ,
37. new Student() { StudentID = 2, StudentName = "Steve", age = 15 } ,
38. new Student() { StudentID = 3, StudentName = "Bill", age = 18 } ,
39. new Student() { StudentID = 4, StudentName = "Ram" , age = 12 } ,
40. new Student() { StudentID = 5, StudentName = "Ron" , age = 21 }
41. };
43. var teenAgerStudents = from s in studentList.GetTeenAgerStudents()
44. select s;
45. foreach (Student teenStudent in teenAgerStudents)
46. Console.WriteLine("Student Name: {0}", teenStudent.StudentName);

49. Output:

Accessing student Ron

1. As you can see from the output, GetTeenAgerStudents() is getting called when you iterate studentList using the foreach loop.
2. [](http://www.tutorialsteacher.com/Content/images/linq/linq-deferred-execution-3.png)Deferred Execution
3. So, in this way you can create custom methods using the ***yield*** keyword to get the advantage of deferred execution.

# Immediate Execution of LINQ Query:

Immediate execution is the reverse of deferred execution. It forces the LINQ query to execute and gets the result immediately. The 'To' [conversion operators](http://www.tutorialsteacher.com/linq/linq-conversion-operators) execute the given query and give the result immediately.

### Method Syntax:

In the following example, ToList() extension method executes the query immediately and returns the result.

C#: Immediate Execution

IList<Student> teenAgerStudents =

studentList.Where(s => s.age > 12 && s.age < 20).ToList();

VB.Net:Immediate Execution

Dim teenAgerStudents As IList(Of Student) =

studentList.Where(Function(s) s.Age > 12 And s.Age < 20).ToList()

### Query Syntax:

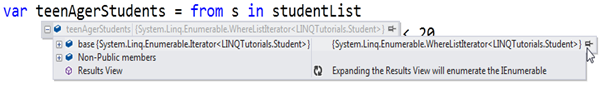
C#:

var teenAgerStudents = from s in studentList

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

select s;

The above query will not execute immediately. You won't find any result as shown below:

[](http://www.tutorialsteacher.com/Content/images/linq/linq-immedite-execution.png)Immediate Execution

Query Syntax doesn't support 'To' operators but can use ToList(), ToArray() or ToDictionary() for immediate execution as below:

C#:

IList<Student> teenAgerStudents = (from s in studentList

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

select s).ToList();

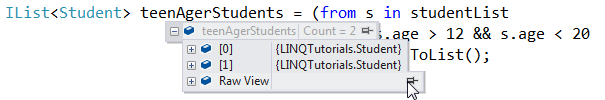
VB.Net:

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

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

Select s).ToList()

You can see the result in the teenAgerStudents collection, as below:

1. [](http://www.tutorialsteacher.com/Content/images/linq/linq-immedite-execution-2.png)Immediate Execution

let keyword:

The 'let' keyword is useful in query syntax. It projects a new range variable, allows re-use of the expression and makes the query more readable.

For example, you can compare string values and select the lowercase string value as shown below:

Example: let in LINQ query - 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 = "Ron" , Age = 21 }

};

var lowercaseStudentNames = from s in studentList

where s.StudentName.ToLower().StartsWith("r")

select s.StudentName.ToLower();

As you can see, the ToLower() method is used multiple times in the above query. The following example use 'let' to introduce new variable 'lowercaseStudentName' that will be then used in every where. Thus, **let** keyword to make the query more readable.

Example: let keyword in C#

var lowercaseStudentNames = from s in studentList

let lowercaseStudentName = s.StudentName.ToLower()

where lowercaseStudentName.StartsWith("r")

select lowercaseStudentName;

foreach (var name in lowercaseStudentNames)

Console.WriteLine(name);

Example: let keyword in VB.Net

Dim lowercaseStudentNames = From s In studentList

Let lowercaseStudentName = s.StudentName.ToLower()

Where lowercaseStudentName.StartsWith("r")

Select lowercaseStudentName;

Output:

# ram  roninto keyword:

We have already used the 'into' keyword in grouping. You can also use the 'into' keyword to continue a query after a **select** clause.

Example: into keyword in LINQ

var teenAgerStudents = from s in studentList

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

select s

into teenStudents

where teenStudents.StudentName.StartsWith("B")

select teenStudents;

In the above query, the 'into' keyword introduced a new range variable *teenStudents*, so the first range variable **s** goes out of scope. You can write a further query after the into keyword using a new range variable.

The 'into' keyword in VB.Net used for grouping purposes.

Example: into keyword in LINQ VB.Net

Dim groupQuery = From s In studentList

Group By s.Age Into Group

Sample LINQ Queries:

In this section, you will learn some complex LINQ queries. We will use the following Student and Standard collection for our queries.

Sample Collections:

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

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

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

};

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"}

};

Multiple Select and where operator:

Example: Multiple Select and where operator

var studentNames = studentList.Where(s => s.Age > 18)

.Select(s => s)

.Where(st => st.StandardID > 0)

.Select(s => s.StudentName);

Output:

am

The following query returns Enumerable of anonymous object that has only StudentName property:

Example: LINQ query returns collection of anonymous objects

var teenStudentsName = from s in studentList

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

select new { StudentName = s.StudentName };

teenStudentsName.ToList().ForEach(s => Console.WriteLine(s.StudentName));

Output:

Group By:

The following query returns list students group by StandardID:

Example: LINQ GroupBy query - C#

var studentsGroupByStandard = from s in studentList

group s by s.StandardID into sg

orderby sg.Key

select new { sg.Key, sg };

foreach (var group in studentsGroupByStandard)

{

Console.WriteLine("StandardID {0}:", group.Key);

group.sg.ToList().ForEach(st => Console.WriteLine(st.StudentName ));

}

Output:

am

The output includes Ron who doesn't have any StandardID. So Ron falls under StandardID 0.

To remove a student who doesn't have a StandardID, use a where operator before the group operator:

Example: LINQ GroupBy query - C#

var studentsGroupByStandard = from s in studentList

where s.StandardID > 0

group s by s.StandardID into sg

orderby sg.Key

select new { sg.Key, sg };

Output:

Left outer join:

Use left outer join to display students under each standard. Display the standard name even if there is no student assigned to that standard.

Example: LINQ Left Outer Join - C#

var studentsGroup = from stad in standardList

join s in studentList

on stad.StandardID equals s.StandardID

into sg

select new {

StandardName = stad.StandardName,

Students = sg

};

foreach (var group in studentsGroup)

{

Console.WriteLine(group.StandardName);

group.Students.ToList().ForEach(st => Console.WriteLine(st.StudentName));

}

Output:

ndard 1: 

In the following example of group by query, we sort the group and select only StudentName:

Example: LINQ Left Outer Join - C#

var studentsWithStandard = from stad in standardList

join s in studentList

on stad.StandardID equals s.StandardID

into sg

from std\_grp in sg

orderby stad.StandardName, std\_grp.StudentName

select new {

StudentName = std\_grp.StudentName,

StandardName = stad.StandardName };

foreach (var group in studentsWithStandard)

{

Console.WriteLine("{0} is in {1}", group.StudentName, group.StandardName);

}

Output:

Sorting:

The following query returns list of students by ascending order of StandardID and Age.

Example: Sorting

var sortedStudents = from s in studentList

orderby s.StandardID, s.age

select new {

StudentName = s.StudentName,

Age = s.age,

StandardID = s.StandardID };

sortedStudents.ToList().ForEach(s => Console.WriteLine("Student Name: {0}, Age: {1}, StandardID: {2}", s.StudentName, s.Age , s.StandardID));

Output:

Student Name: Ron, Age: 21, StandardID: 0   
Inner Join:

Example: LINQ Inner join - C#

var studentWithStandard = from s in studentList

join stad in standardList

on s.StandardID equals stad.StandardID

select new {

StudentName = s.StudentName,

StandardName = stad.StandardName

};

studentWithStandard.ToList().ForEach(s => Console.WriteLine("{0} is in {1}", s.StudentName, s.StandardName ));

Output:

John is in Standard 1   
Nested Query:

C#:

var nestedQueries = from s in studentList

where s.age > 18 && s.StandardID ==

(from std in standardList

where std.StandardName == "Standard 1"

select std.StandardID).FirstOrDefault()

select s;

nestedQueries.ToList().ForEach(s => Console.WriteLine(s.StudentName));