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

## Advantages of LINQ

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

Example: LINQ Query to Array

// Data source

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

// LINQ Query

var myLinqQuery = from name in names

where name.Contains('a')

select name;

// Query execution

foreach(var name in myLinqQuery)

Console.Write(name + " ");

Why LINQ?

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

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

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

class Student

{

public int StudentID { get; set; }

public String StudentName { get; set; }

public int Age { get; set; }

}

class Program

{

static void Main(string[] args)

{

Student[] studentArray = {

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

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

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

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

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

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

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

};

Student[] students = new Student[10];

int i = 0;

foreach (Student std in studentArray)

{

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

{

students[i] = std;

i++;

}

}

}

}

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

delegate bool FindStudent(Student std);

class StudentExtension

{

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

{

int i=0;

Student[] result = new Student[10];

foreach (Student std in stdArray)

if (del(std))

{

result[i] = std;

i++;

}

return result;

}

}

class Program

{

static void Main(string[] args)

{

Student[] studentArray = {

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

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

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

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

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

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

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

};

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

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

});

}

}

}

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

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

return std.StudentID == 5;

});

//Also, use another criteria using same delegate

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

return std.StudentName == "Bill";

});

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

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

Example: LINQ

class Program

{

static void Main(string[] args)

{

Student[] studentArray = {

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

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

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

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

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

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

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

};

// Use LINQ to find teenager students

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

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

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

// Use LINQ to find student whose StudentID is 5

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

}

}

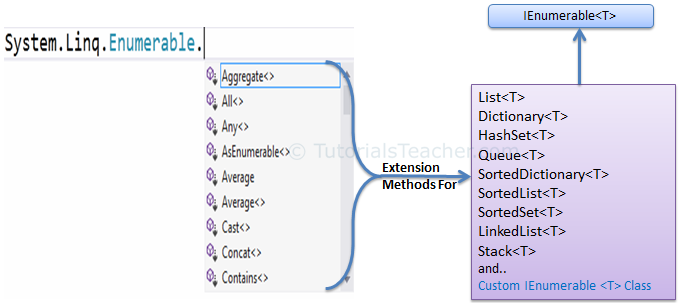
# LINQ API

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

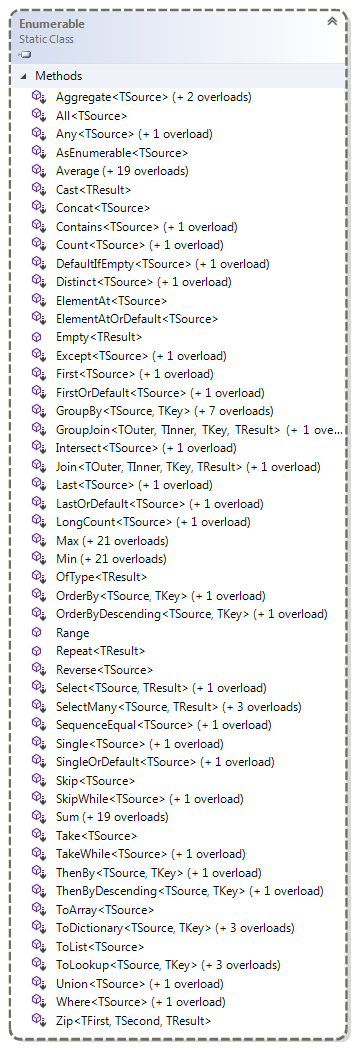
## Enumerable

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

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

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

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

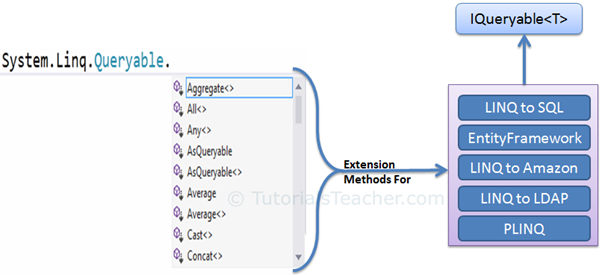
[](https://www.tutorialsteacher.com/Content/images/linq/Enumerable.png)Enumerable Class

## Queryable

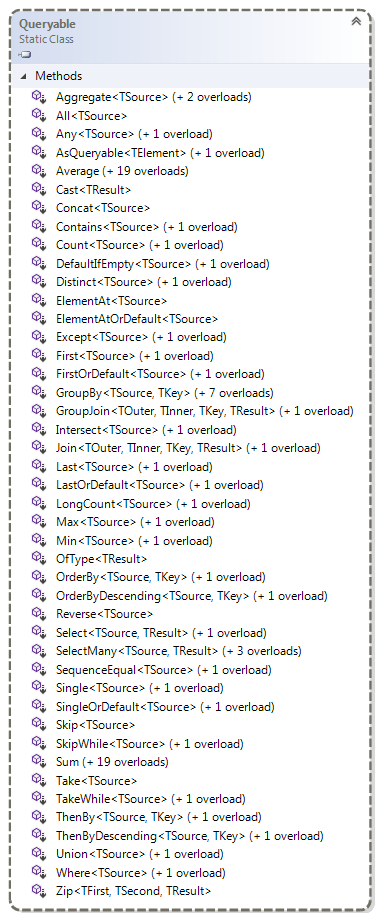
The [Queryable](https://msdn.microsoft.com/en-us/library/system.linq.queryable(v=vs.110).aspx" \o "Queryable class members" \t "_blank) class includes extension methods for classes that implement [IQueryable<t>](https://msdn.microsoft.com/en-us/library/vstudio/bb351562(v=vs.100).aspx" \o "IQueryable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

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

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

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

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

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

LINQ Query Syntax

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

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

## Query Syntax

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

LINQ Query Syntax:

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

<Standard Query Operators> *<lambda expression>*

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

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

Example: LINQ Query Syntax in C#

// string collection

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

"C# Tutorials",

"VB.NET Tutorials",

"Learn C++",

"MVC Tutorials" ,

"Java"

};

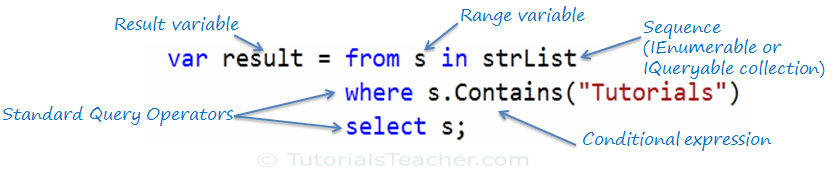
// LINQ Query Syntax

var result = from s in stringList

where s.Contains("Tutorials")

select s;

The following figure shows the structure of LINQ query syntax.

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

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

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

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

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

Example: LINQ Query Syntax in C#

// Student collection

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

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

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

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

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

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

};

// LINQ Query Syntax to find out teenager students

var teenAgerStudent = from s in studentList

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

select s;

LINQ Method Syntax

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

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

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

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

Example: LINQ Method Syntax in C#

// string collection

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

"C# Tutorials",

"VB.NET Tutorials",

"Learn C++",

"MVC Tutorials" ,

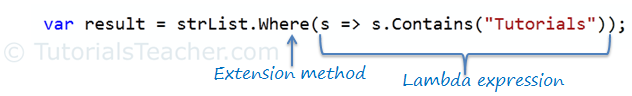
"Java"

};

// LINQ Query Syntax

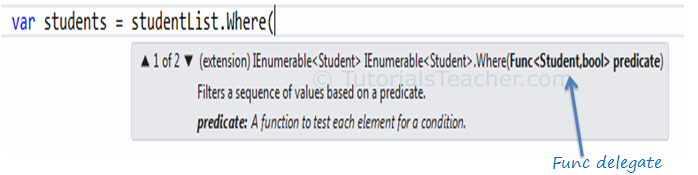
var result = stringList.Where(s => s.Contains("Tutorials"));

The following figure illustrates the structure of LINQ method syntax.

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

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

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

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

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

Example: Method Syntax in C#

// Student collection

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

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

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

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

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

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

};

// LINQ Method Syntax to find out teenager students

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

.ToList<Student>();

Example: Method Syntax in VB.Net

// Student collection

Dim studentList = New List(Of Student) From {

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

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

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

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

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

}

// LINQ Method Syntax to find out teenager students

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

.ToList()

# Anatomy of the Lambda Expression

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

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

Example: Anonymous Method in C#

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

Example: Anonymous method in VB.Net

Dim isStudentTeenAger = Function(s As Student) As Boolean

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

End Function

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

Example: Lambda Expression in C#

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

Example: Lambda Expression in VB.Net

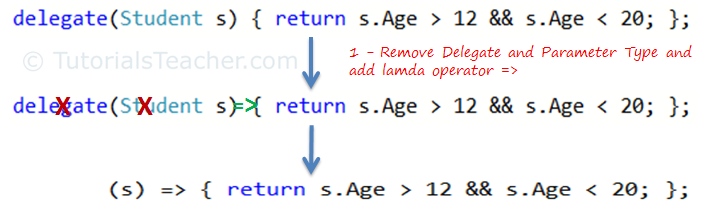
Function(s) s.Age > 12 And s.Age < 20

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

Example: Anonymous method in C#

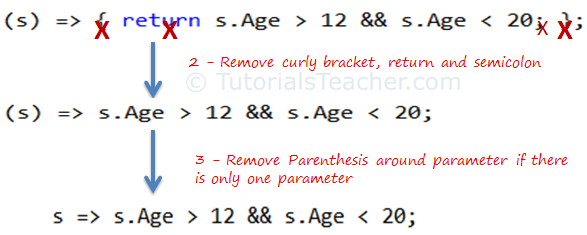
delegate(Student s) { return s.Age > 12 && s.Age < 20; };

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

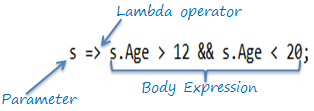
[](https://www.tutorialsteacher.com/Content/images/linq/lambda-expression-1.png)Lambda Expression from Anonymous Method

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

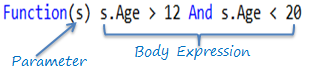
Also, we can remove parenthesis (), if we have only one parameter.

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

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

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

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

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

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

 Note:

VB.Net doesn't support lambda operator =>

## Lambda Expression with Multiple Parameters

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

Example: Specify Multiple Parameters in Lambda Expression C#

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

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

Example: Specify Parameter Type

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

Example: Specify Multiple Parameters in Lambda Expression VB.Net

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

## Lambda Expression without Parameter

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

Example: Lambda Expression without Parameter

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

## Multiple Statements in Lambda Expression Body

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

Example: Multi Statements Lambda expression C#

(s, youngAge) =>

**{**

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

Return s.Age >= youngAge;

**}**

Example: Multi Statements Lambda Expression VB.Net

Function(s , youngAge)

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

Return s.Age >= youngAge

End Function

## Declare Local Variable in Lambda Expression Body

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

Example: Local Variable in Lambda expression C#

s =>

{

**int youngAge = 18;**

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

return s.Age >= youngAge;

}

Example: Local Variable in Lambda Expression VB.Net

Function(s)

**Dim youngAge As Integer = 18**

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

Return s.Age >= youngAge

End Function

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

## Assign Lambda Expression to Delegate

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

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

Example: Lambda Expression Assigned to Func Delegate C#

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

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

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

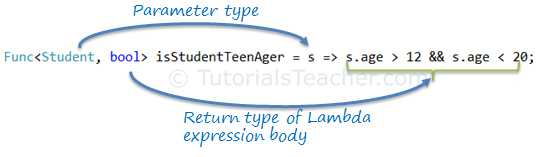
Example: Lamda Expression Assigned to Func Delegate VB.Net

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

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

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

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

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

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

bool isStudentTeenAger(Student s)

{

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

}

## Action Delegate

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

Example: Lamda Expression Assigned to Action Delegate C#

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

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

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

Example: Lamda Expression Assigned to Action Delegate VB.Net

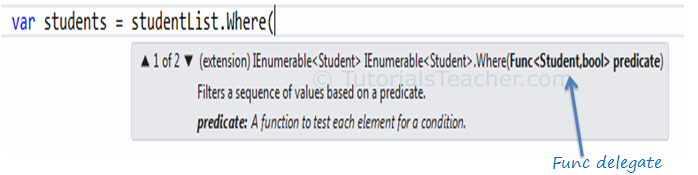
Dim printStudentDetail As Action(Of Student) = Sub(s) Console.WriteLine("Name: {0}, Age: {1} ", s.StudentName, s.Age)

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

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

## Lambda Expression in LINQ Query

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

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

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

Example: Func Delegate in LINQ Method Syntax

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

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

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

Example: Func Delegate in LINQ Query Syntax

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

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

var teenStudents = from s in studentList

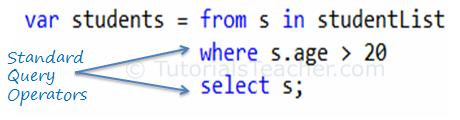
where isStudentTeenAger(s)

select s;

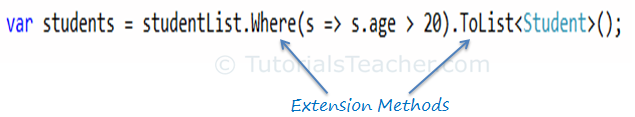
# Standard Query Operators

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

## Standard Query Operators in Query Syntax

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

## Standard Query Operators in Method Syntax

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

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

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

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