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>*

<https://www.tutorialsteacher.com/Content/images/linq/linq-query-syntax.png>

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).

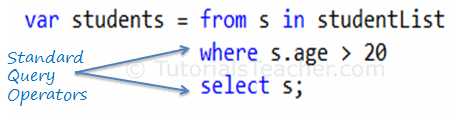
LINQ query syntax always ends with a Select or Group clause. The Select clause is used to shape the data. You can select the whole object as it is or only some properties of it.

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

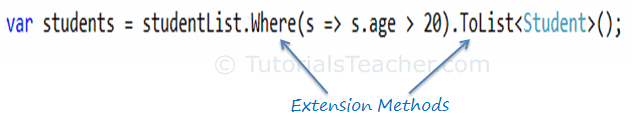
# **Standard Query Operators**

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 |

Learn each Standard Query Operators in the next sections.

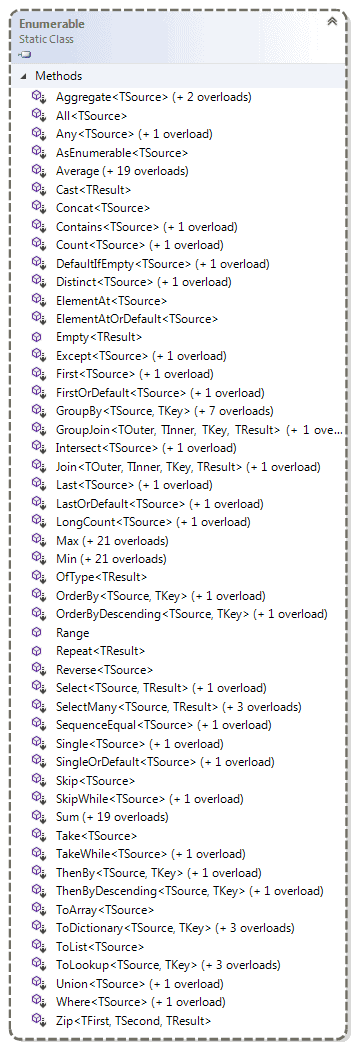
# **LINQ API in .NET**

We can write LINQ queries for the classes that implement [IEnumerable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ienumerable-1?view=netframework-4.8" \o "IEnumerable Members" \t "_blank) or [IQueryable<T>](https://docs.microsoft.com/en-us/dotnet/api/system.linq.iqueryable-1?view=netframework-4.8" \o "IQueryable Members" \t "_blank) interface. The *[System.Linq](https://docs.microsoft.com/en-us/dotnet/api/system.linq?view=netframework-4.8" \o "system.linq" \t "_blank)* namespace includes the following classes and interfaces require for LINQ queries.

## Enumerable

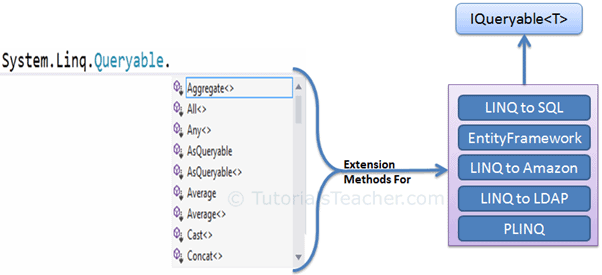
The [Enumerable](https://msdn.microsoft.com/en-us/library/system.linq.enumerable(v=vs.110).aspx)) class includes extension methods for the classes that implement IEnumerable<T> interface, for example all the built-in collection classes implement IEnumerable<T> interface and so we can write LINQ queries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

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

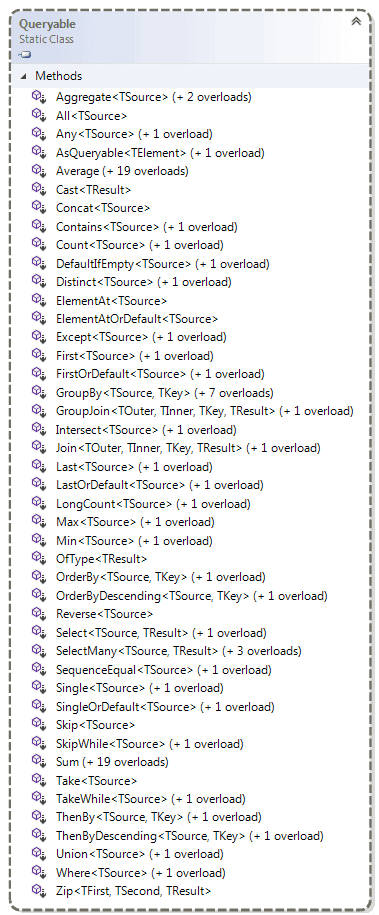
[](https://www.tutorialsteacher.com/Content/images/linq/Enumerable.png)

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.

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)

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

[](https://www.tutorialsteacher.com/Content/images/linq/queryable.png)Queryable 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 the 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 the IQueryable<T> interface.
6. Remote query provider implements e.g. Linq-to-SQL, LINQ-to-Amazon etc.

# **Filtering Operator - Where**

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

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

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

## Where

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

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

Where method Overloads:

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

Func<TSource, bool> predicate);

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

Func<TSource, int, bool> predicate);

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

# **Filtering Operator - OfType**

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

### **OfType in Query Syntax**

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

Example: OfType operator in C#

IList mixedList = new ArrayList();

mixedList.Add(0);

mixedList.Add("One");

mixedList.Add("Two");

mixedList.Add(3);

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

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

select s;

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

select s;

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

Example: OfType operator in VB.Net:

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

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

One  
Two  
0  
3  
Bill

### **OfType in Method Syntax**

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

Example: OfType in C#

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

Example: OfType in VB.Net

Dim stringResult = mixedList.OfType(Of String)

stringResult would contain following elements.

One  
Two

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

 Points to Remember :

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

# **Sorting Operators: OrderBy & OrderByDescending**

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

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

## OrderBy

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

Example: OrderBy in Query Syntax C#

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

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

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

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

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

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

};

var orderByResult = from s in studentList

orderby s.StudentName

select s;

var orderByDescendingResult = from s in studentList

orderby s.StudentName descending

select s;

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

Example: OrderBy in Query Syntax VB.Net

Dim orderByResult = From s In studentList

Order By s.StudentName

Select s

Dim orderByDescendingResult = From s In studentList

Order By s.StudentName Descending

Select s

## OrderBy in Method Syntax

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

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

OrderBy Overload Methods:

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

Func<TSource, TKey> keySelector);

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

Func<TSource, TKey> keySelector,

IComparer<TKey> comparer);

## Multiple Sorting

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

Example: Multiple sorting in Query syntax C#

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

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

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

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

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

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

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

};

var orderByResult = from s in studentList

orderby s.StudentName, s.Age

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

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

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

# **Sorting Operators: ThenBy & ThenByDescending**

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

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

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

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

Example: ThenBy & ThenByDescending

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

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

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

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

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

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

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

};

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

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

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

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

### **GroupBy in Query Syntax**

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

Example: GroupBy in Query syntax C#

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

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

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

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

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

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

};

var groupedResult = from s in studentList

group s by s.Age;

//iterate each group

foreach (var ageGroup in groupedResult)

{

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

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

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

}

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

# **Joining Operator: Join**

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

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

## Join

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

### **Join in Method Syntax**

The Join extension method has two overloads as shown below.

Join Overload Methods:

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

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

Func<TInner, TKey> innerKeySelector,

Func<TOuter, TInner, TResult> resultSelector);

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

IEnumerable<TInner> inner,

Func<TOuter, TKey> outerKeySelector,

Func<TInner, TKey> innerKeySelector,

Func<TOuter, TInner, TResult> resultSelector,

IEqualityComparer<TKey> comparer);

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

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

Example: Join operator C#

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

"One",

"Two",

"Three",

"Four"

};

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

"One",

"Two",

"Five",

"Six"

};

var innerJoin = strList1.Join(strList2,

str1 => str1,

str2 => str2,

(str1, str2) => str1);