# LINQ - 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 (Language Integrated Query) is uniform query syntax in C# and VB.NET to retrieve data from different sources and formats. It is integrated in C# or VB, thereby eliminating the mismatch between programming languages and databases, as well as providing a single querying interface for different types of data sources. LINQ queries return results as objects. It enables you to uses object-oriented approach on the result set and not to worry about transforming different formats of results into objects.



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

// 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 + " ");

# mida tähendab märk: =>

# 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 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:

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);

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;

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

John  
Bill  
Ron

**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.

var filteredResult = studentList.Where(s => s.Age > 12 && 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.

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.

# OfType - Filtering Operator. 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;

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>();

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.

# ThenBy - Sorting Operator. The ThenBy and ThenByDescending extension methods are used for sorting on multiple fields.

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 following elements after sorting:

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

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

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

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

Example: ThenBy & ThenByDescending VB.Net

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

.ThenBy(Function(s) s.Age)

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

.ThenByDescending(Function(s) s.Age)

Points to Remember :

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

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

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

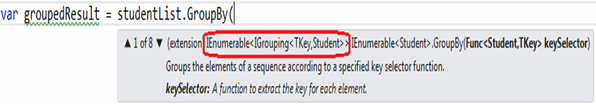
GroupBy

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

 Note:

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

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

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

GroupBy in Query Syntax

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

Example: GroupBy in Query syntax C#

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

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

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

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

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

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

};

var groupedResult = from s in studentList

group s by s.Age;

//iterate each group

foreach (var ageGroup in groupedResult)

{

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

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

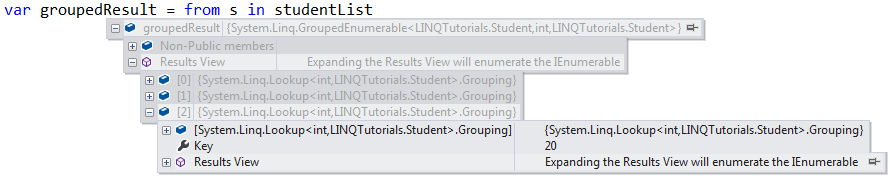
Console.WriteLine("Student Name: {0}", s.StudentName);

}

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.

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

**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);

}

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);

}

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.

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

| 1. 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);

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

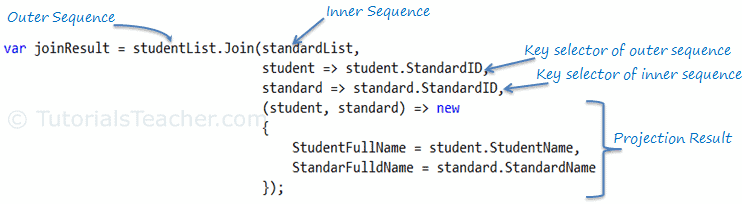
{

StudentName = student.StudentName,

StandardName = standard.StandardName

});

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

[](https://www.tutorialsteacher.com/Content/images/linq/linq-join-operator.png)**join operator**

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

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

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

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

};

Output:

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

Points to Remember :

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

select ...

# GroupJoin - 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.

GroupJoin requires same parameters as Join. GroupJoin has following two overload methods:

GroupJoin Overload Methods:

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);

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);

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.

Now, let's understand GroupJoin 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; }

}

Consider the following GroupJoin query example.

Example: GroupJoin in Method syntax 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 groupJoin = standardList.GroupJoin(studentList, //inner sequence

std => std.StandardID, //outerKeySelector

s => s.StandardID, //innerKeySelector

(std, studentsGroup) => new // resultSelector

{

Students = studentsGroup,

StandarFulldName = std.StandardName

});

foreach (var item in groupJoin)

{

Console.WriteLine(item.StandarFulldName );

foreach(var stud in item.Students)

Console.WriteLine(stud.StudentName);

}

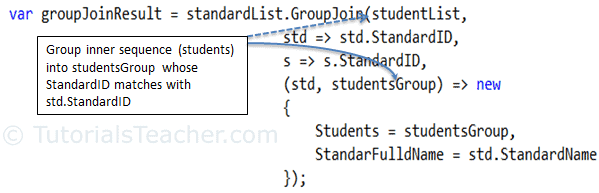
Output:

Standard 1:  
John,  
Moin,  
Standard 2:  
Bill,  
Ram,  
Standard 3:

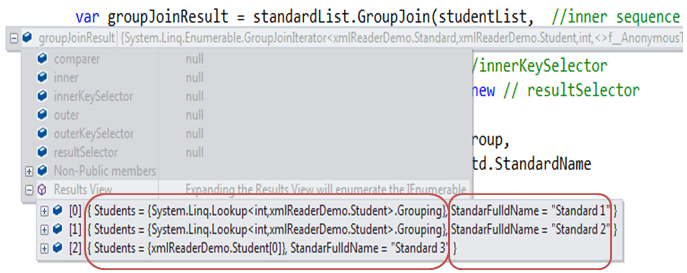
In the above example of GroupJoin query, standardList is the outer sequence, because the query starts from it. The first parameter in GroupJoin method is to specify the inner sequence, which is studentList in the above example. The second and third parameters of the GroupJoin() method are to specify 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.

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.

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.

[](https://www.tutorialsteacher.com/Content/images/linq/linq-groupjoin.png)Grouping Operator - GroupJoin

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.

[](https://www.tutorialsteacher.com/Content/images/linq/groupJoin-result.png)GroupJoin Result in Debug View

You can access the result using a 'foreach' loop. Each element will have the StandardFullName & Students property, where Students will be a collection.

Example: Access GroupJoin Result in C#

foreach (var item in groupJoinResult)

{

Console.WriteLine(item.StandarFulldName );

foreach(var stud in item.Students)

Console.WriteLine(stud.StudentName);

}

Output:

Standard 1:  
John,  
Moin,  
Standard 2:  
Bill,  
Ram,  
Standard 3:

GroupJoin in Query Syntax

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.

Syntax: GroupJoin in Query Syntax

from ... in outerSequence

join ... in innerSequence

on outerKey equals innerKey

into groupedCollection

select ...

The following example demonstrates the GroupJoin in query syntax.

Example: GroupJoin 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 groupJoin = from std in standardList

join s in studentList

on std.StandardID equals s.StandardID

into studentGroup

select new {

Students = studentGroup ,

StandardName = std.StandardName

};

foreach (var item in groupJoin)

{

Console.WriteLine(item.StandarFulldName );

foreach(var stud in item.Students)

Console.WriteLine(stud.StudentName);

}

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.

 Note:

Use of the **equals** operator to match key selector. == is not valid.

# 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.

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](https://www.tutorialsteacher.com/csharp/csharp-anonymous-type) containing the Name and Age property.

Example: Select 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);

**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 } ,

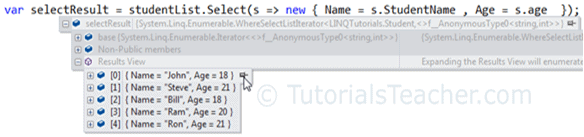
new Student() { StudentID = 5, StudentName = "Ron" , Age = 21 }

};

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.

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

Select Many

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

# All, Any - 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);

# 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);

Output:

True

# 11. 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() 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

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**

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.

# Aggregate - 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](https://www.tutorialsteacher.com/linq/linq-aggregation-operator-aggregate#aggregate) | Performs a custom aggregation operation on the values in the collection. |
| [Average](https://www.tutorialsteacher.com/linq/linq-aggregation-operator-average) | calculates the average of the numeric items in the collection. |
| [Count](https://www.tutorialsteacher.com/linq/linq-aggregation-operator-count) | Counts the elements in a collection. |
| LongCount | Counts the elements in a collection. |
| [Max](https://www.tutorialsteacher.com/linq/linq-aggregation-operator-max) | Finds the largest value in the collection. |
| Min | Finds the smallest value in the collection. |
| [Sum](https://www.tutorialsteacher.com/linq/linq-aggregation-operator-sum) | Calculates sum of the values in the collection. |

**Aggregate**

The Aggregate method performs an accumulate operation. Aggregate extension method has the following overload methods:

Aggregate() Overloads:

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

Func<TSource, TSource, TSource> func);

public static TAccumulate Aggregate<TSource, TAccumulate>(this IEnumerable<TSource> source,

TAccumulate seed,

Func<TAccumulate, TSource, TAccumulate> func);

public static TResult Aggregate<TSource, TAccumulate, TResult>(this IEnumerable<TSource> source,

TAccumulate seed,

Func<TAccumulate, TSource, TAccumulate> func,

Func<TAccumulate, TResult> resultSelector);

The following example demonstrates Aggregate method that returns comma seperated elements of the string list.

Example: Aggregate in Method Syntax C#

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

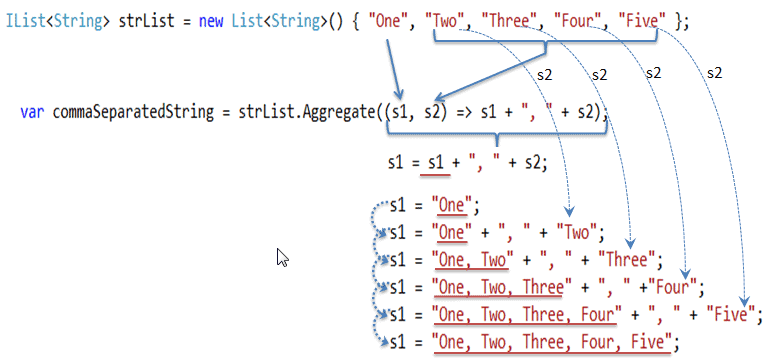
var commaSeperatedString = strList.Aggregate((s1, s2) => s1 + ", " + s2);

Console.WriteLine(commaSeperatedString);

Output:

One, Two, Three, Four, Five

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.

[](https://www.tutorialsteacher.com/Content/images/linq/linq-aggregate-1.png)Aggregate extension method

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.

**Aggregate Method with Seed Value**

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**); .

The following example uses string as a seed value in the Aggregate extension method.

Example: Aggregate with Seed Value 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 }

};

string commaSeparatedStudentNames = studentList.Aggregate<Student, string>(

"Student Names: ", // seed value

(str, s) => str += s.StudentName + "," );

Console.WriteLine(commaSeparatedStudentNames);

Output:

Student Names: John, Moin, Bill, Ram, Ron,

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.

The following example use Aggregate operator to add the age of all the students.

Example: Aggregate with Seed Value 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 }

};

int SumOfStudentsAge = studentList.Aggregate<Student, int>(0,

(totalAge, s) => totalAge += s.Age );

**Aggregate Method with Result Selector**

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.

Example: Aggregate with Result Selector 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 }

};

string commaSeparatedStudentNames = studentList.Aggregate<Student, string,string>(

String.Empty, // seed value

(str, s) => str += s.StudentName + ",", // returns result using seed value, String.Empty goes to lambda expression as str

str => str.Substring(0,str.Length - 1 )); // result selector that removes last comma

Console.WriteLine(commaSeparatedStudentNames);

Output:

John, Moin, Bill, Ram, Ron

# Average - 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.

The following example demonstrate Agerage method that returns average value of all the integers in the collection.

Example: Average Method C#

IList<int> intList = new List<int>>() { 10, 20, 30 };

var avg = intList.Average();

Console.WriteLine("Average: {0}", avg);

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.

Example: Average 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 avgAge = studentList.Average(s => s.Age);

Console.WriteLine("Average Age of Student: {0}", avgAge);

Output:

Average Age of Student: 17.4

# 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() Overloads:

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 demonstrates Count() on primitive collection.

Example: Count() - C#

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

var totalElements = intList.Count();

Console.WriteLine("Total Elements: {0}", totalElements);

var evenElements = intList.Count(i => i%2 == 0);

Console.WriteLine("Even Elements: {0}", evenElements);

Output:

Total Elements: 5  
Even Elements: 3

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

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 totalStudents = studentList.Count();

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

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

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

Output:

Total Students: 5  
Number of Adult 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();

# Max - The Max() method 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 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);

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:

Example: Max() in C#

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

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.

# 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);

# 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:

1st Element in intList: 10  
1st Element in strList: One  
2nd Element in intList: 21  
2nd Element in strList: Two  
3rd Element in intList: 30  
3rd Element in strList: Three  
10th Element in intList: 0 - default int value  
10th Element in strList: - default string value (null)  
intList.ElementAt(9) throws an exception: Index out of range  
-------------------------------------------------------------  
Run-time 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.

# 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.

First() Á FirstOrDefault() Overloads:

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:

1st Element in intList: 7  
1st Even Element in intList: 10  
1st Element in strList:  
emptyList.First() throws an InvalidOperationException  
-------------------------------------------------------------  
Run-time 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:

1st Element in intList: 7  
1st Even Element in intList: 10  
1st Element in strList:  
1st 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 element

# Last, LastOrDefault - 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.

| 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() Á LastOrDefault() Overloads:

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:

Last Element in intList: 87  
Last Even Element in intList: 50  
Last Element in strList: Five  
emptyList.Last() throws an InvalidOperationException  
-------------------------------------------------------------  
Run-time 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:

Last Element in intList: 7  
Last Even Element in intList: 10  
Last Element in strList:  
Last 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 exception: Sequence contains no matching element

# Single, SingleOrDefault - 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.

| 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() Á SingleOrDefault() Overloads:

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());

https://www.tutorialsteacher.com/Content/images/bulb-glow.png 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

# 21. SequenceEquel - 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());

https://www.tutorialsteacher.com/Content/images/bulb-glow.png 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.

# Concat - The Concat() method appends two sequences of the same type and returns a new sequence (collection).

Example: Concat in C#

IList<string> collection1 = new List<string>() { "One", "Two", "Three" };

IList<string> collection2 = new List<string>() { "Five", "Six"};

var collection3 = collection1.Concat(collection2);

foreach (string str in collection3)

Console.WriteLine(str);

Output:

One  
Two  
Three  
Five  
Six

Example: Concat in C#

IList<int> collection1 = new List<int>() { 1, 2, 3 };

IList<int> collection2 = new List<int>() { 4, 5, 6 };

var collection3 = collection1.Concat(collection2);

foreach (int i in collection3)

Console.WriteLine(i);

Output:

1  
2  
3  
4  
5  
6

Concat operator is not supported in query syntax in C# or VB.Net.

# 23. DefaultEmpty - 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:

Count: 1  
Value:  
Count: 1  
Value: 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:

Count: 1  
Value: 0  
Count: 1  
Value: 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:

Count: 1  
Student ID:  
Count: 1  
Student ID: 0

# 24. 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[]  
Count: 0  
Type: Student[]  
Count: 0

**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:

Total Count: 10  
Value at index 0 : 10  
Value at index 1 : 11  
Value at index 2 : 12  
Value at index 3 : 13  
Value at index 4 : 14  
Value at index 5 : 15  
Value at index 6 : 16  
Value at index 7 : 17  
Value at index 8 : 18  
Value at index 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:

Total Count: 10  
Value at index 0: 10  
Value at index 1: 10  
Value at index 2: 10  
Value at index 3: 10  
Value at index 4: 10  
Value at index 5: 10  
Value at index 6: 10  
Value at index 7: 10  
Value at index 8: 10  
Value at index 9: 10

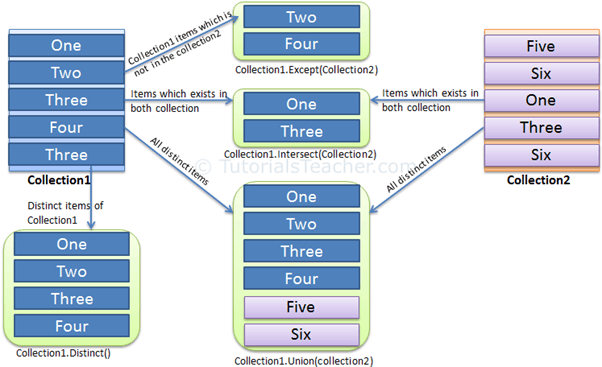
In the above example, Enumerable.Repeat<int>(10, 10) creates collection with 100 integer type elements with the repeated value of 10. First parameter specifies the values of all the elements and second parameter specifies the number of elements to create.

# 25. Distinct

The following table lists all Set operators available in LINQ.

| Set Operators | Usage |
| --- | --- |
| [Distinct](https://www.tutorialsteacher.com/linq/linq-set-operators-distinct#distinct) | Returns distinct values from a collection. |
| [Except](https://www.tutorialsteacher.com/linq/linq-set-operators-except) | Returns the difference between two sequences, which means the elements of one collection that do not appear in the second collection. |
| [Intersect](https://www.tutorialsteacher.com/linq/linq-set-operators-intersect) | Returns the intersection of two sequences, which means elements that appear in both the collections. |
| [Union](https://www.tutorialsteacher.com/linq/linq-set-operators-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:

[](https://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:

One  
Two  
Three  
1  
2  
3  
4  
5

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  
Steve  
Bill  
Ron

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

# 26. 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)

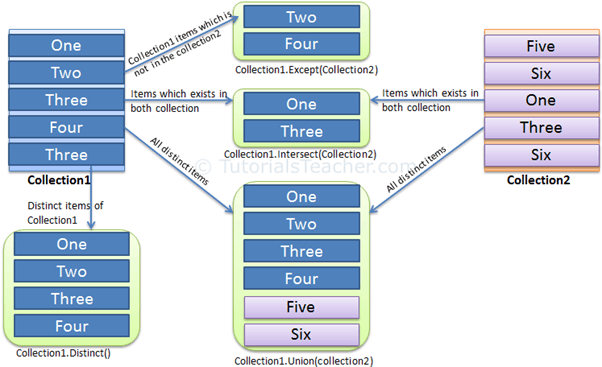
Console.WriteLine(std.StudentName);

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:

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

# 27. 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)

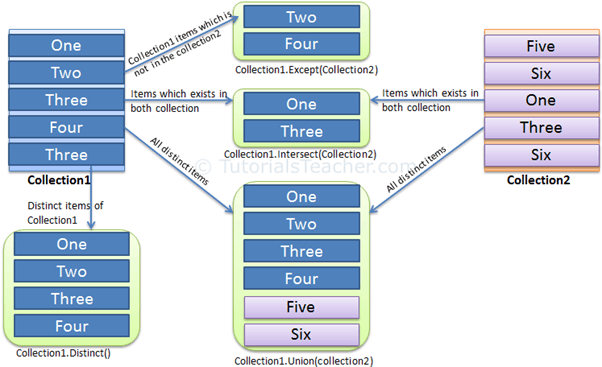
Console.WriteLine(std.StudentName);

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:

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

# 28. 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);

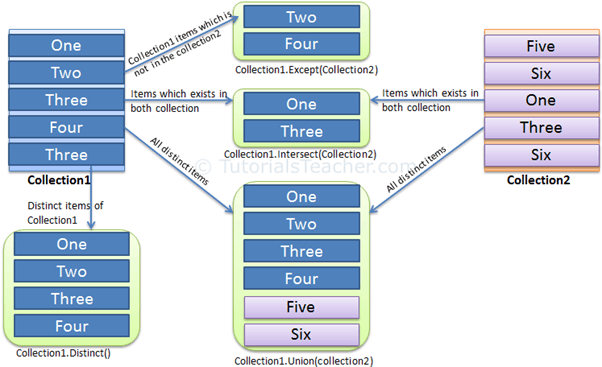
Output:

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:

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

# 29. Skip, SkipWhile - Partitioning operators split the sequence (collection) into two parts and return one of the parts.

| Method | Description |
| --- | --- |
| [Skip](https://www.tutorialsteacher.com/linq/linq-partitioning-operators-skip-skipwhile#skip) | Skips elements up to a specified position starting from the first element in a sequence. |
| [SkipWhile](https://www.tutorialsteacher.com/linq/linq-partitioning-operators-skip-skipwhile#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](https://www.tutorialsteacher.com/linq/linq-partitioning-operators-take-takewhile) | 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  
Four  
Five

**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.

**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  
One  
Two  
Four  
Five  
Six

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.

# 30. Take, TakeWhile - Partitioning operators split the sequence (collection) into two parts and returns one of the parts.

**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().

**TakeWhile**

The TakeWhile() 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  
Four