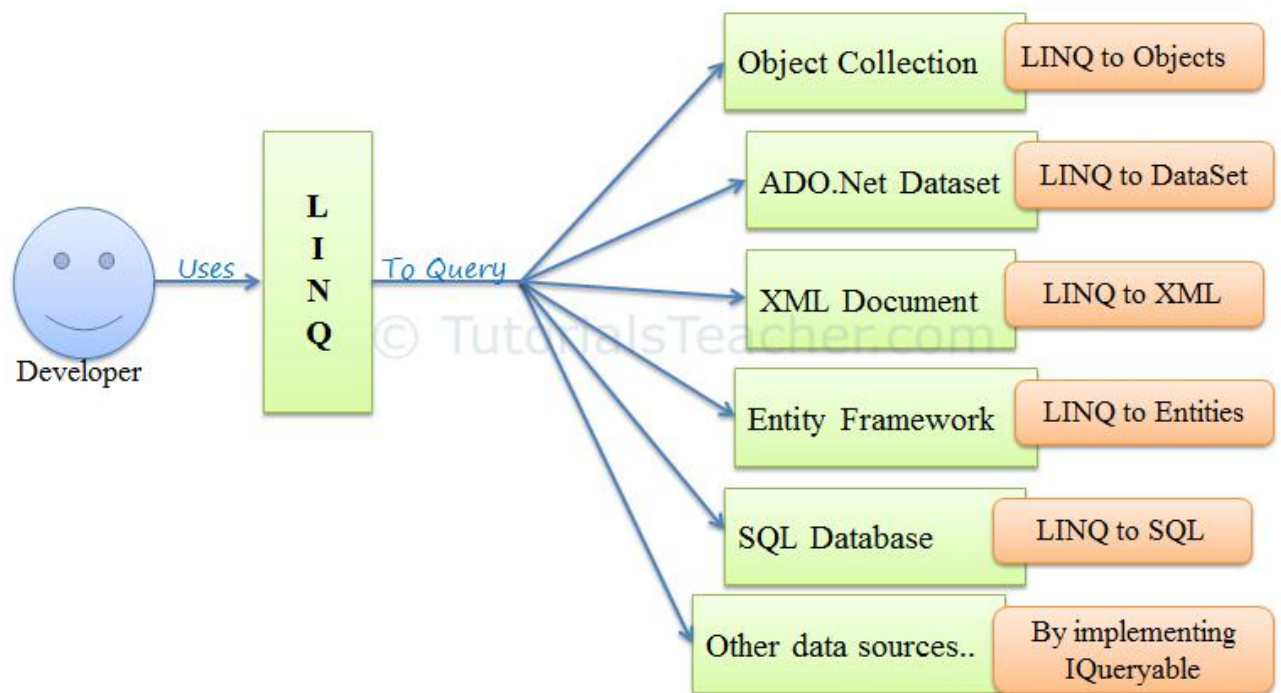


LINQ

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

For example, SQL is a Structured Query Language used to save and retrieve data from a database. In the same way, LINQ is a structured query syntax built in C# and VB.NET used to save and retrieve data from different types of data sources like an Object Collection, SQL server database, XML, web service etc.

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



```
class Student
{
    public int StudentID { get; set; }
    public String StudentName { get; set; }
    public int Age { get; set; }
}
class Program
{
```

```

static void Main(string[] args)
{
    Student[] studentArray = {
        new Student() { StudentID = 1, StudentName = "John", Age =
18 },
        new Student() { StudentID = 2, StudentName = "Steve", Age
= 21 },
        new Student() { StudentID = 3, StudentName = "Bill", Age
= 25 },
        new Student() { StudentID = 4, StudentName = "Ram" , Age =
20 },
        new Student() { StudentID = 5, StudentName = "Ron" , Age =
31 },
        new Student() { StudentID = 6, StudentName = "Chris", Age
= 17 },
        new Student() { StudentID = 7, StudentName = "Rob", Age = 1
9 },
    };

    Student[] students = new Student[10];

    int i = 0;

    foreach (Student std in studentArray)
    {
        if (std.Age > 12 && std.Age < 20)
        {
            students[i] = std;
            i++;
        }
    }
}

```

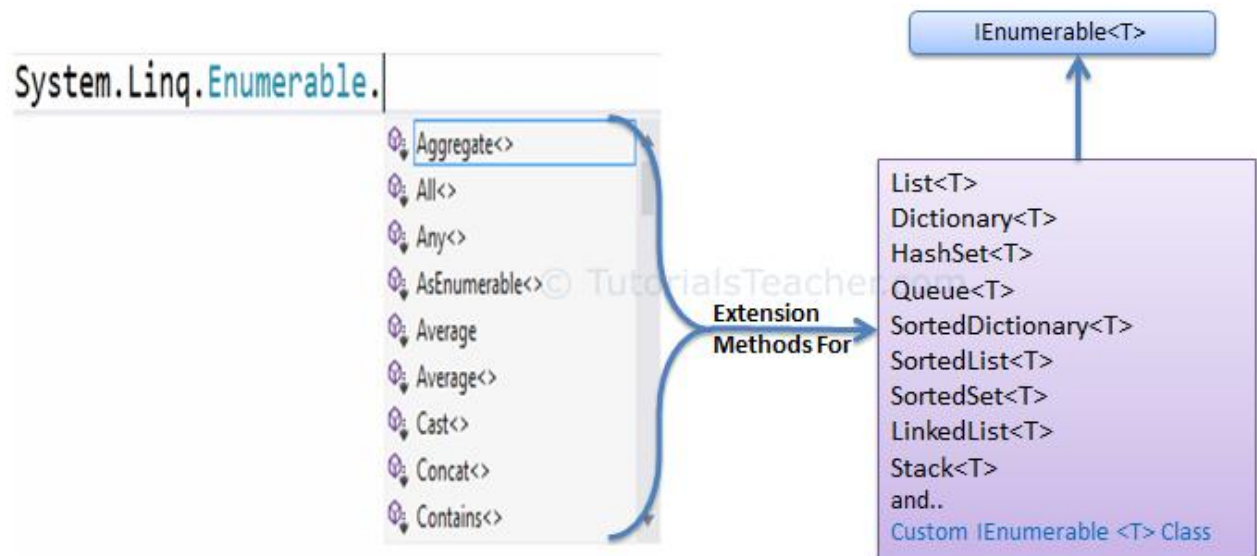
LINQ API

LINQ is nothing but the collection of extension methods for classes that implements `IEnumerable` and `IQueryable` interface. `System.Linq` namespace includes the necessary classes & interfaces for LINQ. `Enumerable` and `Queryable` are two main static classes of LINQ API that contain extension methods.

Enumerable

Enumerable class includes extension methods for the classes that implement `IEnumerable<T>` interface, this include all the collection types in `System.Collections.Generic` namespaces such as `List<T>`, `Dictionary<T>`, `SortedList<T>`, `Queue<T>`, `HashSet<T>`, `LinkedList<T>` etc.

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



Queryable

The **Queryable** class includes extension methods for classes that implement `IQueryable<T>` interface. `IQueryable<T>` is used to provide querying capabilities against a specific data source where the type of the data is known. For example, Entity Framework api implements `IQueryable<T>` interface to support LINQ queries with underlying database like SQL Server.

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

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

System.Linq.Queryable.

- Aggregate<>
- All<>
- Any<>
- AsQueryable
- AsQueryable<>
- Average
- Average<>
- Cast<>
- Concat<>

Extension
Methods For

IQueryable<T>

LINQ to SQL

EntityFramework

LINQ to Amazon

LINQ to LDAP

PLINQ

© TutorialsTeacher.com

LINQ Query Syntax

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

1. Query Syntax or Query Expression Syntax
2. Method Syntax or Method extension syntax or Fluent

Query Syntax:

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

SYNTAX:

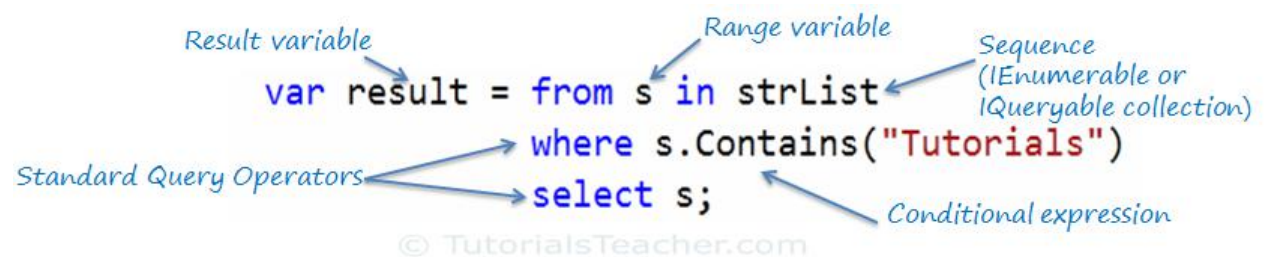
from <range variable> in <IEnumerable<T> or IQueryable<T> Collection>

<Standard Query Operators> <lambda expression>

<select or groupBy operator> <result formation>

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

```
// string collection IList<string> stringList = new List<string>() {  
    "C# Tutorials",  
    "VB.NET Tutorials",  
    "Learn C++",  
    "MVC Tutorials" ,  
    "Java"  
};  
// LINQ Query Syntax  
var result = from s in stringList  
              where s.Contains("Tutorials")  
              select s;
```



Query syntax starts with a **From** clause followed by a **Range** variable. The **From** clause is structured

like `"From rangeVariableName in IEnumerablecollection"`. In English, this means, from each object in the collection. It is similar to a foreach loop: `foreach(Student s in studentList)`.

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

```
// Student collection
IList<Student> studentList = new List<Student>>()
{
    new Student() { StudentID = 1, StudentName = "John", Age = 13} ,
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20} ,
    new Student() { StudentID = 5, StudentName = "Ron", Age = 15 }
};
// LINQ Query Syntax to find out teenager students
var teenAgerStudent
= from s in studentList
    where s.Age > 12 && s.Age < 20
    select s;
```

LINQ Method Syntax

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

Method syntax (also known as fluent syntax) uses extension methods included in the **Enumerable** or **Queryable** static class, similar to how you would call the extension method of any class.

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

```
/ string collectionIList<string> stringList = new List<string>() {
    "C# Tutorials",
    "VB.NET Tutorials",
    "Learn C++",
    "MVC Tutorials" ,
    "Java"
};
// LINQ Query Syntax
var result = stringList.Where(s => s.Contains("Tutorials"));
```

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

© TutorialsTeacher.com

Extension method

Lambda expression

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

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

```
var students = studentList.Where()
```

▲ 1 of 2 ▼ (extension) IEnumerable<Student> IEnumerable<Student>.Where(Func<Student,bool> predicate)

Filters a sequence of values based on a predicate.

predicate: A function to test each element for a condition.

Func delegate

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

```
// Student collection
IList<Student> studentList = new List<Student>()
{
    new Student() { StudentID = 1, StudentName = "John", Age = 13 } ,
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21 } ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18 } ,
    new Student() { StudentID = 4, StudentName = "Ram", Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron", Age = 15 }
};
// LINQ Method Syntax to find out teenager students
var teenAgerStudents = studentList.Where(s => s.Age > 12 && s.Age < 20)
    .ToList<Student>();
```

```
Dim studentList = New List(Of Student) From {
    New Student() With {.StudentID = 1,
        .StudentName = "John", .Age = 13},
    New Student() With {.StudentID = 2,
        .StudentName = "Moin", .Age = 21},
    New Student() With {.StudentID = 3,
        .StudentName = "Bill", .Age = 18},
    New Student() With {.StudentID = 4,
        .StudentName = "Ram", .Age = 20},
    New Student() With {.StudentID = 5,
        .StudentName = "Ron", .Age = 15}
}
// LINQ Method Syntax to find out teenager students
Dim teenAgerStudents As IList(Of Student) =
    studentList.Where(Function(s) s.Age > 12 And s.Age < 20)
        .ToList()
```


Lambda Expression

C# 3.0(.NET 3.5) introduced the lambda expression along with LINQ. The lambda expression is a shorter way of representing Anonymous Method using some special syntax.

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

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

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

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

Lambda Expression with Multiple parameters:

1-

```
(s, youngAge) => s.Age >= youngAge;
```

2-

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

Lambda expression without any parameter:

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

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

Multiple statements in body expression:

```
(s, youngAge) =>{  
    Console.WriteLine("Lambda expression with multiple statements in the body");  
}
```

```
Return s.Age >= youngAge;}
```

Func Delegate:

Use the Func<> delegate when you want to return something from a lambda expression. The last parameter type in a Func<> delegate is the return type and rest are input parameters. Visit [Func delegate](#) section of C# tutorials to know more about it.

```
Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;  
Student std = new Student() { age = 21 };  
bool isTeen = isStudentTeenAger(std); // returns false
```

Action Delegate:

Unlike the Func delegate, an Action delegate can only have input parameters. Use the [Action delegate](#) type when you don't need to return any value from lambda expression.

```
Action<Student> PrintStudentDetail = s => Console.WriteLine(  
"Name: {0}, Age: {1} ", s.StudentName, s.Age);  
Student std = new Student(){ StudentName = "Bill", Age=21};  
  
PrintStudentDetail(std); //output: Name: Bill, Age: 21
```

Lambda Expression in LINQ Query:

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

```
var students = studentList.Where(|
```

▲ 1 of 2 ▼ (extension) IEnumerable<Student> IEnumerable<Student>.Where(Func<Student,bool> predicate)

Filters a sequence of values based on a predicate.

predicate: A function to test each element for a condition.

Func delegate

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

```
IList<Student> studentList = new List<Student>(){...};  
Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;  
var teenStudents = studentList.Where(isStudentTeenAger)  
                                .ToList<Student>();
```

```
IList<Student> studentList = new List<Student>(){...};  
Func<Student, bool> isStudentTeenAger = s => s.age > 12 && s.age < 20;  
var teenStudents = from s in studentList  
                   where isStudentTeenAger(s)  
                   select s;
```


Standard Query Operators

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

Standard Query Operators in Query Syntax:

```
var students = from s in studentList
                where s.age > 20
                select s;
```

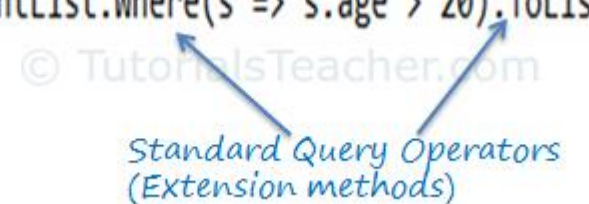
Standard Query Operators



Standard Query Operators in Method Syntax:

```
var students = studentList.Where(s => s.age > 20).ToList<Student>();
```

Standard Query Operators (Extension methods)



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

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

Classification	Standard Query Operators
Filtering	Where, OfType
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse

Grouping	GroupBy, ToLookup
Join	GroupJoin, Join
Projection	Select, SelectMany
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Quantifiers	All, Any, Contains
Elements	ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Set	Distinct, Except, Intersect, Union
Partitioning	Skip, SkipWhile, Take, TakeWhile
Concatenation	Concat
Equality	SequenceEqual
Generation	DefaultEmpty, Empty, Range, Repeat
Conversion	AsEnumerable, AsQueryable, Cast, ToArray, ToDictionary, ToList

1-Filtering Operators – Where

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

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

Filtering Operators	Description
Where	Returns values from the collection based on a predicate function
OfType	Returns values from the collection based on a specified type. However, it will depend on their ability to cast to a specified type.

Where

Where clause in Query Syntax:

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

```
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 = from s in studentList  
                     where s.Age > 12 && s.Age < 20  
                     select s.StudentName;
```

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

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

```
Func<Student, bool> isTeenAger = delegate(Student s) {  
    return s.Age > 12 && s.Age < 20;  
};  
var filteredResult = from s in studentList  
                     where isTeenAger(s)  
                     select s;
```

2-Filtering Operator - OfType

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

OfType in Query Syntax:

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

```
ICollection 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;
```

OfType in Method Syntax:

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

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

3-Sorting Operators: OrderBy & OrderByDescending

A sorting operator arranges the elements of the collection in ascending or descending order.

Sorting Operator	Description
OrderBy	Sorts the elements in the collection based on specified fields in ascending or decending order.
OrderByDescending	Sorts the collection based on specified fields in descending order. Only valid in method syntax.
ThenBy	Only valid in method syntax. Used for second level sorting in ascending order.
ThenByDescending	Only valid in method syntax. Used for second level sorting in descending order.
Reverse	Only valid in method syntax. Sorts the collection in reverse order.

LINQ includes following sorting operators.

OrderBy:

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

```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,
    new Student() { StudentID = 2, StudentName = "Steve", Age = 15 }
    ,
    new Student() { StudentID = 3, StudentName = "Bill", Age = 25 } ,
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }
};
var orderByResult = from s in studentList
                    orderby s.StudentName
                    select s;
var orderByDescendingResult = from s in studentList
                               orderby s.StudentName descending
                               select s;
```


OrderBy in Method Syntax:

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

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

```
IList<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,  
    new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,  
    new Student() { StudentID = 3, StudentName = "Bill", Age = 25 } ,  
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,  
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }  
};  
var studentsInAscOrder = studentList.OrderBy(s => s.StudentName);
```

OrderByDescending:

OrderByDescending sorts the collection in descending order.

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

```
IList<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 18 } ,  
    new Student() { StudentID = 2, StudentName = "Steve", Age = 15 } ,  
    new Student() { StudentID = 3, StudentName = "Bill", Age = 25 } ,  
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 } ,  
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }  
};  
var studentsInDescOrder = studentList.OrderByDescending  
    (s => s.StudentName);
```

Multiple Sorting:

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

4-Sorting Operators: ThenBy & ThenByDescending

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

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

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

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

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

```

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).ThenBy
Descending(s => s.Age);

```

5-Grouping Operators: GroupBy & ToLookup

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

Grouping Operators	Description
GroupBy	The GroupBy operator returns groups of elements based on some key value. Each group is represented by <code>IGrouping<TKey, TElement></code> object.
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.

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

```
var groupedResult = studentList.GroupBy(|
```

▲ 1 of 8 ▼ (extension) `IEnumerable<IGrouping<TKey,Student>>` `IEnumerable<Student>.GroupBy(Func<Student,TKey> keySelector)`

Groups the elements of a sequence according to a specified key selector function.

keySelector: A function to extract the key for each element.

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.

```
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);
}
```

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);
}

```

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);  
}
```

6-Joining Operator: Join

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

Joining Operators	Usage
Join	The Join operator joins two sequences (collections) based on a key and returns a resulted sequence.
GroupJoin	The GroupJoin operator joins two sequences based on keys and returns groups of sequences. It is like Left Outer Join of SQL.

Join:

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

Join in Method Syntax:

The Join extension method has two overloads as shown below.

```
ICollection<string> strList1 = new List<string>() {  
    "One",  
    "Two",  
    "Three",  
    "Four"  
};  
ICollection<string> strList2 = new List<string>() {  
    "One",  
    "Two",  
    "Five",  
    "Six"  
};  
var innerJoin = strList1.Join(strList2,
```

```

        str1 => str1,
        str2 => str2,
        (str1, str2) => str1);

```

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

```

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; }
}

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.StandardNam
                                });

```


Outer Sequence

Inner Sequence

Key selector of outer sequence

Key selector of inner sequence

Projection Result

```
var joinResult = studentList.Join(standardList,
    student => student.StandardID,
    standard => standard.StandardID,
    (student, standard) => new
    {
        StudentFullName = student.StudentName,
        StandarFullName = standard.StandardName
    });
```

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.

```
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
                };
```

7-Joining Operator: 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.

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; }
}
```

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,
            StandarFullldName = std.StandardName
        });
foreach (var item in groupJoin)
{
    Console.WriteLine(item.StandarFullldName );

    foreach(var stud in item.Students)
        Console.WriteLine(stud.StudentName);
}

```

Syntax: GroupJoin in Query syntax

```

from ... in outerSequence

    join ... in innerSequence

    on outerKey equals innerKey

    into groupedCollection

    select ...

```

```

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.StandarFullldName );

    foreach(var stud in item.Students)
        Console.WriteLine(stud.StudentName);
}
```

8 - Projection Operators: Select, SelectMany

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

Select:

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

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

```
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.

```
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 following example of the select clause returns a collection of **anonymous type** containing the Name and Age property.

```
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 selectResultforeach (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:

```
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 });
```

All:

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

```

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.

```

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

```

9-Quantifier Operator: Contains

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

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

```
IList<int> intList = new List<int>() { 1, 2, 3, 4, 5 };bool result =  
intList.Contains(10); // returns false
```

```
IList<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 18 }  
    ,  
    new Student() { StudentID = 2, StudentName = "Steve", Age = 1  
5 } ,  
    new Student() { StudentID = 3, StudentName = "Bill", Age = 25  
    } ,  
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20 }  
    ,  
    new Student() { StudentID = 5, StudentName = "Ron" , Age = 19 }  
};  
Student std = new Student(){ StudentID =3, StudentName = "Bill"};bool  
result = studentList.Contains(std); //returns false
```

```
class StudentComparer : IEqualityComparer<Student>  
{  
    public bool Equals(Student x, Student y)  
    {  
        if (x.StudentID == y.StudentID &&  
            x.StudentName.ToLower() == y.StudentName.ToLow  
er())  
            return true;  
        return false;  
    }  
    public int GetHashCode(Student obj)
```



```
    {  
        return obj.GetHashCode();  
    }  
}
```

```
ICollection<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 18 },  
    ,  
    new Student() { StudentID = 2, StudentName = "Steve", Age = 1  
5 } ,  
    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
```

10-Aggregation Operators:

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

Aggregate:

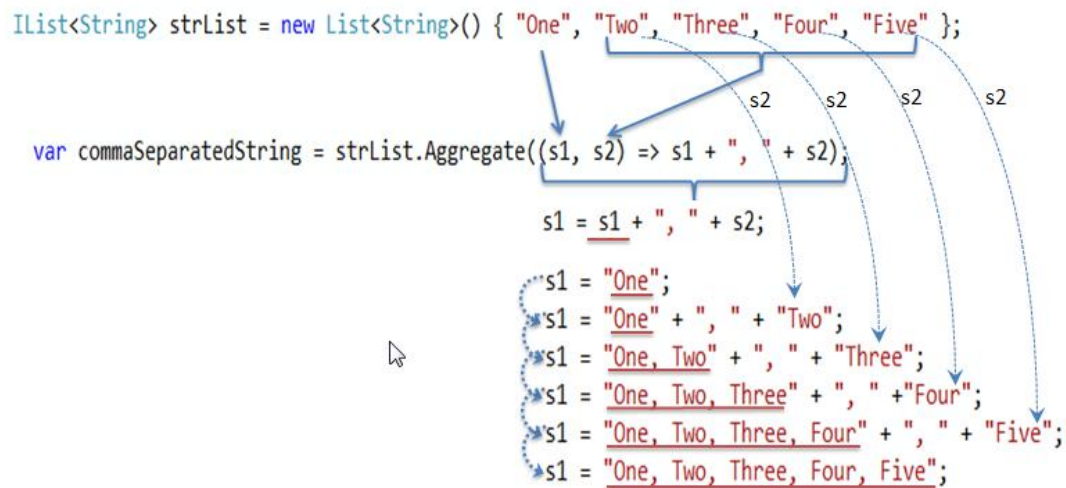
The Aggregate method performs an accumulate operation.

```
IList<String> strList = new List<String>() { "One", "Two", "Three", "Four", "Five"};
var commaSeperatedString = strList.Aggregate((s1, s2) => s1 + ", " + s2);
```

```
Console.WriteLine(commaSeperatedString);
```

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.



11-Aggregation Operator: 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.

```
ICollection<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.

```
ICollection<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);
```

12-Aggregation Operator: Count

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

```
ICollection<Student> studentList = new List<Student>() {  
    new Student() { StudentID = 1, StudentName = "John", Age = 13}  
    ,  
    new Student() { StudentID = 2, StudentName = "Moin", Age = 21  
    } ,  
    new Student() { StudentID = 3, StudentName = "Bill", Age = 18  
    } ,  
    new Student() { StudentID = 4, StudentName = "Ram" , Age = 20}  
    ,  
    new Student() { StudentID = 5, StudentName = "Mathew" , Age =  
15 }  
};  
var numOfStudents = studentList.Count();  
Console.WriteLine("Number of Students: {0}", numOfStudents);
```

13-Aggregation Operator: Max

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

```
ICollection<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 );
```

The following example demonstrates Max() method on the complex type 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 }  
};  
var oldest = studentList.Max(s => s.Age);  
Console.WriteLine("Oldest Student Age: {0}", oldest);
```

14-Aggregation Operator: Sum

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

```
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 );
```

```
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;  
    return 0;  
});
```

```

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

```

15-Element Operators: ElementAt, ElementAtOrDefault

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

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.

Element Operators (Methods)	Description
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.

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 ou
t of range");
Console.WriteLine(intList.ElementAt(9));

```

```

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());

```

16-Equality Operator: SequenceEqual

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

```
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);
```

17-Concatenation Operator: Concat

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

```
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);
```

18-Generation Operator:

DefaultIfEmpty:

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

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

Consider the following example.

```
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));
```

Generation Operators: Empty, Range, Repeat

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

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

Empty:

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

```
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 );
```

Range:

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

```
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));
```

```
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
```

Repeat:

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

```
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));
```

```
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
```

Set operator: Distinct

The following table lists all Set operators available in LINQ.

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

Distinct:

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

```

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);

```

```

One
Two
Three
1
2
3
4
5

```

```

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();
    }
}

```

```

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);

```

Set operator: Except

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

```

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);

```

```

One
Two
Three

```
