




LINQ

(Language-Integrated Query)

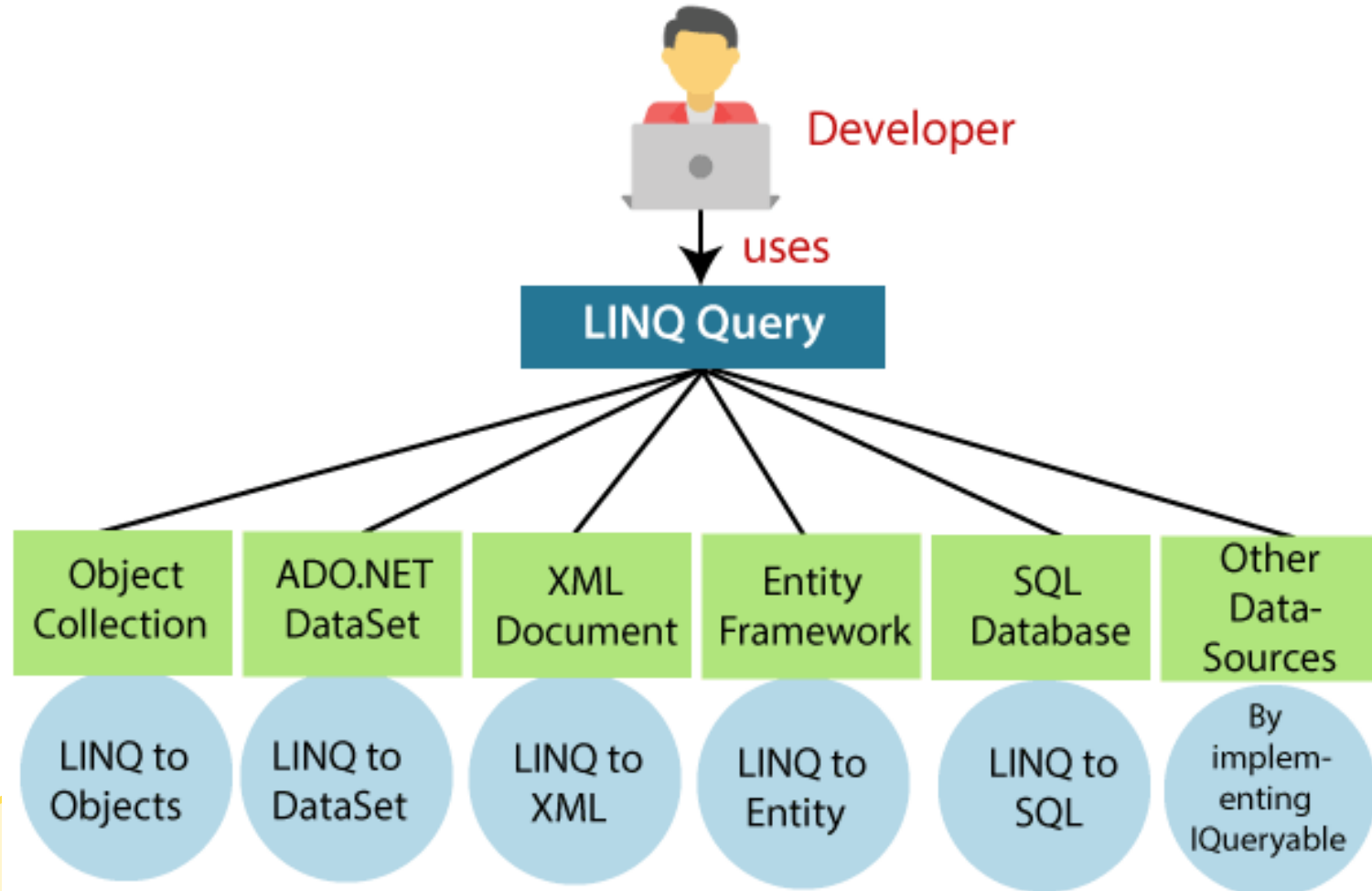
A powerful set of technologies based on the integration of query capabilities directly into the C# language



Introduction

- The first-class language construct/ built-in C# .NET, just like classes, methods, events.
- A consistent query experience to query objects (LINQ to Objects), relational databases (LINQ to SQL), and XML (LINQ to XML).
- LINQ ably integrates queries in C# and Visual Basic.
- LINQ is a Structured Query Syntax and database independent.
- LINQ functionality can be achieved by importing ***System.Linq*** namespace in our application.

Introduction



Prerequisites

- There are no prerequisites required to learn LINQ.
- It is good to have .NET Framework and Visual Studio installed on your computer.

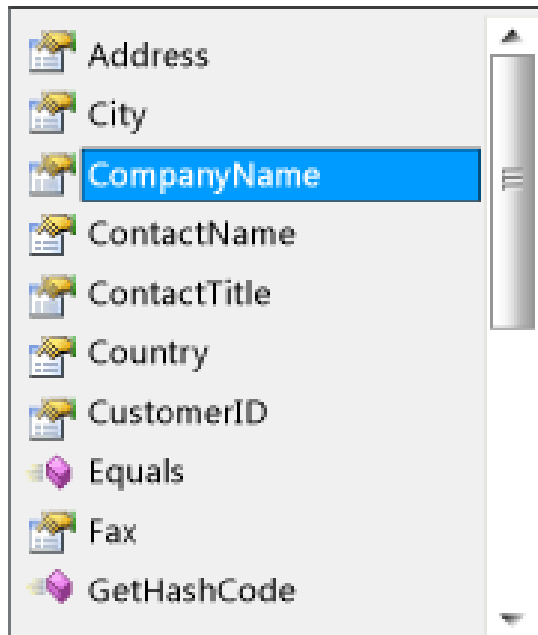
Why LINQ?

- LINQ is used to retrieve the data from different types of sources such as XML, docs, collections, ADO.Net DataSet, Web Service, MS SQL Server, RDBMS, In-memory Objects and other databases servers.
- LINQ was introduced in C# 3.0 & .NET Framework 3.5 Visual Studio 2008.
- ***IEnumerable*** interface can be used for querying, filtering, grouping, ordering, & projecting data.

Introduction

```
Northwnd nw = new  
Northwnd(@"northwnd.mdf");  
  
var companyNameQuery =  
    from cust in nw.Customers  
    where cust.
```

```
Dim nw As New _  
Northwnd("c:\northwnd.mdf")  
  
Dim companyNameQuery = _  
    From cust In nw.Customers _  
    Where cust.
```



string Customer.CompanyName

LINQ Query Syntax

from <variable> in <collection>

<where, joining, grouping, operators etc.> <lambda expression>

<select or groupBy operator> <format the results>

- “**from**” keyword will form the starting point of the LINQ query followed by a user defined variable followed by “in” which specifies our source collection or Data source followed by a where clause, if there is a specific condition in the query that can be used before the select to filter out the records and select is followed by group by and into the clause.

LINQ Query Example

```
int[] Num = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
IEnumerable<int> result = from numbers in Num
```

```
    where numbers > 3
```

```
    select numbers;
```


LINQ Query C# Example

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace Linqtutorials
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] Num = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
            //LINQ Query Syntax to Print Numbers Greater than 3
            IEnumerable<int> result = from numbers in Num
                                    where numbers >3
                                    select numbers;

            var sorted = from i in myarray orderby i descending select i;
            var oddNumbers_sample = from i in myarray
                                    where i % 2 == 1
                                    select i;

            foreach (var item in result)
            {
                Console.WriteLine(item);
            }
            Console.ReadLine();
        }
    }
}
```

LINQ Query Syntax

<u><i>Clause</i></u>	<u><i>Description</i></u>
From	Identifier
In	Source Collection
Let	Expression
Where	Boolean Expression
order by	Expression
Select	Expression
group by	Expression
intro	Expression

LINQ Query Example

```
// Specify the data source.  
int[] scores = { 100, 22, 150, 50 };  
  
// Define the query expression.  
IEnumerable<int> scoreQuery =  
    from score in scores  
    where score > 80  
    select score;  
  
// Execute the query.  
foreach (int i in scoreQuery)  
{  
    Console.Write(i + " ");  
}  
  
// Output: 100 150
```

LINQ Method Syntax

```
int[] Num = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
//LINQ Method Syntax to Print Numbers Greater than 3
```

```
IEnumerable<int> result = Num.Where(n => n > 3).ToList();
```

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ Method Syntax

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            int[] Num = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
            //LINQ Method Syntax to Print Numbers Greater than 3
```

```
            IEnumerable<int> result = Num.Where(n => n > 3).ToList();
```

```
            foreach (var item in result)
```

```
            {
```

```
                Console.WriteLine(item);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

LINQ Lambda Expressions

- Lambda expression is a function without a name.
- It makes the syntax short and precise.
- Its scope is limited to where it is used as an expression. We cannot reuse it afterward.
- (Input Parameter) => Method Expression

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

LINQ Lambda Expressions

```
namespace LINQExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<string> countries = new List<string>();  
            countries.Add("India");  
            countries.Add("US");  
            countries.Add("Australia");  
            countries.Add("Russia");  
            IEnumerable<string> result = countries.Select(x => x);  
            foreach (var item in result)  
            {  
                Console.WriteLine(item);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

LINQ Lambda Expressions

```
namespace LINQExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };  
            Console.WriteLine("Below are the even numbers");  
            Console.WriteLine("");  
            IEnumerable<int> evennumber = numbers.Where(x => x % 2 == 0);  
            foreach (var item in evennumber)  
            {  
                Console.WriteLine(item + " is an even number");  
            }  
            Console.WriteLine("-----");  
            Console.WriteLine("Below are the Odd numbers");  
            Console.WriteLine("");  
            IEnumerable<int> oddnumber = numbers.Where(x => x % 2 != 0);  
            foreach (var item in oddnumber)  
            {  
                Console.WriteLine(item + " is an odd number");  
            }  
            Console.ReadLine();  
        }  
    }  
}
```


LINQ Min Function to Get Minimum Value From List

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

```
namespace Linqtutorials  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int[] Num = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
            Console.WriteLine("The Minimum value in the given array is:");  
            int minimumNum = Num.Min();  
            int maximumNum = Num.Max();  
            int summation = Num.Sum();  
            int numberOfElement = Num.Count();  
            int average1 = Num.Average();  
            Console.WriteLine("The minimum Number is {0}", minimumNum);  
            Console.ReadLine();  
        }  
    }  
}
```

LINQ Aggregate Function with example

- The ***Aggregate()*** function will perform the action on the first and second elements and then carry forward the result.
-

LINQ Aggregate() Function Example in C#

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Linqtutorials
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] Num = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
            Console.WriteLine("Find the Product of the elements:");
            double Average = Num.Aggregate((a, b) => a * b);
            Console.WriteLine("The Product is {0}", Average); //Output 362880 (((((((1*2)*3)*4)*5)*6)*7)*8)*9
            string[] charlist = { "a", "b", "c", "d" };
            var concta = charlist.Aggregate((a, b) => a + ',' + b);
            Console.WriteLine("Concatenated String: {0}",concta); // Output a,b,c,d
            Console.ReadLine();
        }
    }
}
```

LINQ Projection Operators (Select, SelectMany)

- Projection is an operation that transforms an object into a new form that often consists of only the properties which we use frequently.

Types of LINQ Projection Operators

<u>Operator</u>	<u>Description</u>	<u>Query Syntax</u>
Select	This operator projects values based on transform functions.	select
SelectMany	This operator projects sequence of values that are based on a transform function and then flattens them into one sequence	Use multiple from clauses

LINQ Projection Operators (Select, SelectMany)

- Projection is an operation that transforms an object into a new form that often consists of only the properties which we use frequently.
- Select projection operator is used to select data from a collection. This LINQ select operator is same as SQL select clause.
- `var result = from u in userslist`
- `select u;`

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

```
namespace Linqtutorials
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<Student> Objstudent = new List<Student>()  
            {  
                new Student() { StudentId = 1, Name = "Suresh", Marks = 500 },  
                new Student() { StudentId = 2, Name = "Rohini", Marks = 300 },  
                new Student() { StudentId = 3, Name = "Madhav", Marks = 400 }  
            };  
            var result = from s in Objstudent  
                           select new {SName =s.Name,SID = s.StudentId,SMarks = s.Marks };  
            foreach (var item in result)  
            {  
                Console.WriteLine("The StudentName is {0} ID is {1} Marks is {2}", item.SName, item.SID, item.SMarks);  
            }  
            Console.ReadLine();  
        }  
    }  
    class Student  
    {  
        public int StudentId { get; set; }  
        public string Name { get; set; }  
        public int Marks { get; set; }  
    }  
}
```

LINQ Select Projection Operator

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
namespace Linqtutorials
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Student> Objstudent = new List<Student>()
            {
                new Student() { StudentId = 1, Name = "Suresh", Marks = 500 },
                new Student() { StudentId = 2, Name = "Rohini", Marks = 300 },
                new Student() { StudentId = 3, Name = "Madhav", Marks = 400 }
            };
            var result = Objstudent.Select(student => new
            {
                SName = student.Name,
                SID = student.StudentId,
                SMarks = student.Marks
            });
            foreach (var item in result)
            {
                Console.WriteLine("The StudentName is {0} ID is {1} Marks is {2}", item.SName, item.SID, item.SMarks);
            }
            Console.ReadLine();
        }
    }
}

class Student
{
    public int StudentId { get; set; }
    public string Name { get; set; }
    public int Marks { get; set; }
}
}
```

LINQ Select Operator in Method Syntax

LINQ SelectMany Projection Operator

- ***SelectMany*** operator is used to select values from a collection of collections, i.e., nested collection. It does this operation by itself, thereby reducing the lines of code that a normal program would typically have

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

LINQ SelectMany Projection Operator

```
namespace Linqtutorials
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<Student> Objstudent = new List<Student>()  
            {  
                new Student() { Name = "Ravi Varma", Gender = "Male", Subjects = new List<string> { "Mathematics", "Physics" } },  
                new Student() { Name = "Vikram Sharma", Gender = "Male", Subjects = new List<string> { "Social Studies", "Chemistry" } }  
            };  
            var Subjects = Objstudent.SelectMany(x => x.Subjects);  
            foreach (var Subject in Subjects)  
            {  
                Console.WriteLine(Subject);  
            }  
            Console.ReadLine();  
        }  
    }  
    class Student  
    {  
        public string Name { get; set; }  
        public string Gender { get; set; }  
        public List<string> Subjects { get; set; }  
    }  
}
```

LINQ Where Clause Filtering Operator

- Filtering operators are used to filter list/collection data based on filter conditions.
- A filtering operator in LINQ specifies the statement should only affect rows that meet specified criteria.
- The criteria expressed as predicates, the where clause is not a mandatory clause of LINQ statements but can be used to limit the number of records affected by a LINQ.
- The where clause is only used to extract records from select, delete, update,, etc.

LINQ Where Clause Filtering Operator

<u>Operator</u>	<u>Description</u>	<u>Query Syntax</u>
Where	This operator is used to select values from the list based on predicate functions	where
OfType	This operator is used to select values based on their ability to caste a specified type.	Not Applicable

LINQ Where Clause Filtering Operator

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Linqtutorials
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] countries = { "India", "Australia", "USA", "Argentina", "Peru", "China" };
            IEnumerable<string> result = countries.Where(x => x.StartsWith("A"));
            foreach (var country in result)
            {
                Console.WriteLine(country);
            }
            Console.ReadLine();
        }
    }
}
```

LINQ Where Clause Filtering Operator

- ***where*** clause to specify a condition just like we do in normal SQL.
- Then we used a lambda expression to specify an input parameter x and use `StartsWith` function on the input parameter to denote that we want only those countries from the array whose name starts with “A”.

LINQ Where Clause Filtering Operator

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Linqtutorials
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] countries = { "India", "Australia", "USA", "Argentina", "Peru", "China" };
            IEnumerable<string> result = from x in countries
                                        where x.StartsWith("A")
                                        select x;
            foreach (var country in result)
            {
                Console.WriteLine(country);
            }
            Console.ReadLine();
        }
    }
}
```

LINQ Where Clause Filtering Operator

```
IEnumerable<string> result = from x in countries  
    where x.StartsWith("A")  
    where x.EndsWith("s")  
    select x;
```


LINQ Sorting Operators (Order By / Then By / Reverse)

- sorting operators are used to change the order or sequence of data (either ascending or descending) based on one or more attributes.
-

LINQ Sorting Operators (Order By / Then By / Reverse)

<u>Operator</u>	<u>Description</u>	<u>Query Syntax</u>
OrderBy	This operator will sort values in ascending order.	orderby
OrderByDescending	This operator will sort values in descending order.	orderby ... descending
ThenBy	This operator is used to perform secondary sorting in ascending order.	orderby
ThenByDescending	This operator is used to perform sorting in descending order.	Orderby descending
Reverse	This operator is used to reverse the order of elements in a collection.	Not Applicable

LINQ OrderBy Operator (Ascending)

- OrderBy operator is used to sort list/collection values in ascending order.
- In LINQ, if we use orderby operator by default, it will sort a list of values in ascending order; we don't need to add any ascending condition in the query statement

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

LINQ OrderBy Operator (Ascending)

```
namespace Linqtutorials  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<Student> Objstudent = new List<Student>()  
            {  
                new Student() { Name = "Suresh Dasari", Gender = "Male", Subjects = new List<string> { "Mathematics", "Physics" } },  
                new Student() { Name = "Rohini Alavala", Gender = "Female", Subjects = new List<string> { "Entomology", "Botany" } }  
            };  
            var studentname = Objstudent.OrderBy(x => x.Name);  
            foreach (var student in studentname)  
            {  
                Console.WriteLine(student.Name);  
            }  
            Console.ReadLine();  
        }  
    }  
    class Student  
    {  
        public string Name { get; set; }  
        public string Gender { get; set; }  
        public List<string> Subjects { get; set; }  
    }  
}
```

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

LINQ OrderBy Operator (Descending)

```
namespace LINQExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<Student> Objstudent = new List<Student>()  
            {  
                new Student() { Name = "Suresh Dasari", Gender = "Male", Subjects = new List<string> { "Mathematics", "Physics" } },  
                new Student() { Name = "Rohini Alavala", Gender = "Female", Subjects = new List<string> { "Entomology", "Botany" } },  
            };  
            var studentname = Objstudent.OrderByDescending(x => x.Name);  
            foreach (var student in studentname)  
            {  
                Console.WriteLine(student.Name);  
            }  
            Console.ReadLine();  
        }  
    }  
    class Student  
    {  
        public string Name { get; set; }  
        public string Gender { get; set; }  
        public List<string> Subjects { get; set; }  
    }  
}
```

LINQ ThenBy Sorting Operator

- **ThenBy** sorting operator is used to implement sorting on multiple fields, and by default, the **ThenBy** operator will sort collection items in ascending order.
- Generally, in LINQ **ThenBy** operator is used along with the **OrderBy** operator to implement sorting on multiple fields in the list/collection.
- If we want more than one sorting condition, we can use the **ThenBy** clause with the **OrderBy** clause.
- **OrderBy** is the primary sorting operator, and **ThenBy** is a secondary sorting operator.

LINQ ThenByDescending Sorting Operator

- ***ThenByDescending*** sorting operator is used to implement sorting on multiple fields in the list/collection.
- By default, the ***ThenByDescending*** operator will sort list items in descending order, and in LINQ, we use the ***ThenByDescending*** operator along with the ***OrderBy*** operator.
- In LINQ ***ThenByDescending*** operator is used to specify the second sorting condition to be descending, and the ***OrderBy*** operator is used to specify the primary sorting condition.

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

```
namespace LINQExamples
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<Student> Objstudent = new List<Student>()  
            {  
                new Student() { RoleId=1, Name = "Suresh Dasari", Gender = "Male", Subjects = new List<string> { "Mathematics","Physics" } },  
                new Student() { RoleId=2, Name = "Rohini Alavala", Gender = "Female", Subjects = new List<string> { "Entomology", "Botany" } }  
            };  
            var studentname = Objstudent.OrderBy(x => x.Name).ThenByDescending(x => x.RoleId);  
            foreach (var student in studentname)  
            {  
                Console.WriteLine("Name={0} StudentId={1}", student.Name, student.RoleId);  
            }  
            Console.ReadLine();  
        }  
    }  
}  
class Student  
{  
    public int RoleId { get; set; }  
    public string Name { get; set; }  
    public string Gender { get; set; }  
    public List<string> Subjects { get; set; }  
}
```


SQL ThenByDescending Operator with Query Syntax

```
IOrderedEnumerable<Student> studentname = from x in Objstudent
                                             orderby x.Name, x.RoleId descending
                                             select x;
foreach (var student in studentname)
{
    Console.WriteLine("Name={0} StudentId={1}", student.Name, student.RoleId);
}
Console.ReadLine();
```

SQL ThenBy Operator with Query Syntax

```
IOrderedEnumerable<Student> studentname = from x in Objstudent
                                           orderby x.Name, x.RoleId
                                           select x;

foreach (var student in studentname)
{
    Console.WriteLine("Name={0} StudentId={1}", student.Name,
student.RoleId);
}

Console.ReadLine();
```

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
namespace LINQExamples
```

```
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Student> Objstudent = new List<Student>()
            {
                new Student() { RoleId=1, Name = "Suresh Dasari", Gender = "Male", Subjects = new List<string> { "Mathematics", "Physics" } },
                new Student() { RoleId=2, Name = "Rohini Alavala", Gender = "Female", Subjects = new List<string> { "Entomology", "Botany" } },
            };
            Console.WriteLine("Iterating Students in the Normal order");
            Console.WriteLine("");
            IEnumerable<Student> Students = Objstudent;
            foreach (var item in Students)
            {
                Console.WriteLine("StudentId={0} Name={1} Gender={2}", item.RoleId, item.Name, item.Gender);
            }
            Console.WriteLine("-----");
            Console.WriteLine("Iterating Students in the Reverse order");
            Console.WriteLine("");
            IEnumerable<Student> result = Students.Reverse();
            foreach (var item in result)
            {
                Console.WriteLine("StudentId={0} Name={1} Gender={2}", item.RoleId, item.Name, item.Gender);
            }
            Console.ReadLine();
        }
    }
}

class Student
{
    public int RoleId { get; set; }
    public string Name { get; set; }
    public string Gender { get; set; }
    public List<string> Subjects { get; set; }
}
```

LINQ Reverse() Method

LINQ Partition Operators (Take, Skip, TakeWhile, SkipWhile)

- **partition** operators are helpful to partition list/collection items into two parts and return one part of list items.
- **Take** operator is used to get the specified number of elements in sequence from the list/collection.
- The LINQ Take operator will return a specified number of elements from starting of the collection or list.
- **TakeWhile** operator is used to get the elements from the list/collection data source as long as the specified condition holds true in the expression.
- When the condition stops holding true, it does not return those elements.

LINQ Partition Operators (Take, Skip, TakeWhile, SkipWhile)

<i>Operator</i>	<i>Description</i>	<i>Query Syntax</i>
TAKE	This operator returns a specified number of elements in the sequence.	Take
TAKEWHILE	This operator returns the elements in a sequence that satisfies the specified condition.	TakeWhile
SKIP	This operator is used to skip the specified number of elements in a sequence and return the remaining elements.	Skip
SKIPWHILE	This operator is used to skip elements in sequence based on the condition, which is defined as true.	SkipWhile

using System;

using System.Collections.Generic;

using System.Linq;

LINQ Partition Operators (Take, Skip, TakeWhile, SkipWhile)

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            string[] countries = { "India", "USA", "Russia", "China", "Australia", "Argentina" };
```

```
            IEnumerable<string> result = countries.Take(3);
```

```
            foreach (string s in result)
```

```
            {
```

```
                Console.WriteLine(s);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

```
using System;
```

```
using System.Collections.Generic;
```

LINQ Take Operator in Query Syntax Example

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            string[] countries = { "India", "USA", "Russia", "China", "Australia", "Argentina" };
```

```
            IEnumerable<string> result = (from x in countries select x).Take(3);
```

```
            foreach (string s in result)
```

```
            {
```

```
                Console.WriteLine(s);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ TakeWhile Partition Operator

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            string[] countries = { "India", "USA", "Russia", "China", "Australia", "Argentina" };
```

```
            IEnumerable<string> result = (from x in countries select x).Take(3);
```

```
            foreach (string s in result)
```

```
            {
```

```
                Console.WriteLine(s);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```



```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ TakeWhile Operator

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            string[] countries = { "US", "UK", "Russia", "China", "Australia", "Argentina" };
```

```
            IEnumerable<string> result = countries.TakeWhile(x => x.StartsWith("U"));
```

```
            foreach (string s in result)
```

```
            {
```

```
                Console.WriteLine(s);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ TakeWhile in Query Syntax Example

```
namespace LINQExamples
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string[] countries = { "US", "UK", "Russia", "China", "Australia", "Argentina" };  
            IEnumerable<string> result = (from x in countries select x).TakeWhile(x =>  
x.StartsWith("U"));  
            foreach (string s in result)  
            {  
                Console.WriteLine(s);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ Skip Operator

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            string[] countries = { "US", "UK", "Russia", "China", "Australia", "Argentina" };
```

```
            IEnumerable<string> result = (from x in countries select x).TakeWhile(x =>  
x.StartsWith("U"));
```

```
            foreach (string s in result)
```

```
            {
```

```
                Console.WriteLine(s);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

LINQ Partition Operators (Take, Skip, TakeWhile, SkipWhile)

- ***Skip*** operator is used to skip the specified number of elements from the list/collection and returns the remaining elements from a collection
- ***SkipWhile*** operator is used to skip the elements in sequence until the specified condition holds true and returns the remaining elements from the list/collection
- conversion operators are useful to convert the type of elements in the list/collection. In LINQ.

using System;

using System.Collections.Generic;

using System.Linq;

LINQ Partition Operators (Take, Skip, TakeWhile, SkipWhile)

```
namespace LINQExamples
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string[] countries = { "US", "UK", "India", "Russia", "China", "Australia",  
"Argentina" };  
            IEnumerable<string> result = countries.Skip(3);  
            foreach (string s in result)  
            {  
                Console.WriteLine(s);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ SkipWhile Partition Operator

```
namespace LINQExamples
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string[] countries = { "US", "UK", "India", "Russia", "China", "Australia",  
"Argentina" };  
            IEnumerable<string> result = countries.SkipWhile(x => x.StartsWith("U"));  
            foreach (string s in result)  
            {  
                Console.WriteLine(s);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ SkipWhile in Query Syntax Example

```
namespace LINQExamples
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string[] countries = { "US", "UK", "India", "Russia", "China", "Australia",  
"Argentina" };  
            IEnumerable<string> result = (from x in countries select x).SkipWhile(x =>  
x.StartsWith("U"));  
            foreach (string s in result)  
            {  
                Console.WriteLine(s);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

LINQ Conversion Operators (ToList, ToArray, ToLookup, Cast, OfType)

- ***ToList*** operator takes the elements from the given source and returns a new List. So that means the input would be converted to type List.
- ***ToArray()*** operator is used to convert the input elements in collection to an Array.
- ***ToLookup*** operator is an extension method, and it is useful to extract a set of key/value pairs from the source.
- Each element in the resultant collection is a generic Lookup object, which holds the key and subsequence items that match the key.

LINQ Conversion Operators (ToList, ToArray, ToLookup, Cast, OfType)

- **Cast** operator is used to cast/convert all the elements present in a collection into a specified data type of new collection.
- In case if we try to cast/convert different types of elements (string/integer) in the collection, the conversion will fail, and it will throw an exception.
- **OfType** operator is used to return only the elements of a specified type, and other elements will be ignored from the list/collection.

LINQ Conversion Operators (ToList, ToArray, ToLookup, Cast, OfType)

<i>Operator</i>	<i>Description</i>
ToList	It converts a collection to List
ToArray	It converts a collection to an array
ToLookUp	It converts a group of elements into Lookup<tkey, telement>
Cast	It converts non-generic list to generic list (List to IEnumerable)
OfType	It is useful to filter collection based on a specified type
AsEnumerable	It is useful to convert input elements as IEnumerable
AsQueryable	It converts IEnumerable to IQueryable
ToDictionary	It converts input elements into a dictionary based on the key selector function.

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ Conversion Operators (ToList, ToArray, ToLookup, Cast, OfType)

```
namespace LINQExamples
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string[] countries = { "US", "UK", "India", "Russia", "China", "Australia",  
"Argentina" };  
            IEnumerable<string> result = (from x in countries select x).SkipWhile(x =>  
x.StartsWith("U"));  
            foreach (string s in result)  
            {  
                Console.WriteLine(s);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ Conversion Operators (ToList, ToArray, ToLookup, Cast, OfType)

```
namespace LINQExamples
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string[] countries = { "US", "UK", "India", "Russia", "China", "Australia",  
"Argentina" };  
            List<string> result = countries.ToList();  
            foreach (string s in result)  
            {  
                Console.WriteLine(s);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ ToList() in Query Syntax Example

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            string[] countries = { "US", "UK", "India", "Russia", "China", "Australia",  
"Argentina" };
```

```
            List<string> result = (from x in countries select x).ToList();
```

```
            foreach (string s in result)
```

```
            {
```

```
                Console.WriteLine(s);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ ToArray() Method

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            string[] countries = { "US", "UK", "India", "Russia", "China", "Australia",  
"Argentina" };
```

```
            List<string> result = (from x in countries select x).ToList();
```

```
            foreach (string s in result)
```

```
            {
```

```
                Console.WriteLine(s);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

LINQ ToList Method

```
namespace LINQExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string[] countries = { "US", "UK", "India", "Russia", "China", "Australia",  
"Argentina" };  
            IEnumerable<string> result = (from x in countries select x).SkipWhile(x =>  
x.StartsWith("U"));  
            foreach (string s in result)  
            {  
                Console.WriteLine(s);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ ToArray Method

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            string[] countries = { "US", "UK", "India", "Russia", "China", "Australia",  
"Argentina" };
```

```
            string[] countryarray = countries.ToArray();
```

```
            foreach (string s in countryarray)
```

```
            {
```

```
                Console.WriteLine(s);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```


using System;

using System.Collections.Generic;

using System.Linq;

LINQ ToArray() Operator in Query Syntax Example

```
namespace LINQExamples
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            string[] countries = { "US", "UK", "India", "Russia", "China", "Australia",  
"Argentina" };  
            string[] countryarray = (from x in countries select x).ToArray();  
            foreach (string s in countryarray)  
            {  
                Console.WriteLine(s);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
```

LINQ ToLookup() Operator in Method Syntax Example

```
namespace Linqtutorials
{
    class Program
    {
        static void Main(string[] args)
        {
            List<Employee> objEmployee = new List<Employee>()
            {
                new Employee(){ Name="Ashish Sharma", Department="Marketing", Country="India"},
                new Employee(){ Name="John Smith", Department="IT", Country="Australia"}
            };
            var emp = objEmployee.ToLookup(x => x.Department);
            Console.WriteLine("Grouping Employees by Department");
            Console.WriteLine("-----");
            foreach (var KeyValurPair in emp)
            {
                Console.WriteLine(KeyValurPair.Key);
                // Lookup employees by Department
                foreach (var item in emp[KeyValurPair.Key])
                {
                    Console.WriteLine($"{item.Name} {item.Department} {item.Country}");
                }
            }
            Console.ReadLine();
        }
    }
}

class Employee
{
    public string Name { get; set; }
    public string Department { get; set; }
    public string Country { get; set; }
}
```

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

```
namespace LinqTutorials
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<Employee> objEmployee = new List<Employee>()  
            {  
                new Employee(){ Name="Ashish Sharma", Department="Marketing", Country="India"},  
                new Employee(){ Name="John Smith", Department="IT", Country="Australia"}  
            };  
            var emp = (from employee in objEmployee select employee).ToLookup(x => x.Department);  
            Console.WriteLine("Grouping Employees by Department");  
            Console.WriteLine("-----");  
            foreach (var KeyValurPair in emp)  
            {  
                Console.WriteLine(KeyValurPair.Key);  
                // Lookup employees by Department  
                foreach (var item in emp[KeyValurPair.Key])  
                {  
                    Console.WriteLine($"{item.Name} {item.Department} {item.Country}");  
                }  
            }  
            Console.ReadLine();  
        }  
    }  
}  
  
class Employee  
{  
    public string Name { get; set; }  
    public string Department { get; set; }  
    public string Country { get; set; }  
}
```

LINQ ToLookup in Query Syntax Example

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

```
namespace Linqtutorials
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<Employee> objEmployee = new List<Employee>()  
            {  
                new Employee(){ Name="Ashish Sharma", Department="Marketing", Country="India"},  
                new Employee(){ Name="John Smith", Department="IT", Country="Australia"}  
            };  
            var emp = (from employee in objEmployee select employee).ToLookup(x => x.Department);  
            Console.WriteLine("Grouping Employees by Department");  
            Console.WriteLine("-----");  
            foreach (var KeyValurPair in emp)  
            {  
                Console.WriteLine(KeyValurPair.Key);  
                // Lookup employees by Department  
                foreach (var item in emp[KeyValurPair.Key])  
                {  
                    Console.WriteLine($"{item.Name} {item.Department} {item.Country}");  
                }  
            }  
            Console.ReadLine();  
        }  
    }  
}  
  
class Employee  
{  
    public string Name { get; set; }  
    public string Department { get; set; }  
    public string Country { get; set; }  
}
```

LINQ Cast() Method

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Collections;
```

LINQ Cast Conversion Operator Example

```
namespace LINQExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            ArrayList obj = new ArrayList();  
            obj.Add("India");  
            obj.Add("USA");  
            obj.Add("UK");  
            obj.Add("Australia");  
            IEnumerable<string> result = obj.Cast<string>();  
            foreach (var item in result)  
            {  
                Console.WriteLine(item);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Collections;
```

Result of LINQ Cast Operator Example

```
namespace LINQExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            ArrayList obj = new ArrayList();  
            obj.Add("India");  
            obj.Add("USA");  
            obj.Add("UK");  
            obj.Add("Australia");  
            obj.Add(1);  
            IEnumerable<string> result = obj.Cast<string>();  
            foreach (var item in result)  
            {  
                Console.WriteLine(item);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Collections;
```

LINQ OfType() Method

```
namespace LINQExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            ArrayList obj = new ArrayList();  
            obj.Add("India");  
            obj.Add("USA");  
            obj.Add("UK");  
            obj.Add("Australia");  
            obj.Add(1);  
            IEnumerable<string> result = obj.Cast<string>();  
            foreach (var item in result)  
            {  
                Console.WriteLine(item);  
            }  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;
```

```
using System.Collections;
```

```
using System.Collections.Generic;
```

```
using System.Linq;
```

LINQ OfType Operator Example

```
namespace Linqtutorials
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            ArrayList obj = new ArrayList();
```

```
            obj.Add("India");
```

```
            obj.Add("USA");
```

```
            obj.Add("UK");
```

```
            obj.Add("Australia");
```

```
            obj.Add(1);
```

```
            IEnumerable<string> result = obj.OfType<string>();
```

```
            foreach (var item in result)
```

```
            {
```

```
                Console.WriteLine(item);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```


LINQ AsEnumerable() Method

- ***AsEnumerable()*** method is used to convert/cast specific types of given list items to its IEnumerable equivalent type.
- ***AsQueryable*** operator/method is useful for converting input list elements to IQueryable<T> list, and AsQueryable is a method of System.Linq.Queryable class.
- ***ToDictionary*** operator is useful to convert list/collection (IEnumerable<T>) items to a new dictionary object (Dictionary<TKey, TValue>), and it will optimize the list/collection items by getting only required values

Example of LINQ AsEnumerable() Method

```
using System;
using System.Linq;

namespace LINQExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] numarray = new int[] { 1, 2, 3, 4 };
            var result = numarray.AsEnumerable();
            foreach (var number in result)
            {
                Console.WriteLine(number);
            }
            Console.ReadLine();
        }
    }
}
```

```
using System;  
using System.Linq;  
using System.Collections.Generic;
```

LINQ AsQueryable Method

```
namespace LINQExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<Student> objStudent = new List<Student>()  
            {  
                new Student() { Name = "Suresh Dasari", Gender = "Male", Location="Chennai" },  
                new Student() { Name = "Rohini Alavala", Gender = "Female", Location="Chennai" }  
            };  
            IQueryable<Student> query = objStudent.AsQueryable().Where(student => student.Name.Contains("Alavala"));  
            foreach (var student in query)  
            {  
                Console.WriteLine(student.Name);  
            }  
            Console.ReadLine();  
        }  
    }  
    class Student  
    {  
        public string Name { get; set; }  
        public string Gender { get; set; }  
        public string Location { get; set; }  
    }  
}
```

```
using System;  
using System.Linq;  
using System.Collections.Generic;
```

LINQ ToDictionary() Method

```
namespace LINQExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<Student> objStudent = new List<Student>()  
            {  
                new Student() { Name = "Suresh Dasari", Gender = "Male", Location="Chennai" },  
                new Student() { Name = "Rohini Alavala", Gender = "Female", Location="Chennai" }  
            };  
            IQueryable<Student> query = objStudent.AsQueryable().Where(student => student.Name.Contains("Alavala"));  
            foreach (var student in query)  
            {  
                Console.WriteLine(student.Name);  
            }  
            Console.ReadLine();  
        }  
    }  
    class Student  
    {  
        public string Name { get; set; }  
        public string Gender { get; set; }  
        public string Location { get; set; }  
    }  
}
```

```
using System;  
using System.Linq;  
using System.Collections.Generic;
```

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            List<Student> objStudent = new List<Student>()
```

```
            {
```

```
                new Student() { Id=1,Name = "Suresh Dasari", Gender = "Male",Location="Chennai" },
```

```
                new Student() { Id=2,Name = "Rohini Alavala", Gender = "Female", Location="Chennai" }
```

```
            };
```

```
            var student = objStudent.ToDictionary(x => x.Id, x => x.Name);
```

```
            foreach (var stud in student)
```

```
            {
```

```
                Console.WriteLine(stud.Key + "\t" + stud.Value);
```

```
            }
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
    class Student
```

```
    {
```

```
        public int Id { get; set; }
```

```
        public string Name { get; set; }
```

```
        public string Gender { get; set; }
```

```
        public string Location { get; set; }
```

```
    }
```

```
}
```

LINQ ToDictionary Operator Example

LINQ Element Operators (FirstOrDefault, Last, ElementAt, SingleOrDefault)

- ***element*** operators are useful for returning a first/last element of the list, a single element from the collection, or a specific element based on index value from a collection.
- ***First()*** method/operator returns the first element from the sequence of items in the list/collection or the first element in the sequence of items in the list based on the specified condition.
- In case if there are no elements present in the list/collection based on specified conditions, then LINQ First() method will throw an error.

LINQ Element Operators (FirstOrDefault, Last, ElementAt, SingleOrDefault)

- **FirstOrDefault** operator is used to return the first element from the list/collection, and it's same as LINQ First operator, but the only difference is in case if the list returns no elements, then the LINQ FirstOrDefault method will return the default value.
- **Last()** method is useful to return the last element from the list/collection, and this LINQ Last() method will throw an exception if the list/collection returns no elements.
- **LastOrDefault()** method/operator is used to return the last element from the list/collection, and it's same as LINQ Last() method, but only the difference is it will return the default value in case if the list/collection contains no element.

<u>Operator</u>	<u>Description</u>
First	It returns the first element in sequence or the first element from the collection based on the specified condition.
FirstOrDefault	It's same as First, but it returns the default value if no element is found in the collection.
Last	It returns the last elements in sequence or the last element from the list based on matching criteria.
LastOrDefault	It's same as Last, but it returns a default value if no element is found in the collection.
ElementAt	It returns an element from the list based on the specified index position.
ElementAtOrDefault	It's same as ElementAt, but it returns the default value if no element is present at the specified index of the collection.
Single	It returns a single specific element from the collection.
SingleOrDefault	It's same as Single, but it returns a default value if no element is found in the collection.
DefaultEmpty	It returns the default value if the list or collection contains empty or null values.


```
using System;  
using System.Linq;  
using System.Collections.Generic;
```

LINQ First() Method

```
namespace LINQExamples  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            List<Student> objStudent = new List<Student>()  
            {  
                new Student() { Id=1,Name = "Suresh Dasari", Gender = "Male",Location="Chennai" },  
                new Student() { Id=2,Name = "Rohini Alavala", Gender = "Female", Location="Chennai" }  
            };  
            var student = objStudent.ToDictionary(x => x.Id, x => x.Name);  
            foreach (var stud in student)  
            {  
                Console.WriteLine(stud.Key + "\t" + stud.Value);  
            }  
            Console.ReadLine();  
        }  
    }  
    class Student  
    {  
        public int Id { get; set; }  
        public string Name { get; set; }  
        public string Gender { get; set; }  
        public string Location { get; set; }  
    }  
}
```

```
using System;
```

```
using System.Linq;
```

LINQ First() in Method Syntax Example

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            int[] objList = { 1, 2, 3, 4, 5 };
```

```
            int result = objList.First();
```

```
            Console.WriteLine("Element from the List: {0}", result);
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

using System;
using System.Linq;

Result of LINQ First() in Method Syntax

Example

```
namespace LINQExamples
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int[] objList = { 1, 2, 3, 4, 5 };  
            int result = (from l in objList select l).First();  
            Console.WriteLine("Element from the List: {0}", result);  
            Console.ReadLine();  
        }  
    }  
}
```

using System;

using System.Linq;

LINQ FirstOrDefault Method

namespace LINQExamples

{

class Program

{

static void Main(string[] args)

{

int[] objList = { 1, 2, 3, 4, 5 };

int result = (from l in objList select l).First();

Console.WriteLine("Element from the List: {0}", result);

Console.ReadLine();

}

}

}

```
using System;
```

```
using System.Linq;
```

Example of LINQ FirstOrDefault in Method Syntax

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            int[] objList = { 1, 2, 3, 4, 5 };
```

```
            int[] objVals = { };
```

```
            int result = objList.FirstOrDefault();
```

```
            int val = objVals.FirstOrDefault();
```

```
            Console.WriteLine("Element from the List1: {0}", result);
```

```
            Console.WriteLine("Element from the List2: {0}", val);
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

using System;

using System.Linq;

Example of LINQ FirstOrDefault in Query Syntax

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            int[] objList = { 1, 2, 3, 4, 5 };
```

```
            int[] objVals = { };
```

```
            int result = (from l in objList select l).FirstOrDefault();
```

```
            int val = (from x in objVals
```

```
                        select x).FirstOrDefault();
```

```
            Console.WriteLine("Element from the List1: {0}", result);
```

```
            Console.WriteLine("Element from the List2: {0}", val);
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```

using System;

using System.Linq;

LINQ Last() Method

namespace LINQExamples

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int[] objList = { 1, 2, 3, 4, 5 };  
            int result = objList.Last();  
            Console.WriteLine("Element from the List: {0}", result);  
            Console.ReadLine();  
        }  
    }  
}
```

```
using System;
```

```
using System.Linq;
```

Example of LINQ Last() in Query Syntax

```
namespace LINQExamples
```

```
{
```

```
    class Program
```

```
    {
```

```
        static void Main(string[] args)
```

```
        {
```

```
            int[] objList = { 1, 2, 3, 4, 5 };
```

```
            int result = (from l in objList select l).Last();
```

```
            Console.WriteLine("Element from the List: {0}", result);
```

```
            Console.ReadLine();
```

```
        }
```

```
    }
```

```
}
```


using System;

using System.Linq;

LINQ LastOrDefault() Method

namespace LINQExamples

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            int[] objList = { 1, 2, 3, 4, 5 };  
            int result = (from l in objList select l).Last();  
            Console.WriteLine("Element from the List: {0}", result);  
            Console.ReadLine();  
        }  
    }  
}
```