LINQ 32

Part 1 - What is LINQ - Slides
Part 2 - Writing LINQ Queries - Slides
Part 3 - Extension Methods - Slides
Part 4 - Aggregate Functions - Slides
Part 5 - Aggregate() Function - Slides
Part 6 - Restriction Operators - Slides
Part 7 - Projection Operators - Slides
Part 8 - SelectMany Operator - Slides
Part 9 - Select v/s SelectMany - Slides
Part 10 - Ordering Operators - Slides

Part 11 - Ordering Operators in LINQ - II - Slides
Part 12 - Partitioning Operators - Slides
Part 13 - Implement paging using skip and take operators - Slides
Part 14 - LINQ query deferred execution - Slides
Part 15 - Conversion Operators in LINQ - Slides
Part 16 - Cast and OfType operators in LINQ - Slides
Part 17 - AsEnumerable and AsQueryable in LINQ - Slides
Part 18 - GroupBy in LINQ - Slides
Part 19 - Group by multiple keys in linq - Slides
Part 20 - Element Operators in LINQ - Slides

Part 21 - Group Join in LINQ - Slides
Part 22 - Inner Join in LINQ - Slides
Part 23 - Difference between group join and inner join in linq - Slides
Part 24 - Left Outer Join in LINQ - Slides
Part 25 - Cross Join in LINQ - Slides
Part 26 - Set operators in LINQ - Slides
Part 27 - Union, Intersect and Except operators in LINQ - Slides
Part 28 - Generation Operators in LINQ - Slides
Part 29 - Concat operator in LINQ - Slides
Part 30 - SequenceEqual Operator in LINQ - Slides

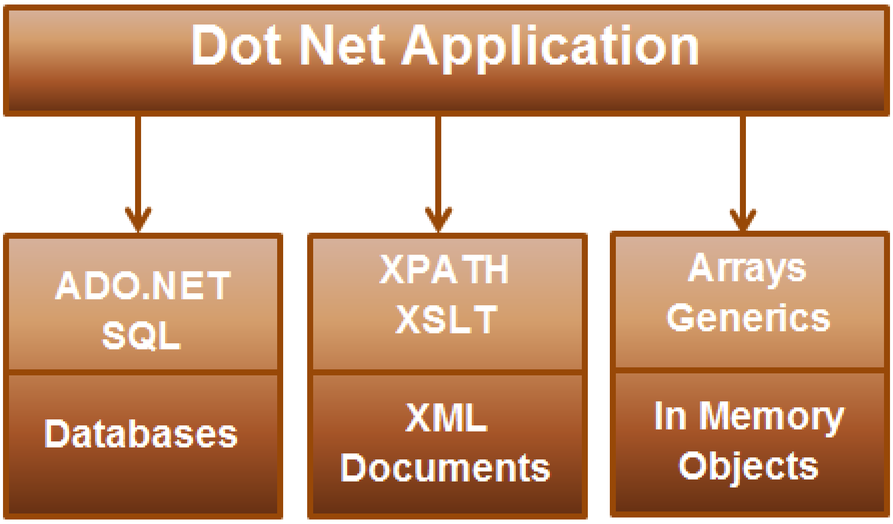
Part 31 - Quantifiers in LINQ - Slides
Part 32 - LinqPad - Slides

Part 1 – What is LINQ

What is LINQ

LINQ stands for **Language Integrated Query**. LINQ enables us to query any type of data store (SQL Server, XML documents, Objects in memory etc).

Why should we use LINQ and what are the benefits of using LINQ



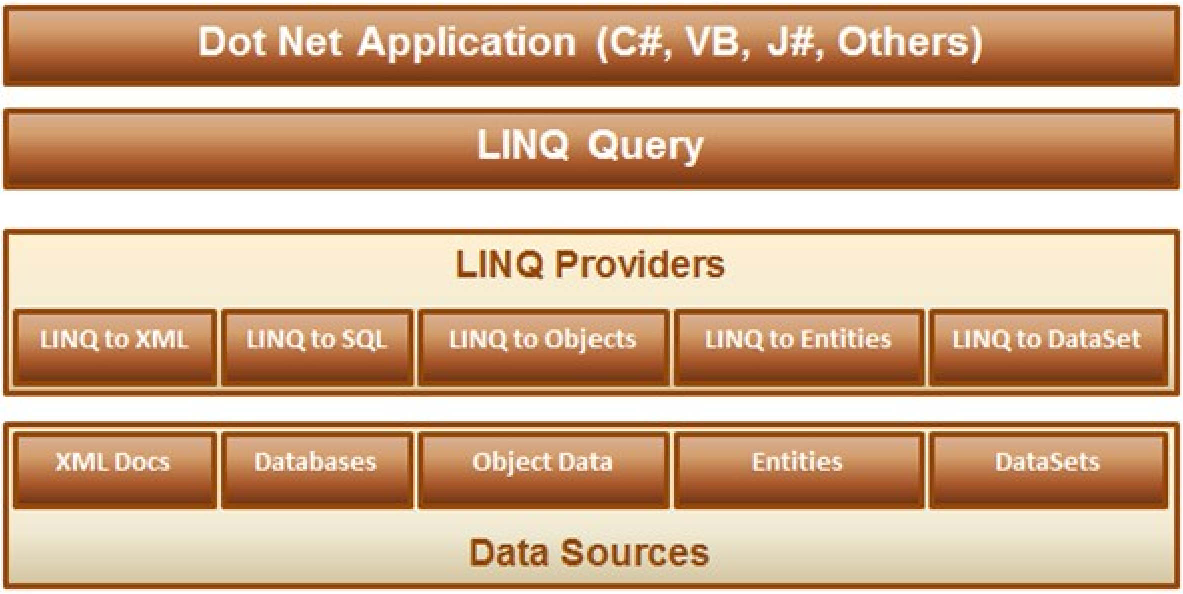
If the .NET application that is being developed
a) Requires data from SQL Server - Then the developer has to understand ADO.NET code and SQL specific to SQL Server Database
b) Requires data from an XML document - Then the developer has to understand XSLT & XPATH queries

c) Need to query objects in memory (List<Customer>, List<Order> etc) - Then the developer has to understand how to work with objects in memory

LINQ enables us to work with these different data sources using a similar coding style without having the need to know the syntax specific to the data source. In our upcoming videos we will discuss querying different data sources using LINQ.

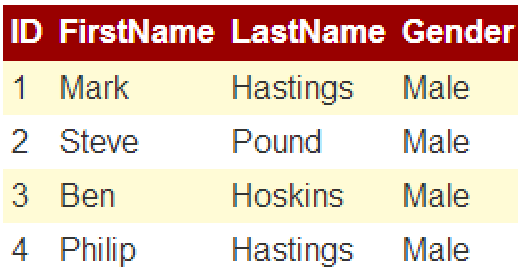
Another benefit of using LINQ is that it provides intellisense and compile time error checking.

LINQ Architecture & LINQ Providers



1. LINQ query can be written using any .NET supported programming language
2. LINQ provider is a component between the LINQ query and the actual data source, which converts the LINQ query into a format that the underlying data source can understand. For example LINQ to SQL provider converts a LINQ query to T-SQL that SQL Server database can understand.

For example, the application that we are developing should display male students in a GridView control as shown below.



ID	FirstName	LastName	Gender
1	Mark	Hastings	Male
2	Steve	Pound	Male
3	Ben	Hoskins	Male
4	Philip	Hastings	Male

To achieve this
Step 1: We first create the required table

```
Create Table Students
(
    ID int primary key identity,
    FirstName nvarchar(50),
    LastName nvarchar(50),
    Gender nvarchar(50)
)
GO

Insert into Students values ('Mark', 'Hastings', 'Male')
Insert into Students values ('Steve', 'Pound', 'Male')
Insert into Students values ('Ben', 'Hoskins', 'Male')
Insert into Students values ('Philip', 'Hastings', 'Male')
Insert into Students values ('Mary', 'Lambeth', 'Female')
GO
```

Step 2: Write the required ADO.NET code to retrieve data from SQL Server database as shown below.

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Data.SqlClient;
namespace Demo
{
    public partial class WebForm1 : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            string cs
= ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;
            SqlConnection con = new SqlConnection(cs);
            SqlCommand cmd = new SqlCommand
                ("Select ID, FirstName, LastName, Gender from Students where
Gender='Male'", con);
            List<Student> listStudents = new List<Student>();
            con.Open();
            SqlDataReader rdr = cmd.ExecuteReader();
            while (rdr.Read())
            {
                Student student = new Student();
                student.ID = Convert.ToInt32(rdr["ID"]);
                student.FirstName = rdr["FirstName"].ToString();
                student.LastName = rdr["LastName"].ToString();
                student.Gender = rdr["Gender"].ToString();

                listStudents.Add(student);
            }
            con.Close();

            GridView1.DataSource = listStudents;
            GridView1.DataBind();
        }
    }

    public class Student
    {
        public int ID { get; set; }
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public string Gender { get; set; }
    }
}
```

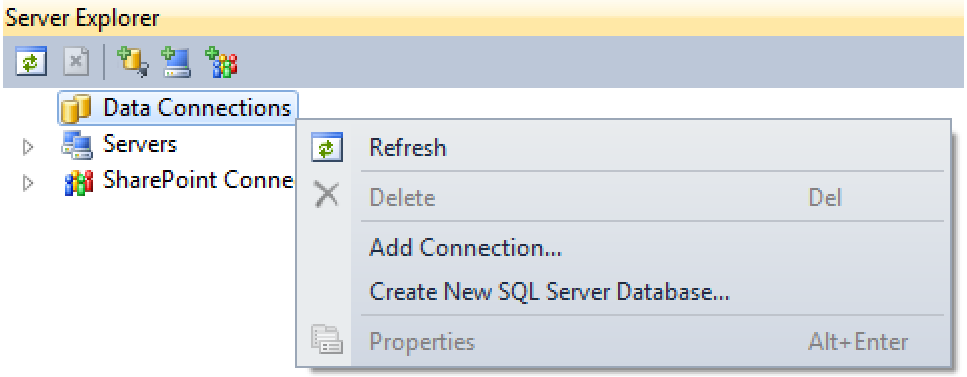
If we misspell table or column names in the SQL Query, we will not know about it at compile time. At run time the page crashes and that's when we will know about this error. Also notice that there is no **intellisense** when typing table and column names. Misspelled column names when reading from the reader will also cause the same problem. With LINQ we will have intellisense and compile time error checking.

Now let's achieve the same thing using LINQ to SQL.

Step 1: Create a new empty asp.net web application and name it **Demo**

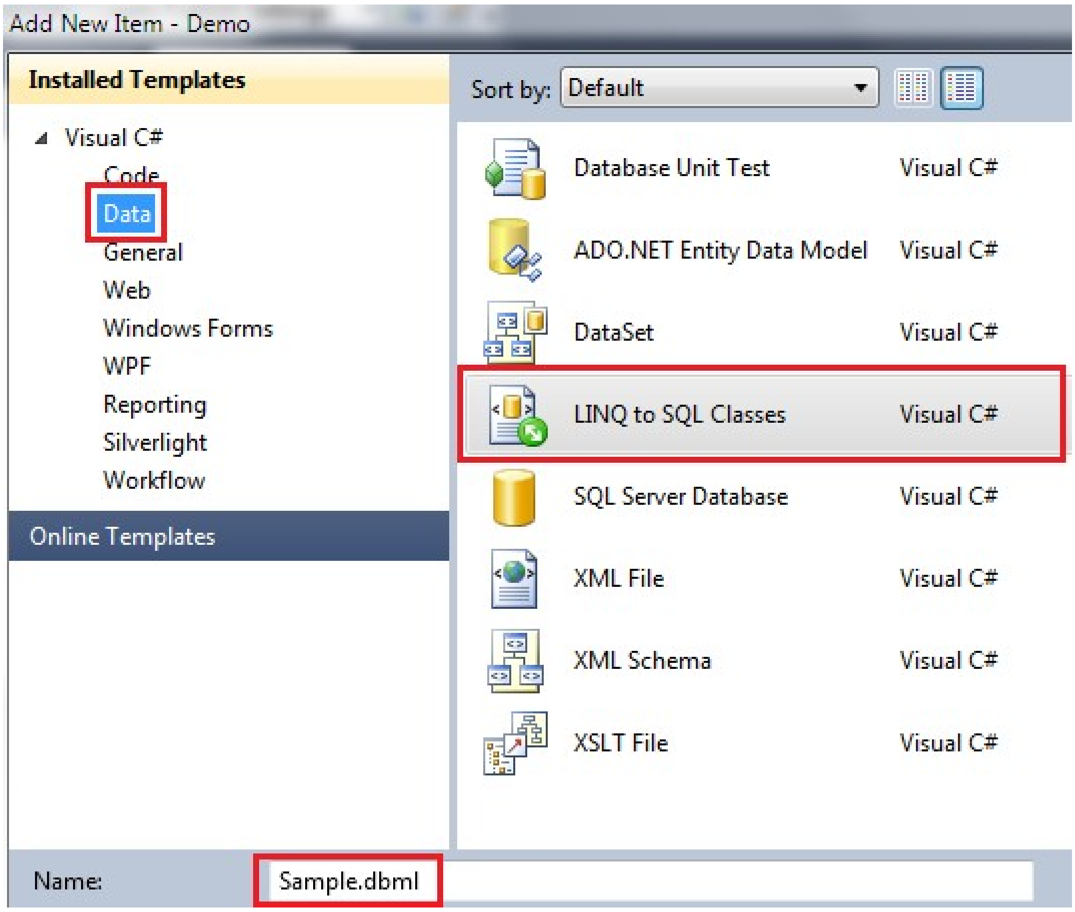
Step 2: Click on **"View"** menu item and select **"Server Explorer"**

Step 3: In **"Server Explorer"** window, right click on **"Data Connections"** and select **"Add Connection"** option



Step 4: Specify your SQL Server name and the credentials to connect to SQL Server. At this point we should be connected to SQL Server from Visual Studio.

Step 5: Adding **LINQ to SQL Classes**
a) Right click on the **"Demo"** project in solution explorer and select **"Add New Item"** option
b) In the **"Add New Item"** dialog box, select **"Data"** under **"Installed Templates"**
c) Select **"LINQ to SQL Classes"**
d) Set **Name = Sample.dbml**
e) Finally click **"Add"** button



Name: Sample.dbml

Step 6: From **"Server Explorer"** window drag and drop **"Students"** table onto **"Sample.dbml"** designer file.

Step 7: Add a webform. Drag and drop a gridview control.

Step 8: Copy and paste the following code in the code-behind file

```
using System;
using System.Linq;
namespace Demo
{
   public partial class WebForm1 : System.Web.UI.Page
   {
      protected void Page_Load(object sender, EventArgs e)
      {
         SampleDataContext dataContext = new SampleDataContext();
         GridView1.DataSource = from student in dataContext.Students
                                where student.Gender == "Male"
                                select student;
         GridView1.DataBind();
      }
   }
}
```

Notice that, with LINQ we are getting intellisense. If we misspell the table or column names we will get to know about them at compile time. Open SQL Profiler. Run the application, and notice the SQL Query that is generated.

Part 2 – Writing LINQ Queries

In this video, we will discuss **different ways of writing LINQ Queries**.

To write LINQ queries we use the **LINQ Standard Query Operators**. The following are a few Examples of Standard Query Operators
select
from
where
orderby
join
groupby

There are 2 ways to write LINQ queries using these Standard Query Operators
1. Using Lambda Expressions. We discussed Lambda Expressions in detail in Part 99 of C# Tutorial

2. Using SQL like query expressions

The **Standard Query Operators** are implemented as extension methods on IEnumerable<T> interface. We will discuss, what extension methods are and how to implement them in a later video session.

For now let's focus on the 2 ways of writing a LINQ query. From a performance perspective there is no difference between the two. Which one to use depends on your personal preference. But keep in mind, behind the scene, LINQ queries written using SQL like query expressions are translated into their lambda expressions before they are compiled.

We will use the following Student class in this demo. GetAllStudents() is a static method that returns List<Student>. Since List<T> implements IEnumerable<T>, the LINQ Standard Query Operators will be available and can be applied on List<Student>.

```
public class Student
{
   public int ID { get; set; }
   public string Name { get; set; }
   public string Gender { get; set; }

   public static List<Student> GetAllStudents()
   {
      List<Student> listStudents = new List<Student>();

      Student student1 = new Student
      {
         ID = 101,
         Name = "Mark",
         Gender = "Male"
      };
      listStudents.Add(student1);

      Student student2 = new Student
      {
         ID = 102,
            Name = "Mary",
            Gender = "Female"
        };
        listStudents.Add(student2);

        Student student3 = new Student
        {
            ID = 103,
            Name = "John",
            Gender = "Male"
        };
        listStudents.Add(student3);

        Student student4 = new Student
        {
            ID = 104,
            Name = "Steve",
            Gender = "Male"
        };
        listStudents.Add(student4);

        Student student5 = new Student
        {
            ID = 105,
            Name = "Pam",
            Gender = "Female"
        };
        listStudents.Add(student5);

        return listStudents;
    }
}
```

The **LINQ query** should return just the **Male** students.

LINQ query using Lambda Expressions.

```
IEnumerable<Student> students = Student.GetAllStudents()
    .Where(student => student.Gender == "Male");
```

LINQ query using using SQL like query expressions

```
IEnumerable<Student> students = from student in Student.GetAllStudents()
                                where student.Gender == "Male"
                                select student;
```

To bind the results of this LINQ query to a GridView

```
GridView1.DataSource = students;
GridView1.DataBind();
```

Part 3 – Extension M**In this video we will discuss**
1. What are Extension Methods
2. How to implement extension methods

What are Extension Methods
According to MSDN, Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.
Extension methods are a special kind of static method, but they are called as if they were instance methods on the extended type.

For client code written in C# and Visual Basic, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.

Let us understand what this definition actually means.
LINQ's standard query operators (select, where etc) are implemented in Enumerable class as extension methods on the IEnumerable<T> interface.

Now look at the following query

```
List<int> Numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

IEnumerable<int> EvenNumbers = Numbers.Where(n => n % 2 == 0);
```

In spite of **Where()** method not belonging to **List<T>** class, we are still able to use it as though it belong to **List<T>** class. This is possible because **Where()** method is implemented as extension method in **IEnumerable<T>** interface and **List<T>** implements **IEnumerable<T>** interface.

How to implement extension methods
We want to define a method in the string class (let's call it ChangeFirstLetterCase), which will change the case of the first letter of the string. For example, if the first letter of the string is lowercase the function should change it to uppercase and viceversa.

We want to be able to call this function on the string object as shown below.

```
string result = strName.ChangeFirstLetterCase();
```

Defining **ChangeFirstLetterCase**() method directly in the **string** class is not possible as we don't own the string class. It belongs to .NET framework. Another alternative is to write a wrapper class as shown below.

```
public class StringHelper
{
   public static string ChangeFirstLetterCase(string inputString)
   {
      if (inputString.Length > 0)
      {
         char[] charArray = inputString.ToCharArray();
         charArray[0] = char.IsUpper(charArray[0]) ?
            char.ToLower(charArray[0]) : char.ToUpper(charArray[0]);
            return new string(charArray);
        }

        return inputString;
    }

}
```

Wrapper class works, but the problem is, we cannot call **ChangeFirstLetterCase**() method using the following syntax.

```
string result = strName.ChangeFirstLetterCase();
```

Instead we have to call it as shown below.

```
string result = StringHelper.ChangeFirstLetterCase(strName);
```

Convert **ChangeFirstLetterCase**() method to an extension method to be able to call it using the following syntax, as though it belongs to string class.

```
string result = strName.ChangeFirstLetterCase();
```

To **convert ChangeFirstLetterCase() method to an extension method**, make the following 2 changes

1. Make StringHelper static class

2. The type the method extends should be passed as a first parameter with this keyword preceeding it.

With these 2 changes, we should be able to call this extension method in the same way we call an instance method. Notice that the extension method shows up in the intellisense as well, but with a different visual clue.

```
string result = strName.ChangeFirstLetterCase();
```

Please note that, we should still be able to call this extension method using wrapper class style syntax. In fact, behind the scene this is how the method actually gets called. Extension methods are just a syntactic sugar.

```
string result = StringHelper.ChangeFirstLetterCase(strName);
```

So, this means we should also be able to call LINQ extension methods (select, where etc), using wrapper class style syntax. Since all LINQ extension methods are defined in Enumerable class, the syntax will be as shown below.

```
List<int> Numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

IEnumerable<int> EvenNumbers = Enumerable.Where(Numbers, n => n % 2 == 0);
```

Part 4 - Aggregate Functions

LINQ Standard Query Operators also called as **LINQ extension methods** can be broadly classified into the following categories
Aggregate Operators
Grouping Operators
Restriction Operators
Projection Operators
Set Operators
Partitioning Operators
Conversion Operators
Element Operators
Ordering Operators
Generation Operators
Query Execution
Join Operators
Custom Sequence Operators
Quantifiers Operators
Miscellaneous Operators

In this video we will discuss the following **LINQ Aggregate** Operators
Min
Max
Sum
Count
Average
Aggregate (**Next Video**)

Example 1:

```
using System;
using System.Linq;
namespace Demo
{
   class Program
   {
      static void Main()
      {
         int[] Numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

         int smallestNumber = Numbers.Min();
         int smallestEvenNumber = Numbers.Where(n => n % 2 == 0).Min();

         int largestNumber = Numbers.Max();
         int largestEvenNumber = Numbers.Where(n => n % 2 == 0).Max();

         int sumOfAllNumbers = Numbers.Sum();
         int sumOfAllEvenNumbers = Numbers.Where(n => n % 2 == 0).Sum();

         int countOfAllNumbers = Numbers.Count();
         int countOfAllEvenNumbers = Numbers.Where(n => n % 2 == 0).Count();

         double averageOfAllNumbers = Numbers.Average();
```

```
            double averageOfAllEvenNumbers = Numbers.Where(n => n % 2 ==
0).Average();

            Console.WriteLine("Smallest Number = " + smallestNumber);
            Console.WriteLine("Smallest Even Number = " + smallestEvenNumber);

            Console.WriteLine("Largest Number = " + largestNumber);
            Console.WriteLine("Largest Even Number = " + largestEvenNumber);

            Console.WriteLine("Sum of All Numbers = " + sumOfAllNumbers);
            Console.WriteLine("Sum of All Even Numbers = " + sumOfAllEvenNumbers);

            Console.WriteLine("Count of All Numbers = " + countOfAllNumbers);
            Console.WriteLine("Count of All Even Numbers = " + countOfAllEvenNumbers);

            Console.WriteLine("Average of All Numbers = " + averageOfAllNumbers);
            Console.WriteLine("Average of All Even Numbers = " +
averageOfAllEvenNumbers);
        }
    }
}
```

Example 2:

```
using System;
using System.Linq;
namespace Demo
{
    class Program
    {
        static void Main()
        {
            string[] countries = { "India", "USA", "UK" };

            int minCount = countries.Min(x => x.Length);
            int maxCount = countries.Max(x => x.Length);

            Console.WriteLine
                  ("The shortest country name has {0} characters in its name", minCount);
            Console.WriteLine
                  ("The longest country name has {0} characters in its name", maxCount);
        }
    }
}
```

Part 5 – Aggregate() Function

In this video we will discuss the use of **Aggregate**() LINQ function. In Part 4 of LINQ Tutorial, we discussed the following functions.

Min
Max
Sum
Count
Average

Let us understand the use of **Aggregate**() function with examples.

Example 1: Consider the following string array.

```
string[] countries = { "India", "US", "UK", "Canada", "Australia" };
```

We want to combine all these strings into a single comma separated string. The output of the program should be as shown below.

```
India, US, UK, Canada, Australia
```

Without LINQ, the program will be as shown below.

```
using System;
namespace Demo
{
    class Program
    {
        static void Main()
        {
            string[] countries = { "India", "US", "UK", "Canada", "Australia" };

            string result = string.Empty;
            for (int i = 0; i < countries.Length; i++)
            {
                result = result + countries[i] + ", ";
            }

            int lastIndex = result.LastIndexOf(",");
            result = result.Remove(lastIndex);

            Console.WriteLine(result);
        }
    }
}
```

With LINQ Aggregate function

```
using System;
using System.Linq;
namespace Demo
{
    class Program
    {
        static void Main()
        {
            string[] countries = { "India", "US", "UK", "Canada", "Australia" };
            string result = countries.Aggregate((a, b) => a + ", " + b);

            Console.WriteLine(result);
        }
    }
}
```

How Aggregate() function works?
Step 1. First **"India"** is concatenated with **"US"** to produce result **"India, US"**
Step 2. Result in **Step 1** is then concatenated with **"UK"** to produce result **"India, US, UK"**
Step 3: Result in **Step 2** is then concatenated with **"Canada"** to produce result **"India, US, UK, Canada"**

This goes on until the last element in the array to produce the final single string **"India, US, UK, Canada, Australia"**

Example 2: Consider the following integer array
```
int[] Numbers = { 2, 3, 4, 5 };
```

Compute the product of all numbers

Without LINQ
```
using System;
namespace Demo
{
    class Program
    {
        static void Main()
        {
            int[] Numbers = { 2, 3, 4, 5 };

            int result = 1;
            foreach (int i in Numbers)
            {
                result = result * i;
            }

            Console.WriteLine(result);
        }
    }
}
```

With LINQ:
```
using System;
using System.Linq;
namespace Demo
{
    class Program
    {
        static void Main()
        {
            int[] Numbers = { 2, 3, 4, 5 };
            int result = Numbers.Aggregate((a, b) => a * b);

            Console.WriteLine(result);
        }
    }
}
```

How Aggregate() function works?
Step 1: Multiply **(2X3)** to produce result **6**
Step 2: Result **(6)** in **Step 1** is then multiplied with **4 (6X4)** to produce result **24**
Step 3: Result **(24)** in **Step 2** is then multiplied with **5 (24X5)** to produce final result **120**

Example 3: Consider the following integer array

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

One of the overloaded version of **Aggregate()** function has a **Seed** parameter. If we pass **10** as the value for Seed parameter

```
int result = Numbers.Aggregate(10, (a, b) => a * b);
```

1200 will be the result

Step 1: Multiply **(10X2)** to produce result **20**
Step 2: Result **(20)** in **Step 1** is then multiplied with **3 (20X3)** to produce result **60**
Step 3: Result **(60)** in **Step 2** is then multiplied with **4 (60X4)** to produce result **240**
Step 4: Result **(240)** in **Step 3** is then multiplied with **5 (240X5)** to produce final result **1200**

6 – Restriction Operators

The **WHERE** standard query operator belong to Restriction Operators category in LINQ. Just like SQL, the WHERE standard query operator in LINQ is used to filter rows. The filter expression is specified using a predicate.

The following are the **2 overloaded versions of WHERE** extension method in Enumerable class

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, bool> predicate);

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate);
```

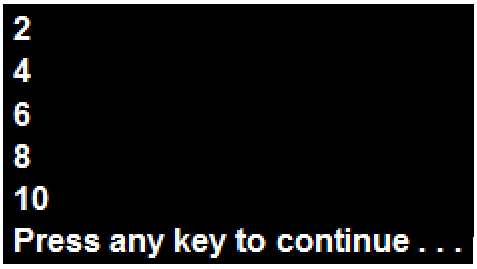
What is a Predicate?
A predicate is a function to test each element for a condition

In the following example, the Lambda expression (num => num % 2 == 0) runs for each element in List<int>. If the number is divisible by 2, then a boolean value true is returned otherwise false.

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace Demo
{
    class Program
    {
        static void Main()
        {
            List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            IEnumerable<int> evenNumbers = numbers.Where(num => num % 2 == 0);

            foreach (int evenNumber in evenNumbers)
            {
                Console.WriteLine(evenNumber);
            }
        }
    }
}

// Using SQL like syntax
IEnumerable<int> evenNumbers = from num in numbers
                                where num % 2 == 0
                                select num;
```

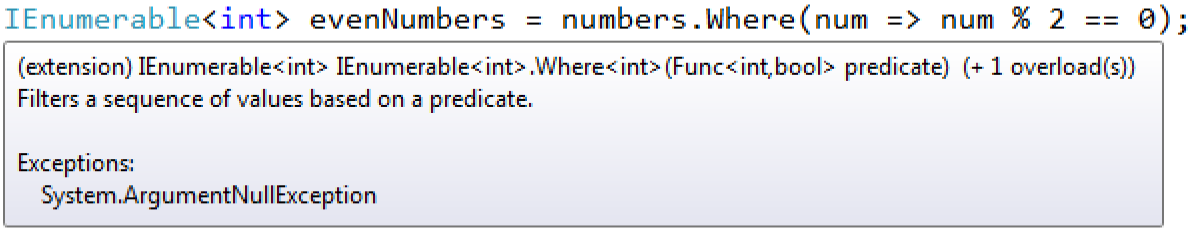
Note: The where query operator is optional.

The program prints all the even numbers



```
2
4
6
8
10
Press any key to continue . . .
```

When you hover the mouse ove **WHERE** method in the above example, visual studio intellisense shows the following. Notice that in this case, the predicate expects an int input parameter and returns a boolean value. The lambda expression that is passed operates on an int type and should return boolean, otherwise there will be compile time error.



```
IEnumerable<int> evenNumbers = numbers.Where(num => num % 2 == 0);
```

```
(extension) IEnumerable<int> IEnumerable<int>.Where<int>(Func<int,bool> predicate)  (+ 1 overload(s))
Filters a sequence of values based on a predicate.

Exceptions:
  System.ArgumentNullException
```

So this means, the line below from the above example

```
IEnumerable<int> evenNumbers = numbers.Where(num => num % 2 == 0);
```

can be rewritten as shown below

```
Func<int, bool> predicate = i => i % 2 == 0;
IEnumerable<int> evenNumbers = numbers.Where(predicate);
```

or like below

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

namespace Demo
{
    class Program
    {
        static void Main()
        {
            List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            IEnumerable<int> evenNumbers = numbers.Where(num => IsEven(num));

            foreach (int evenNumber in evenNumbers)
            {
                Console.WriteLine(evenNumber);
            }
        }

        public static bool IsEven(int number)
        {
            if (number % 2 == 0)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
    }
}
```

Example 2:
The int parameter of the predicate function represents the index of the source element

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, int, bool> predicate);
```

The following program prints the index position of all the even numbers

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace Demo
{
    class Program
    {
        static void Main()
        {
            List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            IEnumerable<int> evenNumberIndexPositions = numbers
                .Select((num, index) => new { Number = num, Index = index })
                .Where(x => x.Number % 2 == 0)
                .Select(x => x.Index);

            foreach (int evenNumber in evenNumberIndexPositions)
            {
                Console.WriteLine(evenNumber);
            }
        }
    }
}
```

Example 3:
Use the following SQL to create **Departments** and **Employees** tables

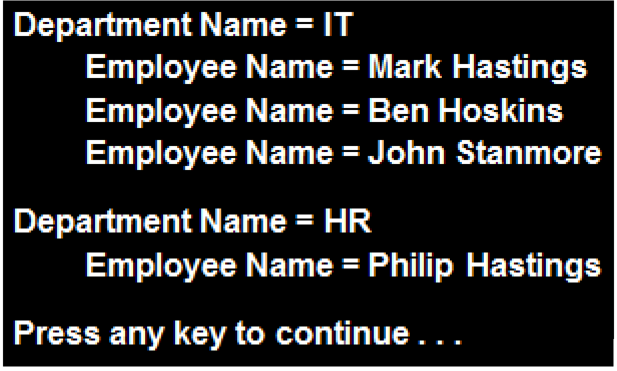
```
Create table Departments
(
     ID int primary key identity,
     Name nvarchar(50),
     Location nvarchar(50)
)
GO

Create table Employees
(
     ID int primary key identity,
     FirstName nvarchar(50),
    LastName nvarchar(50),
    Gender nvarchar(50),
    Salary int,
    DepartmentId int foreign key references Departments(Id)
)
GO

Insert into Departments values ('IT', 'New York')
Insert into Departments values ('HR', 'London')
Insert into Departments values ('Payroll', 'Sydney')
GO

Insert into Employees values ('Mark', 'Hastings', 'Male', 60000, 1)
Insert into Employees values ('Steve', 'Pound', 'Male', 45000, 3)
Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000, 1)
Insert into Employees values ('Philip', 'Hastings', 'Male', 45000, 2)
Insert into Employees values ('Mary', 'Lambeth', 'Female', 30000, 2)
Insert into Employees values ('Valarie', 'Vikings', 'Female', 35000, 3)
Insert into Employees values ('John', 'Stanmore', 'Male', 80000, 1)
GO
```

Add an **ADO.NET entity data model** based on the above 2 tables.

Write a LINQ query to retrieve **IT and HR department names and all the male employees** with in these 2 departments.

```
using System;
using System.Collections.Generic;
using System.Linq;
namespace Demo
{
    class Program
    {
        static void Main()
        {
            EmployeeDBContext context = new EmployeeDBContext();

            IEnumerable<Department> departments = context.Departments
                .Where(dept => dept.Name == "IT" || dept.Name == "HR");

            foreach (Department department in departments)
            {
                Console.WriteLine("Department Name = " + department.Name);
                foreach (Employee employee in department
                    .Employees.Where(emp => emp.Gender == "Male"))
                {
                    Console.WriteLine("\tEmployee Name = " + employee.FirstName
                        + " " + employee.LastName);
                }
                Console.WriteLine();
            }
        }
    }
}
```

Output:



```
Department Name = IT
        Employee Name = Mark Hastings
        Employee Name = Ben Hoskins
        Employee Name = John Stanmore

Department Name = HR
        Employee Name = Philip Hastings

Press any key to continue . . .
```

7 – Projection Operators (Select Operator)

The following 2 standard LINQ query operators belong to **Projection Operators** category.
Select
SelectMany

Projection Operators (Select & SelectMany) are used to transform the results of a query. In this video we will discuss **Select** operator and in a later video session we will discuss **SelectMany** operator.

Select clause in SQL allows to specify what columns we want to retrieve. In a similar fashion LINQ SELECT standard query operator allows us to specify what properties we want to retrieve. It also allows us to perform calculations.

For example, you may have a collection of Employee objects. The following are the properties of the **Employee** class.
EmployeeID
FirstName
LastName
AnnualSalay
Gender

Now using the SELECT projection operator
1. We can select just **EmployeeID** property OR
2. We can select multiple properties (**FirstName & Gender**) into an anonymous type OR
3. Perform calculations
  a) MonthlySalary = AnnualSalay/12
  b) FullName = FirstName + " " + LastName

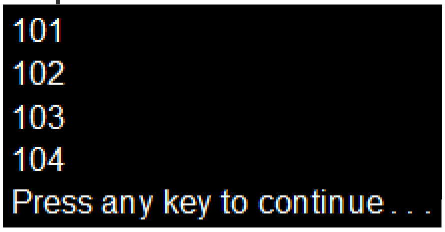
We will be using the following **Employee** class for this demo.

```
public class Employee
{
   public int EmployeeID { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public string Gender { get; set; }
   public int AnnualSalary { get; set; }

   public static List<Employee> GetAllEmployees()
   {
      List<Employee> listEmployees = new List<Employee>
      {
         new Employee
         {
            EmployeeID = 101,
            FirstName = "Tom",
            LastName = "Daely",
            Gender = "Male",
            AnnualSalary = 60000
         },
         new Employee
         {
            EmployeeID = 102,
            FirstName = "Mike",
            LastName = "Mist",
            Gender = "Male",
            AnnualSalary = 72000
         },
         new Employee
         {
            EmployeeID = 103,
            FirstName = "Mary",
            LastName = "Lambeth",
            Gender = "Female",
            AnnualSalary = 48000
         },
         new Employee
         {
            EmployeeID = 104,
            FirstName = "Pam",
            LastName = "Penny",
            Gender = "Female",
            AnnualSalary = 84000
         },
      };

      return listEmployees;
   }
}
```

Example 1: Retrieves just the **EmployeeID** property of all employees

```
IEnumerable<int> employeeIds = Employee.GetAllEmployees()
    .Select(emp => emp.EmployeeID);
foreach (int id in employeeIds)
{
    Console.WriteLine(id);
}
```

Output:

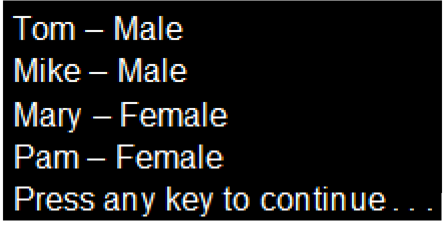


```
101
102
103
104
Press any key to continue . . .
```

Example 2: Projects **FirstName & Gender** properties of all employees into **anonymous type**.

```
var result = Employee.GetAllEmployees().Select(emp => new
                    {
                        FirstName = emp.FirstName,
                        Gender = emp.Gender
                    });
foreach (var v in result)
{
    Console.WriteLine(v.FirstName + " - " + v.Gender);
}
```

Output:

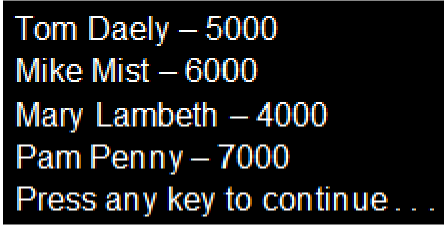


```
Tom – Male
Mike – Male
Mary – Female
Pam – Female
Press any key to continue . . .
```

Example 3: Computes **FullName and MonthlySalay** of all employees and projects these 2 new computed properties into anonymous type.

```
var result = Employee.GetAllEmployees().Select(emp => new
{
    FullName = emp.FirstName + " " + emp.LastName,
    MonthlySalary = emp.AnnualSalary / 12
});

foreach (var v in result)
{
    Console.WriteLine(v.FullName + " - " + v.MonthlySalary);
}
```

Output:

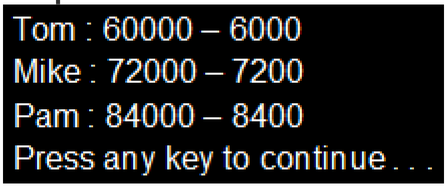


```
Tom Daely – 5000
Mike Mist – 6000
Mary Lambeth – 4000
Pam Penny – 7000
Press any key to continue . . .
```

Example 4: Give **10% bonus** to all employees whose annual salary is greater than **50000** and project all such employee's **FirstName, AnnualSalay and Bonus** into anonymous type.

```
var result = Employee.GetAllEmployees()
                .Where(emp => emp.AnnualSalary > 50000)
                .Select(emp => new
                 {
                    Name = emp.FirstName,
                    Salary = emp.AnnualSalary,
                    Bonus = emp.AnnualSalary * .1
                 });

foreach (var v in result)
{
    Console.WriteLine(v.Name + " : " + v.Salary + " - " + v.Bonus);
}
```

Output:



```
Tom : 60000 – 6000
Mike : 72000 – 7200
Pam : 84000 – 8400
Press any key to continue . . .
```

8 – SelectMany Operator

SelectMany Operator belong to **Projection Operators** category. It is used to project each element of a sequence to an **IEnumerable<T>** and flattens the resulting sequences **into one sequence**.

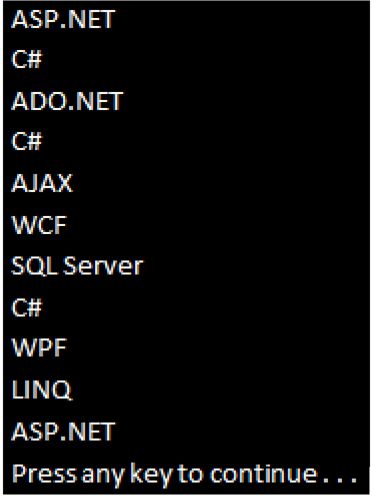
Let us understand this with an example. Consider the following **Student** class. **Subjects** property in this class is a **collection of strings**.

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

   public static List<Student> GetAllStudetns()
   {
      List<Student> listStudents = new List<Student>
      {
         new Student
         {
            Name = "Tom",
            Gender = "Male",
            Subjects = new List<string> { "ASP.NET", "C#" }
         },
         new Student
         {
            Name = "Mike",
            Gender = "Male",
            Subjects = new List<string> { "ADO.NET", "C#", "AJAX" }
         },
         new Student
         {
            Name = "Pam",
            Gender = "Female",
            Subjects = new List<string> { "WCF", "SQL Server", "C#" }
         },
         new Student
         {
            Name = "Mary",
            Gender = "Female",
            Subjects = new List<string> { "WPF", "LINQ", "ASP.NET" }
         },
      };

      return listStudents;
   }
}
```

Example 1: Projects all subject strings of a given a student to an **IEnumerable<string>**. In this example since we have 4 students, there will be 4 IEnumerable<string> sequences, which are then flattened to form a single sequence i.e a single IEnumerable<string> sequence.

```
IEnumerable<string> allSubjects = Student.GetAllStudetns().SelectMany(s =>
s.Subjects);
foreach (string subject in allSubjects)
{
    Console.WriteLine(subject);
}
```

Output:



```
ASP.NET
C#
ADO.NET
C#
AJAX
WCF
SQL Server
C#
WPF
LINQ
ASP.NET
Press any key to continue . . .
```

Example 2: Rewrite **Example1** using **SQL like syntax.** When using SQL like syntax style, we don't use SelectMany, instead we will have an additional from clause, which will get it's data from the results of the first from clause.

```
IEnumerable<string> allSubjects = from student in Student.GetAllStudetns()
                                         from subject in student.Subjects
                                         select subject;

foreach (string subject in allSubjects)
{
    Console.WriteLine(subject);
}
```

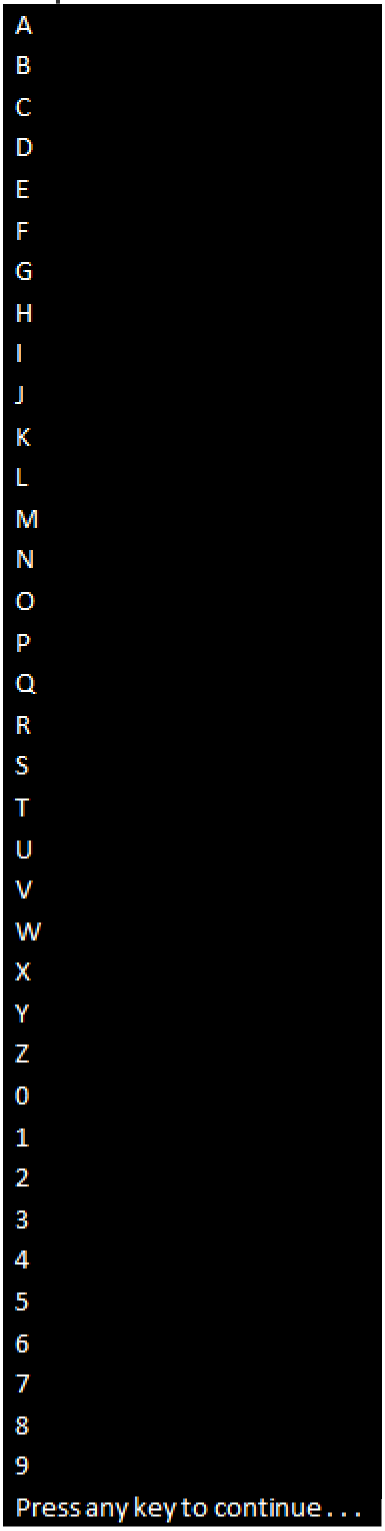
Output:
Same output as in **Example 1**

Example 3: Projects each string to an **IEnumerable<char>**. In this example since we have 2 strings, there will be 2 IEnumerable<char> sequences, which are then flattened to form a single sequence i.e a single IEnumerable<char> sequence.

```
string[] stringArray =
{
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
    "0123456789"
};

IEnumerable<char> result = stringArray.SelectMany(s => s);

foreach (char c in result)
```

```
{
    Console.WriteLine(c);
}
```

Output:



```
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y
Z
0
1
2
3
4
5
6
7
8
9
Press any key to continue . . .
```

Example 4: Rewrite **Example3** using **SQL like syntax.**

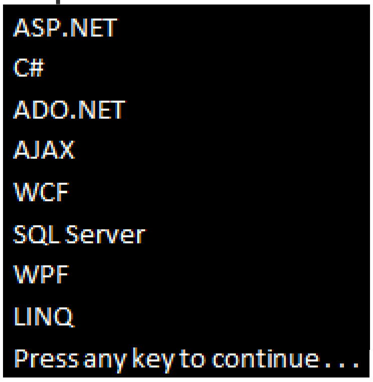
```
string[] stringArray =
{
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
    "0123456789"
};

IEnumerable<char> result = from s in stringArray
                           from c in s
                           select c;

foreach (char c in result)
{
    Console.WriteLine(c);
}
```

Output:
Same output as in **Example 3**

Example 5: Selects only the distinct subjects

```
IEnumerable<string> allSubjects = Student.GetAllStudetns()
                                          .SelectMany(s => s.Subjects).Distinct();
foreach (string subject in allSubjects)
{
    Console.WriteLine(subject);
}
```

Output:



```
ASP.NET
C#
ADO.NET
AJAX
WCF
SQL Server
WPF
LINQ
Press any key to continue . . .
```

Example 6: Rewrite **Example 5** using **SQL like syntax.**

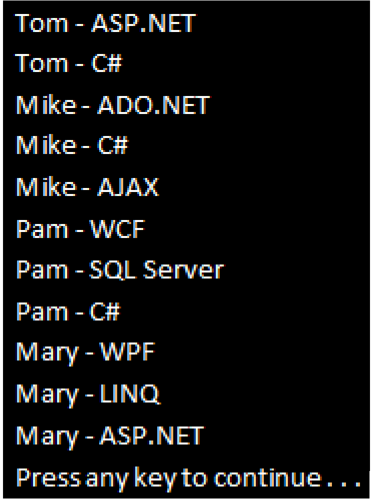
```
IEnumerable<string> allSubjects = (from student in Student.GetAllStudetns()
                                   from subject in student.Subjects
                                   select subject).Distinct();

foreach (string subject in allSubjects)
{
    Console.WriteLine(subject);
}
```

Output:
Same output as in **Example 5**

Example 7: Selects student name along with all the subjects

```
var result = Student.GetAllStudetns().SelectMany(s => s.Subjects, (student, subject) =>
    new { StudentName = student.Name, Subject = subject });

foreach (var v in result)
{
    Console.WriteLine(v.StudentName + " - " + v.Subject);
}
```

Output:



```
Tom - ASP.NET
Tom - C#
Mike - ADO.NET
Mike - C#
Mike - AJAX
Pam - WCF
Pam - SQL Server
Pam - C#
Mary - WPF
Mary - LINQ
Mary - ASP.NET
Press any key to continue . . .
```

Example 8: Rewrite **Example 7** using **SQL like syntax.**

```
var result = from student in Student.GetAllStudetns()
                 from subject in student.Subjects
                 select new { StudnetName = student.Name, Subject = subject };

foreach (var v in result)
{
    Console.WriteLine(v.StudnetName + " - " + v.Subject);
}
```

Output:

Same output as in **Example 7**

Part 9 – Select vs SelectMany

Let us understand the **difference between Select and SelectMany** with an example.

We will be using the following **Student** class in this demo. **Subjects** property in this class is a collection of strings.

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

    public static List<Student> GetAllStudetns()
    {
        List<Student> listStudents = new List<Student>
        {
            new Student
            {
                Name = "Tom",
                Gender = "Male",
                Subjects = new List<string> { "ASP.NET", "C#" }
            },
            new Student
            {
                Name = "Mike",
                Gender = "Male",
                Subjects = new List<string> { "ADO.NET", "C#", "AJAX" }
            },
            new Student
            {
                Name = "Pam",
                Gender = "Female",
                Subjects = new List<string> { "WCF", "SQL Server", "C#" }
            },
            new Student
            {
                Name = "Mary",
                Gender = "Female",
                Subjects = new List<string> { "WPF", "LINQ", "ASP.NET" }
            },
        };

        return listStudents;
    }
}
```

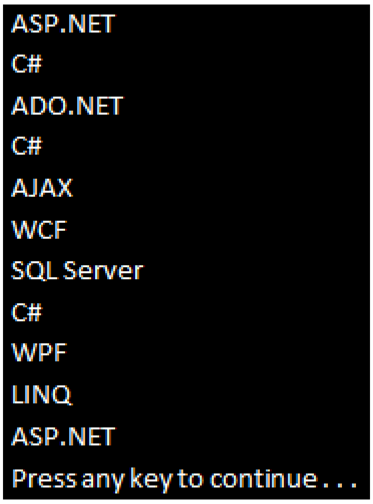
In this example, the **Select()** method returns **List of List<string>**. To print all the subjects we will have to use **2 nested foreach loops**.

```
IEnumerable<List<string>> result = Student.GetAllStudetns().Select(s => s.Subjects);
foreach (List<string> stringList in result)
{
    foreach (string str in stringList)
    {
        Console.WriteLine(str);
    }
}
```

SelectMany() on the other hand, flattens queries that return lists of lists into a **single list.** So in this case to print all the subjects we have to use just one foreach loop.

```
IEnumerable<string> result = Student.GetAllStudetns().SelectMany(s => s.Subjects);
foreach (string str in result)
{
   Console.WriteLine(str);
}
```

Output:



```
ASP.NET
C#
ADO.NET
C#
AJAX
WCF
SQL Server
C#
WPF
LINQ
ASP.NET
Press any key to continue . . .
```

Part 10 - Ordering Operators

The following **5 standard LINQ query operators belong to Ordering Operators** category

OrderBy
OrderByDescending
ThenBy
ThenByDescending
Reverse

OrderBy, OrderByDescending, ThenBy, and **ThenByDescending** can be used to sort data. **Reverse** method simply reverses the items in a given collection.

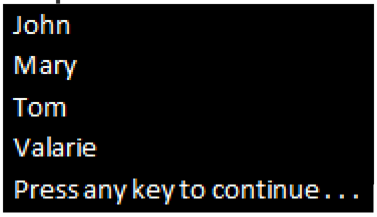
We will use the following **Student** class in this demo.

```
public class Student
{
    public int StudentID { get; set; }
    public string Name { get; set; }
    public int TotalMarks { get; set; }

    public static List<Student> GetAllStudents()
    {
        List<Student> listStudents = new List<Student>
        {
            new Student
            {
                StudentID= 101,
                Name = "Tom",
                TotalMarks = 800
            },
            new Student
            {
                StudentID= 102,
                Name = "Mary",
                TotalMarks = 900
            },
            new Student
            {
                StudentID= 103,
                Name = "Valarie",
                TotalMarks = 800
            },
            new Student
            {
                StudentID= 104,
                Name = "John",
                TotalMarks = 800
            },
        };

        return listStudents;
    }
}
```

Example 1: Sort **Students** by **Name** in **ascending** order

```
IEnumerable<Student> result = Student.GetAllStudents().OrderBy(s => s.Name);
foreach (Student student in result)
{
   Console.WriteLine(student.Name);
}
```

Output:



```
John
Mary
Tom
Valarie
Press any key to continue . . .
```

Example 2: Rewrite **Example 1** using **SQL like** syntax

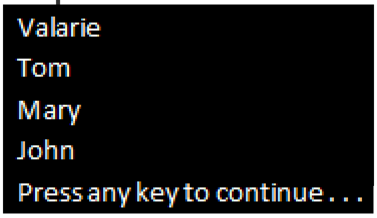
```
IEnumerable<Student> result = from student in Student.GetAllStudents()
                                       orderby student.Name
                                       select student;

foreach (Student student in result)
{
   Console.WriteLine(student.Name);
}
```

Output:
Same as in **Example 1**

Example 3: Sort **Students** by **Name** in **descending** order

```
IEnumerable<Student> result = Student.GetAllStudents().OrderByDescending(s =>
s.Name);
foreach (Student student in result)
{
   Console.WriteLine(student.Name);
}
```

Output:



```
Valarie
Tom
Mary
John
Press any key to continue . . .
```

Example 4: Rewrite **Example 3** using **SQL like** syntax

```
IEnumerable<Student> result = from student in Student.GetAllStudents()
                                       orderby student.Name descending
                                       select student;

foreach (Student student in result)
{
   Console.WriteLine(student.Name);
}
```

Output:
Same as in **Example 1**

11 – Ordering Operators Part II

This is continuation to Part 10. Please watch Part 10 before proceeding.

The following 5 standard LINQ query operators belong to Ordering Operators category
OrderBy
OrderByDescending
ThenBy
ThenByDescending
Reverse

In Part 10, we discussed **OrderBy** & **OrderByDescending** operators. In this video we will discuss
ThenBy
ThenByDescending
Reverse

OrderBy, OrderByDescending, ThenBy, and ThenByDescending can be used to sort data. Reverse method simply reverses the items in a given collection.

We will use the following **Student** class in this demo.

```
public class Student
{
    public int StudentID { get; set; }
    public string Name { get; set; }
    public int TotalMarks { get; set; }

    public static List<Student> GetAllStudetns()
    {
        List<Student> listStudents = new List<Student>
        {
            new Student
            {
                StudentID= 101,
                Name = "Tom",
                TotalMarks = 800
            },
            new Student
            {
                StudentID= 102,
                Name = "Mary",
                TotalMarks = 900
            },
            new Student
            {
                StudentID= 103,
                Name = "Pam",
                TotalMarks = 800
            },
            new Student
            {
                StudentID= 104,
                Name = "John",
                TotalMarks = 800
            },
            new Student
            {
                StudentID= 105,
                Name = "John",
                TotalMarks = 800
            },
        };

        return listStudents;
    }
}
```

OrderBy or **OrderByDescending** work fine when we want to sort a collection just by one value or expression.

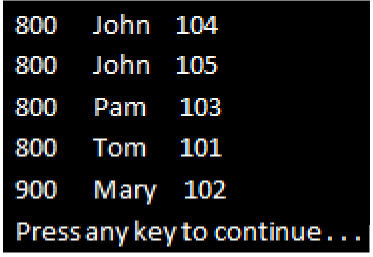
If want to sort by more than one value or expression, that's when we use **ThenBy** or **ThenByDescending** along with **OrderBy** or **OrderByDescending.**

OrderBy or **OrderByDescending** performs the primary sort. **ThenBy** or **ThenByDescending** is used for adding secondary sort. Secondary Sort operators (**ThenBy** or **ThenByDescending**) can be used more than once in the same LINQ query.

Example 1:
a) Sorts **Students** first by **TotalMarks** in ascending order(Primary Sort)
b) The 4 Students with **TotalMarks** of **800,** will then be sorted by Name in ascending order (First Secondary Sort)
c) The **2 Students** with **Name** of **John**, will then be sorted by **StudentID** in ascending order (Second Secondary Sort)

```
IEnumerable<Student> result = Student.GetAllStudetns()
    .OrderBy(s => s.TotalMarks).ThenBy(s => s.Name).ThenBy(s => s.StudentID);
foreach (Student student in result)
{
    Console.WriteLine(student.TotalMarks + "\t" + student.Name + "\t" +
student.StudentID);
}
```

Output:



```
800    John   104
800    John   105
800    Pam    103
800    Tom    101
900    Mary   102
Press any key to continue . . .
```

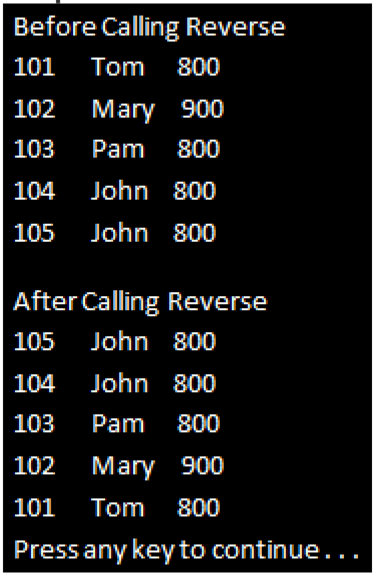
Example 2: Rewrite **Example 1** using **SQL** like syntax. With SQL like syntax we donot use **ThenBy** or **ThenByDescending,** instead we specify the sort expressions using a comma separated list. The first sort expression will be used for primary sort and the subsequent sort expressions for secondary sort.

```
IEnumerable<Student> result = from student in Student.GetAllStudetns()
                                           orderby student.TotalMarks, student.Name,
student.StudentID
                                           select student;
foreach (Student student in result)
{
   Console.WriteLine(student.TotalMarks + "\t" + student.Name + "\t" +
student.StudentID);
}
```

Example 3: Reverses the items in the collection.

```
IEnumerable<Student> students = Student.GetAllStudetns();

Console.WriteLine("Before calling Reverse");
foreach (Student s in students)
{
   Console.WriteLine(s.StudentID + "\t" + s.Name + "\t" + s.TotalMarks);
}

Console.WriteLine();
IEnumerable<Student> result = students.Reverse();

Console.WriteLine("After calling Reverse");
foreach (Student s in result)
{
   Console.WriteLine(s.StudentID + "\t" + s.Name + "\t" + s.TotalMarks);
}
```

Output:



```
Before Calling Reverse
101     Tom     800
102     Mary    900
103     Pam     800
104     John    800
105     John    800

After Calling Reverse
105     John    800
104     John    800
103     Pam     800
102     Mary    900
101     Tom     800
Press any key to continue . . .
```

12 – Partitioning Operartors (Skip and Take)

The following 4 standard LINQ query operators belong to Partitioning Operators category
Take
Skip
TakeWhile
SkipWhile

Take method returns a specified number of elements from the start of the collection. The number of items to return is specified using the count parameter this method expects.

Skip method skips a specified number of elements in a collection and then returns the remaining elements. The number of items to skip is specified using the count parameter this method expects.

Please Note: For the same argument value, the Skip method returns all of the items that the Take method would not return.

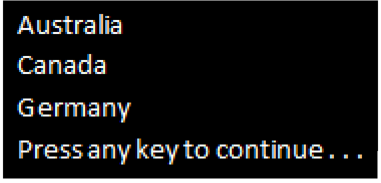
TakeWhile method returns elements from a collection as long as the given condition specified by the predicate is true.

SkipWhile method skips elements in a collection as long as the given condition specified by the predicate is true, and then returns the remaining elements.

Example 1: Retrieves only the first 3 countries of the array.

```
string[] countries = { "Australia", "Canada", "Germany", "US", "India", "UK", "Italy" };

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

foreach (string country in result)
{
    Console.WriteLine(country);
}
```

Output:



```
Australia
Canada
Germany
Press any key to continue . . .
```

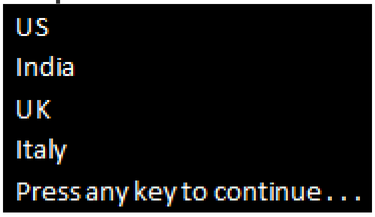
Example 2: Rewrite **Example 1** using SQL like syntax

```
string[] countries = { "Australia", "Canada", "Germany", "US", "India", "UK", "Italy" };

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

foreach (string country in result)
{
    Console.WriteLine(country);
}
```

Example 3: Skips the first 3 countries and retrieves the rest of them

```
string[] countries = { "Australia", "Canada", "Germany", "US", "India", "UK", "Italy" };

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

foreach (string country in result)
{
    Console.WriteLine(country);
}
```

Output:

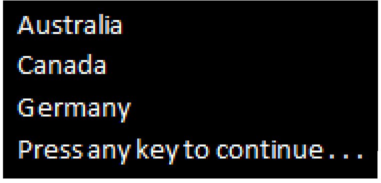


```
US
India
UK
Italy
Press any key to continue . . .
```

Example 4: Return countries starting from the beginning of the array until a country name is hit that does not have length greater than 2 characters.

```
string[] countries = { "Australia", "Canada", "Germany", "US", "India", "UK", "Italy" };

IEnumerable<string> result = countries.TakeWhile(s => s.Length > 2);

foreach (string country in result)
{
    Console.WriteLine(country);
}
```

Output:

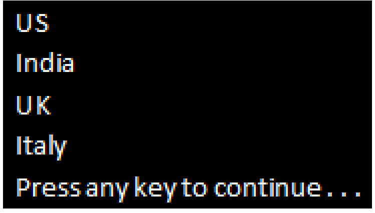


```
Australia
Canada
Germany
Press any key to continue . . .
```

Example 5: Skip elements starting from the beginning of the array, until a country name is hit that does not have length greater than 2 characters, and then return the remaining elements.

```
string[] countries = { "Australia", "Canada", "Germany", "US", "India", "UK", "Italy" };

IEnumerable<string> result = countries.SkipWhile(s => s.Length > 2);

foreach (string country in result)
{
    Console.WriteLine(country);
}
```

Output:



```
US
India
UK
Italy
Press any key to continue . . .
```

13 – Implementing Paging using Skip and Take

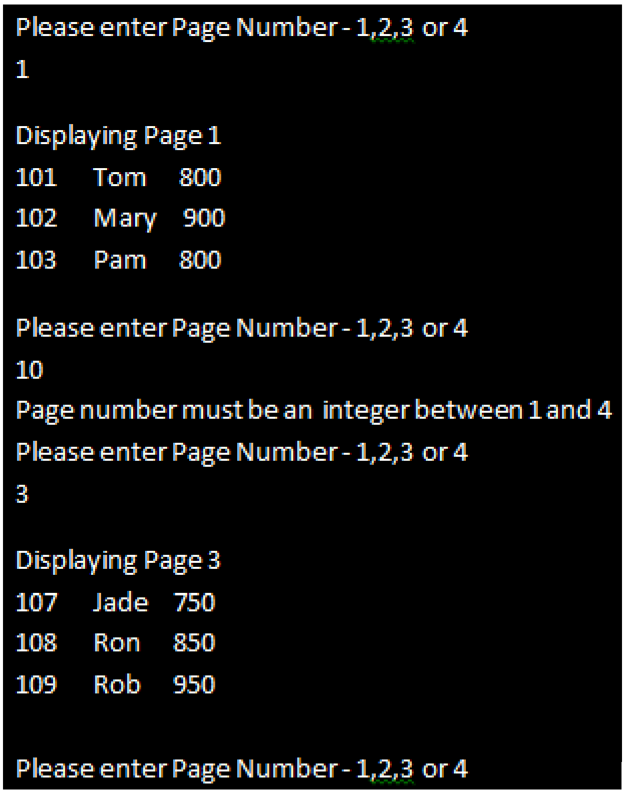
In this video, we will discuss **implementing paging using Skip and Take** operators in LINQ.

We will use the following Student class in this demo. Notice that, there are **11 total Students**. We want to display a maximum of **3 students per page**. So there will be **4 total pages.** The last page, i.e **Page 4** will display the last **2 students**.

```
public class Student
{
    public int StudentID { get; set; }
    public string Name { get; set; }
    public int TotalMarks { get; set; }

    public static List<Student> GetAllStudetns()
    {
        List<Student> listStudents = new List<Student>
        {
            new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },
            new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },
            new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 },
            new Student { StudentID= 104, Name = "John", TotalMarks = 800 },
            new Student { StudentID= 105, Name = "John", TotalMarks = 800 },
            new Student { StudentID= 106, Name = "Brian", TotalMarks = 700 },
            new Student { StudentID= 107, Name = "Jade", TotalMarks = 750 },
            new Student { StudentID= 108, Name = "Ron", TotalMarks = 850 },
            new Student { StudentID= 109, Name = "Rob", TotalMarks = 950 },
            new Student { StudentID= 110, Name = "Alex", TotalMarks = 750 },
            new Student { StudentID= 111, Name = "Susan", TotalMarks = 860 },
        };

        return listStudents;
    }
}
```

Here is what we want to do
1. The program should prompt the user to enter a page number. The Page number must be between 1 and 4.
2. If the user does not enter a valid page number, the program should prompt the user to enter a valid page number.
3. Once a valid page number is entered, the program should display the correct set of Students

For example, **the output of the program** should be as shown below.



```
Please enter Page Number - 1,2,3 or 4
1

Displaying Page 1
101     Tom     800
102     Mary    900
103     Pam     800

Please enter Page Number - 1,2,3 or 4
10
Page number must be an integer between 1 and 4
Please enter Page Number - 1,2,3 or 4
3

Displaying Page 3
107     Jade    750
108     Ron     850
109     Rob     950

Please enter Page Number - 1,2,3 or 4
```

The following console application use **Skip()** and **Take()** operators to achieve this.

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

namespace Demo
{
    class Program
    {
        public static void Main()
        {
            IEnumerable<Student> students = Student.GetAllStudetns();

            do
            {
                Console.WriteLine("Please enter Page Number - 1,2,3 or 4");
                int pageNumber = 0;

                if (int.TryParse(Console.ReadLine(), out pageNumber))
                {
                    if (pageNumber >= 1 && pageNumber <= 4)
                    {
                        int pageSize = 3;
                        IEnumerable<Student> result = students
                                       .Skip((pageNumber - 1) * pageSize).Take(pageSize);

                        Console.WriteLine();
                        Console.WriteLine("Displaying Page " + pageNumber);
                        foreach (Student student in result)
                        {
                            Console.WriteLine(student.StudentID + "\t" +
                                                                        student.Name + "\t" +
student.TotalMarks);
                        }
                        Console.WriteLine();
                    }
                    else
                    {
                        Console.WriteLine("Page number must be an integer between 1 and
4");
                    }
                }
                else
                {
                    Console.WriteLine("Page number must be an integer between 1 and 4");
                }
            } while (1 == 1);
        }
    }
}
```

Please Note: The condition in the while loop puts the program in an infinite loop. To end the program, simply close the console window.

14 – Linq Deferred Execution

In this video we will discuss the concept of **deferred execution**. LINQ queries have two different behaviors of execution
1. Deferred execution
2. Immediate execution

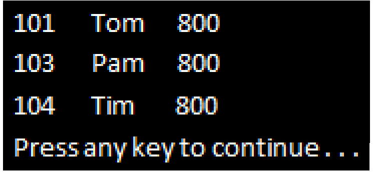
LINQ operators can be broadly classified into 2 categories based on the behaviour of query execution
1. Deferred or Lazy Operators - These query operators use deferred execution.
Examples - select, where, Take, Skip etc
2. Immediate or Greedy Operators - These query operators use immediate execution.
Examples - count, average, min, max, ToList etc

Let us understand these 2 behaviors with examples.

LINQ Deferred Execution Example

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

namespace Demo
{
    public class Student
    {
        public int StudentID { get; set; }
        public string Name { get; set; }
        public int TotalMarks { get; set; }
    }

    class Program
    {
        public static void Main()
        {
            List<Student> listStudents = new List<Student>
            {
                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },
                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },
                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }
            };

            // LINQ Query is only defined here and is not executed at this point
            // If the query is executed at this point, the result should not display Tim
            IEnumerable<Student> result = from student in listStudents
                                          where student.TotalMarks == 800
                                          select student;

            // Add a new student object with TotalMarks = 800 to the source
            listStudents.Add(new Student { StudentID = 104, Name = "Tim", TotalMarks = 
800 });

            // The above query is actually executed when we iterate thru the sequence
            // using the foreach loop. This is proved as Tim is also included in the result
            foreach (Student s in result)
            {
                Console.WriteLine(s.StudentID + "\t" + s.Name + "\t" + s.TotalMarks);
            }
        }
    }
}
```

Output:

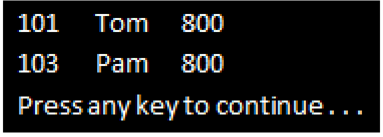


```
101   Tom   800
103   Pam   800
104   Tim   800
Press any key to continue . . .
```

LINQ Immediate Execution Example 1

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

namespace Demo
{
    public class Student
    {
        public int StudentID { get; set; }
        public string Name { get; set; }
        public int TotalMarks { get; set; }
    }

    class Program
    {
        public static void Main()
        {
            List<Student> listStudents = new List<Student>
            {
                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },
                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },
                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }
            };

            // Since we are using ToList() which is a greedy operator
            // the LINQ Query is executed immediately at this point
            IEnumerable<Student> result = (from student in listStudents
                                           where student.TotalMarks == 800
                                           select student).ToList();

            // Adding a new student object with TotalMarks = 800 to the source
            // will have no effect on the result as the query is already executed
            listStudents.Add(new Student { StudentID = 104, Name = "Tim", TotalMarks =
800 });

            // The above query is executed at the point where it is defined.
            // This is proved as Tim is not included in the result
            foreach (Student s in result)
            {
                Console.WriteLine(s.StudentID + "\t" + s.Name + "\t" + s.TotalMarks);
            }
        }
    }
}
```

Output:

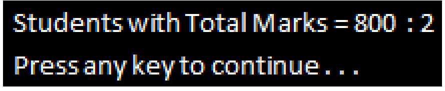


```
101   Tom   800
103   Pam   800
Press any key to continue . . .
```

LINQ Immediate Execution Example 2

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

namespace Demo
{
    public class Student
    {
        public int StudentID { get; set; }
        public string Name { get; set; }
        public int TotalMarks { get; set; }
    }

    class Program
    {
        public static void Main()
        {
            List<Student> listStudents = new List<Student>
            {
                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },
                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },
                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }
            };

            // Since we are using Count() operator, the LINQ Query is executed at this point
            int result = (from student in listStudents
                          where student.TotalMarks == 800
                          select student).Count();

            // Adding a new student object with TotalMarks = 800 to the source
            // will have no effect on the result as the query is already executed
            listStudents.Add(new Student { StudentID = 104, Name = "Tim", TotalMarks = 800 });

            // The above query is executed at the point where it is defined.
            // This is proved as Tim is not included in the count
            Console.WriteLine("Students with Total Marks = 800 : " + result);
        }
    }
}
```

Output:



```
Students with Total Marks = 800  : 2
Press any key to continue . . .
```

15 – Conversion Operators

The following standard LINQ query operators belong to **Conversion Operators** category

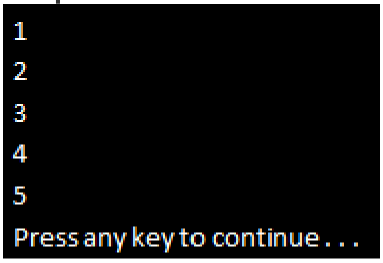
ToList
ToArray
ToDictionary
ToLookup
Cast
OfType
AsEnumerable
AsQueryable

ToList operator extracts all of the items from the source sequence and returns a new **List<T>**. This operator causes the query to be executed immediately. This operator does not use deferred execution.

Example 1: Convert int array to List<int>

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

namespace Demo
{
   class Program
   {
      public static void Main()
      {
         int[] numbers = { 1, 2, 3, 4, 5 };

         List<int> result = numbers.ToList();

         foreach (int i in result)
         {
            Console.WriteLine(i);
         }
      }
   }
}
```

Output:



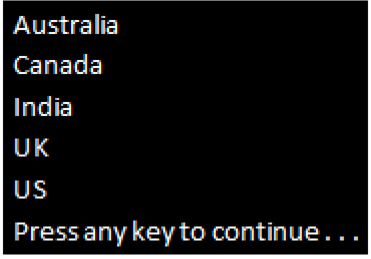
```
1
2
3
4
5
Press any key to continue . . .
```

ToArray operator extracts all of the items from the source sequence and returns a new Array. This operator causes the query to be executed immediately. This operator does not use deferred execution.

Example 2: Convert List<string> to string array. The items in the array should be sorted in ascending order.

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

namespace Demo
{
    class Program
    {
        public static void Main()
        {
            List<string> countries = new List<string>
{ "US", "India", "UK", "Australia", "Canada" };

            string[] result = (from country in countries
                               orderby country ascending
                               select country).ToArray();

            foreach (string str in result)
            {
                Console.WriteLine(str);
            }
        }
    }
}
```

Output:



```
Australia
Canada
India
UK
US
Press any key to continue . . .
```

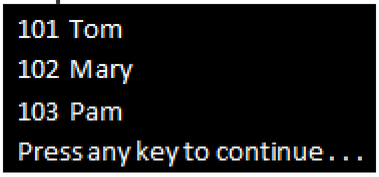
ToDictionary operator extracts all of the items from the source sequence and returns a new Dictionary. This operator causes the query to be executed immediately. This operator does not use deferred execution.

Example 3 : Convert List<Student> to a Dictionary. StudentID should be the key and Name should be the value. In this example, we are using the overloaded of ToDictionary() that takes 2 parameters
a) keySelector - A function to extract a key from each element
b) elementSelector - A function to produce a result element from each element in the sequence

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

```
namespace Demo
{
    public class Student
    {
        public int StudentID { get; set; }
        public string Name { get; set; }
        public int TotalMarks { get; set; }
    }

    class Program
    {
        public static void Main()
        {
            List<Student> listStudents = new List<Student>
            {
                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },
                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },
                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }
            };

            Dictionary<int, string> result = listStudents
                                                                     .ToDictionary(x => x.StudentID, x =>
x.Name);

            foreach (KeyValuePair<int, string> kvp in result)
            {
                Console.WriteLine(kvp.Key + " " + kvp.Value);
            }
        }
    }
}
```

Output:

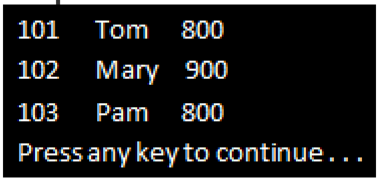


```
101 Tom
102 Mary
103 Pam
Press any key to continue . . .
```

Example 4 : Convert List<Student> to a Dictionary. StudentID should be the key and Student object should be the value. In this example, we are using the overloaded of ToDictionary() that takes 1 parameter
a) keySelector - A function to extract a key from each element

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

namespace Demo
{
    public class Student
    {
        public int StudentID { get; set; }
        public string Name { get; set; }
        public int TotalMarks { get; set; }
    }

    class Program
    {
        public static void Main()
        {
            List<Student> listStudents = new List<Student>
            {
                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },
                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },
                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }
            };

            Dictionary<int, Student> result = listStudents.ToDictionary(x => x.StudentID);

            foreach (KeyValuePair<int, Student> kvp in result)
            {
                Console.WriteLine(kvp.Key + "\t" + kvp.Value.Name + "\t" + 
kvp.Value.TotalMarks);
            }
        }
    }
}
```

Output:



```
101    Tom    800
102    Mary   900
103    Pam    800
Press any key to continue . . .
```

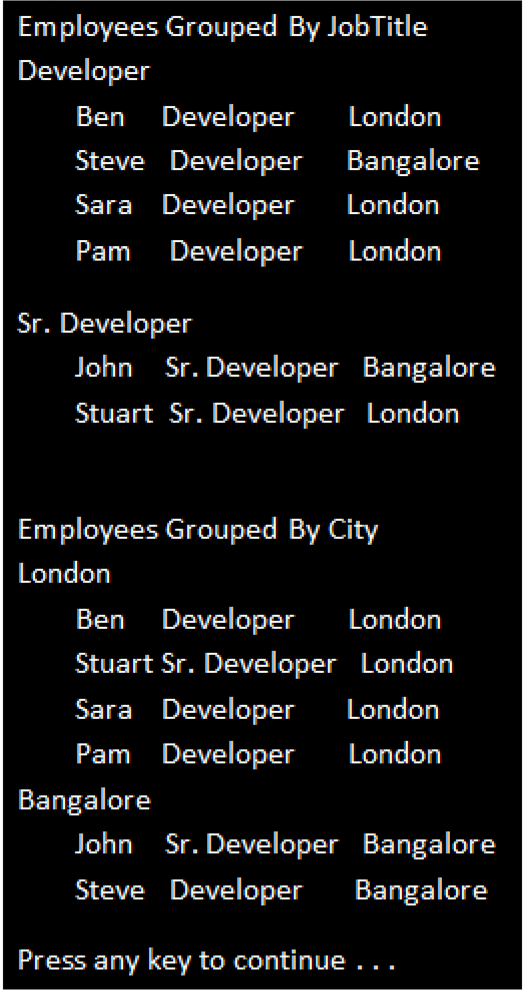
Please Note: Keys in the dictionary must be unique. If two identical keys are created by the keySelector function, the following System.ArgumentException will be thrown at runtime.
Unhandled Exception: System.ArgumentException: An item with the same key has already been added.

ToLookup creates a Lookup. Just like a dictionary, a Lookup is a collection of key/value pairs. A dictionary cannot contain keys with identical values, where as a Lookup can.

Example 5: Create 2 Lookups. First lookup should group Employees by JobTitle, and second lookup should group Employees by City.

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

namespace Demo
{
    public class Employee
    {
        public string Name { get; set; }
        public string JobTitle { get; set; }
        public string City { get; set; }
    }
    class Program
    {
        public static void Main()
        {
            List<Employee> listEmployees = new List<Employee>
            {
                new Employee() { Name = "Ben", JobTitle = "Developer", City = "London" },
                new Employee() { Name = "John", JobTitle = "Sr. Developer", City
= "Bangalore" },
                new Employee() { Name = "Steve", JobTitle = "Developer", City
= "Bangalore" },
                new Employee() { Name = "Stuart", JobTitle = "Sr. Developer", City
= "London" },
                new Employee() { Name = "Sara", JobTitle = "Developer", City = "London" },
                new Employee() { Name = "Pam", JobTitle = "Developer", City = "London" }
            };

            // Group employees by JobTitle
            var employeesByJobTitle = listEmployees.ToLookup(x => x.JobTitle);

            Console.WriteLine("Employees Grouped By JobTitle");
            foreach (var kvp in employeesByJobTitle)
            {
                Console.WriteLine(kvp.Key);
                // Lookup employees by JobTitle
                foreach (var item in employeesByJobTitle[kvp.Key])
                {
                    Console.WriteLine("\t" + item.Name + "\t" + item.JobTitle + "\t" + item.City);
                }
            }

            Console.WriteLine(); Console.WriteLine();

            // Group employees by City
            var employeesByCity = listEmployees.ToLookup(x => x.City);

            Console.WriteLine("Employees Grouped By City");
            foreach (var kvp in employeesByCity)
            {
                Console.WriteLine(kvp.Key);
                // Lookup employees by City
                foreach (var item in employeesByCity[kvp.Key])
                {
                    Console.WriteLine("\t" + item.Name + "\t" + item.JobTitle + "\t" + item.City);
                }
            }
        }
    }
}
```

Output:



```
Employees Grouped By JobTitle
Developer
        Ben     Developer       London
        Steve   Developer       Bangalore
        Sara    Developer       London
        Pam     Developer       London

Sr. Developer
        John    Sr. Developer   Bangalore
        Stuart  Sr. Developer   London


Employees Grouped By City
London
        Ben     Developer       London
        Stuart Sr. Developer    London
        Sara    Developer       London
        Pam     Developer       London
Bangalore
        John    Sr. Developer   Bangalore
        Steve   Developer       Bangalore

Press any key to continue . . .
```

16 – Cast and OfType Operators

The following standard LINQ query operators belong to **Conversion Operators** category

ToList
ToArray
ToDictionary
ToLookup
Cast
OfType
AsEnumerable
AsQueryable

We discussed the following operators in Part 15

ToList
ToArray
ToDictionary
ToLookup

In this video we will discuss

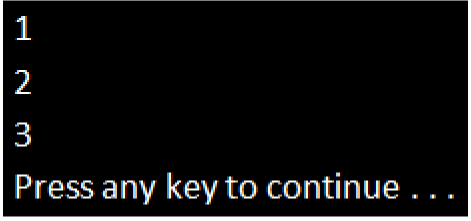
1. Cast and OfType operators
2. Difference between Cast and OfType operators
3. When to use one over the other

Cast operator attempts to convert all of the items within an existing collection to another type and return them in a new collection. If an item fails conversion an exception will be thrown. This method uses deferred execution.

Example :

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

namespace Demo
{
    class Program
    {
        public static void Main()
        {
            ArrayList list = new ArrayList();
            list.Add(1);
            list.Add(2);
            list.Add(3);

            // The following item causes an exception
            // list.Add("ABC");

            IEnumerable<int> result = list.Cast<int>();

            foreach (int i in result)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

Output :



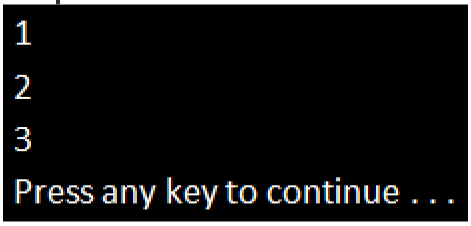
```
1
2
3
Press any key to continue . . .
```

OfType operator will return only elements of the specified type. The other type elements are simply ignored and excluded from the result set.

Example : In the example below, items **"4"** and **"ABC"** will be ignored from the result set. No exception will be thrown.

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

namespace Demo
{
    class Program
    {
        public static void Main()
        {
            ArrayList list = new ArrayList();
            list.Add(1);
            list.Add(2);
            list.Add(3);
            list.Add("4");
            list.Add("ABC");

            IEnumerable<int> result = list.OfType<int>();

            foreach (int i in result)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

Output :



```
1
2
3
Press any key to continue . . .
```

What is the difference between Cast and OfType operators
OfType operator returns only the elements of the specified type and the rest of the items in the collection will be ignored and excluded from the result.

Cast operator will try to cast all the elements in the collection into the specified type. If some of the items fail conversion, InvalidCastException will be thrown.

When to use Cast over OfType and vice versa?
We would generally use Cast when the following 2 conditions are met
1. We want to cast all the items in the collection &
2. We know for sure the collection contains only elements of the specified type

If we want to filter the elements and return only the ones of the specified type, then we would use OfType.

17 – AsEnumerable and AsQueryable

In this video we will discuss the use of **AsEnumerable** and **AsQueryable** operators in LINQ. Both of these operators belong to **Conversion Operators** category.

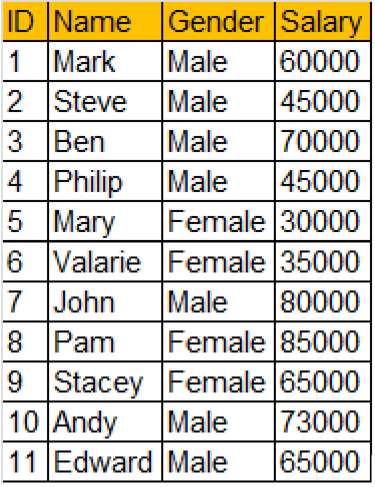
AsQueryable operator: There are 2 overloaded versions of this method.

One overloaded version converts System.Collections.IEnumerable to System.Linq.IQueryable

The other overloaded version converts a generic System.Collections.Generic.IEnumerable<T> to a generic System.Linq.IQueryable<T>

The main use of **AsQueryable** operator is unit testing to mock a **queryable** data source using an in-memory data source. We will discuss this operator in detail with examples in unit testing video series.

AsEnumerable operator: Let us understand the use of this operator with an example. We will be using the following **Employees** table in this demo.



ID	Name	Gender	Salary
1	Mark	Male	60000
2	Steve	Male	45000
3	Ben	Male	70000
4	Philip	Male	45000
5	Mary	Female	30000
6	Valarie	Female	35000
7	John	Male	80000
8	Pam	Female	85000
9	Stacey	Female	65000
10	Andy	Male	73000
11	Edward	Male	65000

Step 1: Execute the following SQL Script to create and populate **Employees** Table

```
Create Table Employees
(
    ID int primary key identity,
    Name nvarchar(50),
    Gender nvarchar(50),
    Salary int
)
GO

Insert into Employees Values('Mark','Male','60000')
Insert into Employees Values('Steve','Male','45000')
Insert into Employees Values('Ben','Male','70000')
Insert into Employees Values('Philip','Male','45000')
Insert into Employees Values('Mary','Female','30000')
Insert into Employees Values('Valarie','Female','35000')
Insert into Employees Values('John','Male','80000')
Insert into Employees Values('Pam','Female','85000')
Insert into Employees Values('Stacey','Female','65000')
Insert into Employees Values('Andy','Male','73000')
Insert into Employees Values('Edward','Male','65000')
```

```
GO
```

Step 2: Create a new Console Application. Name it **Demo**.

Step 3: Right click on the Demo project in Solution Explorer and Add a new LINQ to SQL Classes. Name it **EmployeeDB.dbml**.

Step 4: Click on **View** menu, and select **"Server Explorer".** Expand **Data Connections** and then Drag and Drop **Employees** table onto **EmployeeDB.dbml** designer surface.

Step 5: Copy and paste the following code in Program.cs file. The linq query in this sample, retrieves the **TOP 5 Male Employees By Salary**.

```
using System;
using System.Linq;
namespace Demo
{
    class Program
    {
        public static void Main()
        {
            EmployeeDBDataContext dbContext = new EmployeeDBDataContext();
            // TOP 5 Male Employees By Salary
            var result = dbContext.Employees.Where(x => x.Gender == "Male")
                                    .OrderByDescending(x => x.Salary).Take(5);

            Console.WriteLine("Top 5 Salaried Male Employees");
            foreach (Employee e in result)
            {
                Console.WriteLine(e.Name + "\t" + e.Gender + "\t" + e.Salary);
            }
        }
    }
}
```

Step 6: Now open **SQL Profiler** and run a new trace and then run the console application.

Step 7: Notice that the following SQL Query is executed against the database.

```
exec sp_executesql N'SELECT TOP (5) [t0].[ID], [t0].[Name], [t0].[Gender], [t0].[Salary]
FROM [dbo].[Employees] AS [t0]
WHERE [t0].[Gender] = @p0
ORDER BY [t0].[Salary] DESC',N'@p0 nvarchar(4000)',@p0=N'Male'
```

Step 8: Change the LINQ query in the console application

FROM

```
var result = dbContext.Employees.Where(x => x.Gender == "Male")
                                          .OrderByDescending(x => x.Salary).Take(5);
```

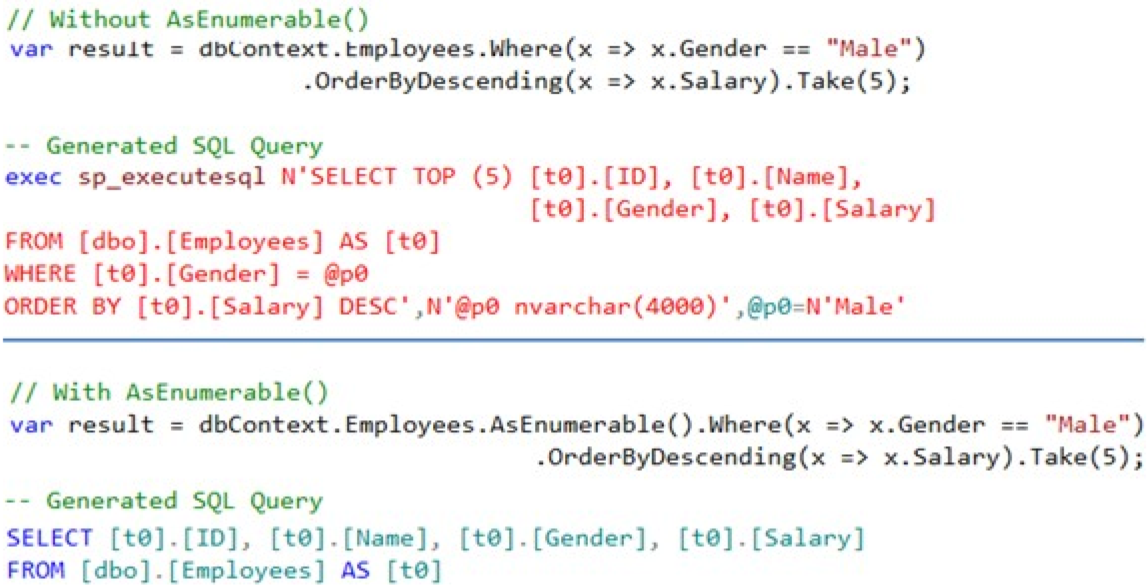
TO

```
var result = dbContext.Employees.AsEnumerable()
                                          .Where(x => x.Gender == "Male")
                                          .OrderByDescending(x => x.Salary).Take(5);
```

Step 9: Run the console application and notice the query generated in SQL Profiler.

```
SELECT [t0].[ID], [t0].[Name], [t0].[Gender], [t0].[Salary]
FROM [dbo].[Employees] AS [t0]
```

Summary:



```
// Without AsEnumerable()
var result = dbContext.Employees.Where(x => x.Gender == "Male")
                        .OrderByDescending(x => x.Salary).Take(5);

-- Generated SQL Query
exec sp_executesql N'SELECT TOP (5) [t0].[ID], [t0].[Name],
                                    [t0].[Gender], [t0].[Salary]
FROM [dbo].[Employees] AS [t0]
WHERE [t0].[Gender] = @p0
ORDER BY [t0].[Salary] DESC',N'@p0 nvarchar(4000)',@p0=N'Male'
```

```
// With AsEnumerable()
var result = dbContext.Employees.AsEnumerable().Where(x => x.Gender == "Male")
                                    .OrderByDescending(x => x.Salary).Take(5);

-- Generated SQL Query
SELECT [t0].[ID], [t0].[Name], [t0].[Gender], [t0].[Salary]
FROM [dbo].[Employees] AS [t0]
```

AsEnumerable operator breaks the query into 2 parts

1. The "inside part" that is the query before AsEnumerable operator is executed as Linq-to-SQL

2. The "ouside part" that is the query after AsEnumerable operator is executed as Linq-to-Objects

So in this example the following SQL Query is executed against SQL Server, all the data is brought into the console application and then the WHERE, ORDERBY & TOP operators are applied on the client-side

```
SELECT [t0].[ID], [t0].[Name], [t0].[Gender], [t0].[Salary]
FROM [dbo].[Employees] AS [t0]
```

So in short, use **AsEnumerable** operator to move query processing to the client side.

art 18 – Group By (Grouping Operators)

GroupBy operator belong to **Grouping Operators** category. This operator takes a flat sequence of items, organize that sequence into groups (**IGrouping<K,V>**) based on a specific key and return groups of sequences.

In short, **GroupBy** creates and returns a sequence of **IGrouping<K,V>**

Let us understand GroupBy with examples.

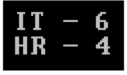
We will use the following **Employee** class in this demo

```
public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Gender { get; set; }
    public string Department { get; set; }
    public int Salary { get; set; }

    public static List<Employee> GetAllEmployees()
    {
        return new List<Employee>()
        {
            new Employee { ID = 1, Name = "Mark", Gender = "Male",
                                Department = "IT", Salary = 45000 },
            new Employee { ID = 2, Name = "Steve", Gender = "Male",
                                Department = "HR", Salary = 55000 },
            new Employee { ID = 3, Name = "Ben", Gender = "Male",
                                Department = "IT", Salary = 65000 },
            new Employee { ID = 4, Name = "Philip", Gender = "Male",
                                Department = "IT", Salary = 55000 },
            new Employee { ID = 5, Name = "Mary", Gender = "Female",
                                Department = "HR", Salary = 48000 },
            new Employee { ID = 6, Name = "Valarie", Gender = "Female",
                                Department = "HR", Salary = 70000 },
            new Employee { ID = 7, Name = "John", Gender = "Male",
                                Department = "IT", Salary = 64000 },
            new Employee { ID = 8, Name = "Pam", Gender = "Female",
                                Department = "IT", Salary = 54000 },
            new Employee { ID = 9, Name = "Stacey", Gender = "Female",
                                Department = "HR", Salary = 84000 },
            new Employee { ID = 10, Name = "Andy", Gender = "Male",
                                Department = "IT", Salary = 36000 }
        };
    }
}
```

Example 1: Get Employee Count By Department

```
var employeeGroup = from employee in Employee.GetAllEmployees()
                    group employee by employee.Department;

foreach (var group in employeeGroup)
{
    Console.WriteLine("{0} - {1}", group.Key, group.Count());
}
```

Output:

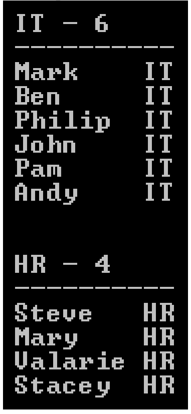


```
IT - 6
HR - 4
```

Example 2: Get Employee Count By Department and also each employee and department name

```
var employeeGroup = from employee in Employee.GetAllEmployees()
                                        group employee by employee.Department;

foreach (var group in employeeGroup)
{
    Console.WriteLine("{0} - {1}", group.Key, group.Count());
    Console.WriteLine("----------");
    foreach (var employee in group)
    {
        Console.WriteLine(employee.Name + "\t" + employee.Department);
    }
    Console.WriteLine(); Console.WriteLine();
}
```

Output:

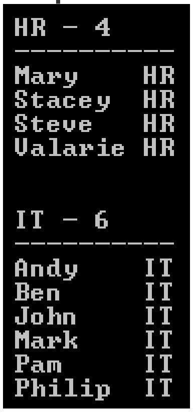


```
IT - 6
----------
Mark    IT
Ben     IT
Philip  IT
John    IT
Pam     IT
Andy    IT


HR - 4
----------
Steve   HR
Mary    HR
Valarie HR
Stacey  HR
```

Example 3: Get Employee Count By Department and also each employee and department name. Data should be sorted first by Department in ascending order and then by Employee Name in ascending order.

```
var employeeGroup = from employee in Employee.GetAllEmployees()
                                        group employee by employee.Department into eGroup
                                        orderby eGroup.Key
                                        select new
                                        {
                                             Key = eGroup.Key,
                                             Employees = eGroup.OrderBy(x => x.Name)
                                        };

foreach (var group in employeeGroup)
{
    Console.WriteLine("{0} - {1}", group.Key, group.Employees.Count());
    Console.WriteLine("----------");
    foreach (var employee in group.Employees)
    {
        Console.WriteLine(employee.Name + "\t" + employee.Department);
    }
    Console.WriteLine(); Console.WriteLine();
}
```

Output:



```
HR - 4
----------
Mary     HR
Stacey   HR
Steve    HR
Valarie  HR


IT - 6
----------
Andy     IT
Ben      IT
John     IT
Mark     IT
Pam      IT
Philip   IT
```

Part 19 – Group by Multiple Keys

In this video, we will discuss **Grouping by multiple keys**. In LINQ, an anonymous type is usually used when we want to group by multiple keys.

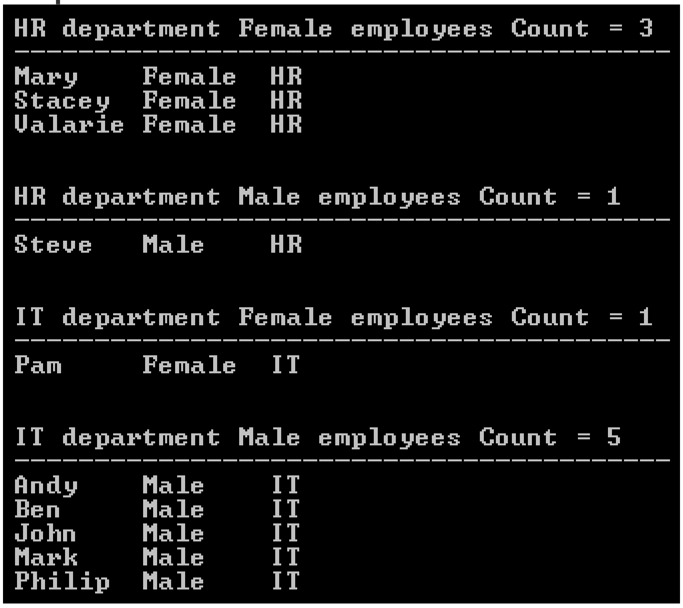
Let us understand this with an example. We will be using the following **Employee** class in this demo. This is the same class used in Part 18. Please watch Part 18 before proceeding.

```
public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public string Gender { get; set; }
    public string Department { get; set; }

    public static List<Employee> GetAllEmployees()
    {
        return new List<Employee>()
        {
            new Employee { ID = 1, Name = "Mark", Gender = "Male",
                                         Department = "IT" },
            new Employee { ID = 2, Name = "Steve", Gender = "Male",
                                         Department = "HR" },
            new Employee { ID = 3, Name = "Ben", Gender = "Male",
                                         Department = "IT" },
            new Employee { ID = 4, Name = "Philip", Gender = "Male",
                                         Department = "IT" },
            new Employee { ID = 5, Name = "Mary", Gender = "Female",
                                         Department = "HR" },
            new Employee { ID = 6, Name = "Valarie", Gender = "Female",
                                         Department = "HR" },
            new Employee { ID = 7, Name = "John", Gender = "Male",
                                         Department = "IT" },
            new Employee { ID = 8, Name = "Pam", Gender = "Female",
                                         Department = "IT" },
            new Employee { ID = 9, Name = "Stacey", Gender = "Female",
                                         Department = "HR" },
            new Employee { ID = 10, Name = "Andy", Gender = "Male",
                                         Department = "IT" },
        };
    }
}
```

Example 1: Group employees by **Department** and then by **Gender**. The employee groups should be sorted first by **Department** and then by **Gender** in ascending order. Also, employees within each group must be sorted in ascending order by Name.

```
var employeeGroups = Employee.GetAllEmployees()
                                        .GroupBy(x => new { x.Department, x.Gender })
                                        .OrderBy(g => g.Key.Department).ThenBy(g =>
g.Key.Gender)
                                        .Select(g => new
                                        {
                                            Dept = g.Key.Department,
                                            Gender = g.Key.Gender,
                                            Employees = g.OrderBy(x => x.Name)
                                        });

foreach(var group in employeeGroups)
{
    Console.WriteLine("{0} department {1} employees count = {2}",
        group.Dept, group.Gender, group.Employees.Count());
    Console.WriteLine("--------------------------------------------");
    foreach (var employee in group.Employees)
    {
        Console.WriteLine(employee.Name + "\t" + employee.Gender
            + "\t" + employee.Department);
    }
    Console.WriteLine(); Console.WriteLine();
}
```

Output:



```
HR department Female employees Count = 3
------------------------------------------
Mary    Female  HR
Stacey  Female  HR
Valarie Female  HR


HR department Male employees Count = 1
------------------------------------------
Steve   Male    HR


IT department Female employees Count = 1
------------------------------------------
Pam     Female  IT


IT department Male employees Count = 5
------------------------------------------
Andy    Male    IT
Ben     Male    IT
John    Male    IT
Mark    Male    IT
Philip  Male    IT
```

Example 2: Rewrite Example 1 using **SQL like syntax**

```
var employeeGroups = from employee in Employee.GetAllEmployees()
                                        group employee by new
                                        {
                                             employee.Department,
                                             employee.Gender
                                        } into eGroup
                                        orderby eGroup.Key.Department ascending,
                                                      eGroup.Key.Gender ascending
                                        select new
                                        {
                                             Dept = eGroup.Key.Department,
                                             Gender = eGroup.Key.Gender,
                                             Employees = eGroup.OrderBy(x => x.Name)
                                        };
```

Part 20 – Element Operators

The following standard query operators belong to Element Operators category
First / FirstOrDefault
Last / LastOrDefault
ElementAt / ElementAtOrDefault
Single / SingleOrDefault
DefaultIfEmpty

Element Operators retrieve a single element from a sequence using the element index or based on a condition. All of these methods have a corresponding overloaded version that accepts a predicate.

First : There are 2 overloaded versions of this method. The first overloaded version that does not have any parameters simply returns the first element of a sequence.

Example 1: Returns the first element from the sequence

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int result = numbers.First();
Console.WriteLine("Result = " + result);
```

Output:

```
Result = 1
```

If the sequence does not contain any elements, then First() method throws an InvalidOperationException.

Example 2: Throws InvalidOperationException.

```
int[] numbers = { };
int result = numbers.First();
Console.WriteLine("Result = " + result);
```

Output:

```
Unhandled Exception: System.InvalidOperationException: Sequence contains no
elements
```

The second overloaded version is used to find the first element in a sequence based on a condition. If the sequence does not contain any elements or if no element in the sequence satisfies the condition then an InvalidOperationException is thrown.

Example 3: Returns the first even number from the sequence

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int result = numbers.First(x => x % 2 == 0);
Console.WriteLine("Result = " + result);
```

Output:

```
Result = 2
```

Example 4: Throws InvalidOperationException, as no element in the sequence satisfies the condition specified by the predicate.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int result = numbers.First(x => x % 2 == 100);
Console.WriteLine("Result = " + result);
```

Output:

```
Unhandled Exception: System.InvalidOperationException: Sequence contains no matching element
```

FirstOrDefault : This is very similar to First, except that this method does not throw an exception when there are no elements in the sequence or when no element satisfies the condition specified by the predicate. Instead, a default value of the type that is expected is returned. For reference types the default is NULL and for value types the default depends on the actual type expected.

Example 5: Returns ZERO. No element in the sequence satisfies the condition, so the default value (ZERO) for int is returned.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int result = numbers.FirstOrDefault(x => x % 2 == 100);
Console.WriteLine("Result = " + result);
```

Last : Very similar to First, except it returns the last element of the sequence.

LastOrDefault : Very similar to FirstOrDefault, except it returns the last element of the sequence.

ElementAt : Returns an element at a specified index. If the sequence is empty or if the provided index value is out of range, then an ArgumentOutOfRangeException is thrown.

Example 6: Returns element from the sequence that is at index position 1.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
int result = numbers.ElementAt(1);
Console.WriteLine("Result = " + result);
```

Output:

```
Result = 2
```

Example 7: Throws ArgumentOutOfRangeException

```
int[] numbers = { };
int result = numbers.ElementAt(0);
Console.WriteLine("Result = " + result);
```

Output:

```
Unhandled Exception: System.ArgumentOutOfRangeException: Index was out of range. Must be non-negative and less than the size of the collection.
```

ElementAtOrDefault : Similar to ElementAt except that this method does not throw an exception, if the sequence is empty or if the provided index value is out of range. Instead, a default value of the type that is expected is returned.

Single : There are 2 overloaded versions of this method. The first overloaded version that does not have any parameters returns the only element of the sequence.

Example 8: Returns the only element (1) of the sequence.

```
int[] numbers = { 1 };
int result = numbers.Single();
Console.WriteLine("Result = " + result);
```

Output:

```
Result = 1
```

Single() method throws an exception if the sequence is empty or has more than one element.

Example 9: Throws InvalidOperationException as the sequence contains more than ONE element.

```
int[] numbers = { 1, 2 };
int result = numbers.Single();
Console.WriteLine("Result = " + result);
```

Output:

```
Unhandled Exception: System.InvalidOperationException: Sequence contains more than one element
```

The second overloaded version of the Single() method is used to find the only element in a sequence that satisfies a given condition. An exception will be thrown if any of the following is true
a) If the sequence does not contain any elements OR
b) If no element in the sequence satisfies the condition OR
c) If more than one element in the sequence satisfies the condition

Example 10: Throws InvalidOperationException as more than one element in the sequence satisfies the condition

```
int[] numbers = { 1, 2, 4 };
int result = numbers.Single(x => x % 2 == 0);
Console.WriteLine("Result = " + result);
```

Output:

```
Unhandled Exception: System.InvalidOperationException: Sequence contains more than one matching element
```

SingleOrDefault : Very similar to Single(), except this method does not throw an exception when the sequence is empty or when no element in the sequence satisfies the given condition. Just like Single(), this method will still throw an exception, if more than one element in the sequence satisfies the given condition.

Example 11: Throws InvalidOperationException as more than one element in the sequence satisfies the given condition

```
int[] numbers = { 1, 2, 4 };
int result = numbers.SingleOrDefault(x => x % 2 == 0);
Console.WriteLine("Result = " + result);
```

Output:

```
Unhandled Exception: System.InvalidOperationException: Sequence contains more than one matching element
```

DefaultIfEmpty : If the sequence on which this method is called is not empty, then the values of the original sequence are returned.

Example 12 : Returns a copy of the original sequence

```
int[] numbers = { 1, 2, 3 };
IEnumerable<int> result = numbers.DefaultIfEmpty();
```

```
foreach (int i in result)
{
    Console.WriteLine(i);
}
```

Output:

```
1
2
3
```

If the sequence is empty, then **DefaultIfEmpty**() returns a sequence with the default value of the expected type.

Value type	Default value
bool	`false`
byte	0
char	'\0'
decimal	0.0M
double	0.0D
enum	The value produced by the expression (E)0, where E is the enum identifier.
float	0.0F
int	0

Value type	Default value
long	0L
sbyte	0
short	0
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to `null`.
uint	0
ulong	0
ushort	0

The `null` keyword is a literal that represents a null reference, one that does not refer to any object. `null` is the default value of reference-type variables.

Example 13 : Since the sequence is empty, a sequence containing the default value (ZERO) of int is returned.

```
int[] numbers = { };
IEnumerable<int> result = numbers.DefaultIfEmpty();
foreach (int i in result)
```

```
{
    Console.WriteLine(i);
}
```

Output:

```
0
```

The other overloaded version with a parameter allows us to specify a default value. If this method is called on a sequence that is not empty, then the values of the original sequence are returned. If the sequence is empty, then this method returns a sequence with the specified defualt value.

Example 14 : Since the sequence is empty, a sequence containing the specified default value (10) is returned.

```
int[] numbers = { };
IEnumerable<int> result = numbers.DefaultIfEmpty(10);
foreach (int i in result)
{
    Console.WriteLine(i);
}
```

Output:

```
10
```

Part 21 – Group Join

The following are the different types of joins in LINQ
Group Join - We will discuss in this video
Inner Join - Discussed in Part 22
Left Outer Join
Cross Join

In this video, we will discuss **Group Join**. Group Join produces hierarchical data structures. Each element from the first collection is paired with a set of correlated elements from the second collection.

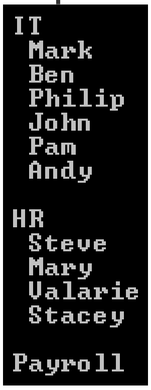
Let us understand **Group Join** with an **example.** Consider the following **Department** and **Employee** classes. A Department may have ZERO or MORE employees.

```
public class Department
{
    public int ID { get; set; }
    public string Name { get; set; }

    public static List<Department> GetAllDepartments()
    {
        return new List<Department>()
        {
            new Department { ID = 1, Name = "IT"},
            new Department { ID = 2, Name = "HR"},
            new Department { ID = 3, Name = "Payroll"},
        };
    }
}

public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int DepartmentID { get; set; }

    public static List<Employee> GetAllEmployees()
    {
        return new List<Employee>()
        {
            new Employee { ID = 1, Name = "Mark", DepartmentID = 1 },
            new Employee { ID = 2, Name = "Steve", DepartmentID = 2 },
            new Employee { ID = 3, Name = "Ben", DepartmentID = 1 },
            new Employee { ID = 4, Name = "Philip", DepartmentID = 1 },
            new Employee { ID = 5, Name = "Mary", DepartmentID = 2 },
            new Employee { ID = 6, Name = "Valarie", DepartmentID = 2 },
            new Employee { ID = 7, Name = "John", DepartmentID = 1 },
            new Employee { ID = 8, Name = "Pam", DepartmentID = 1 },
            new Employee { ID = 9, Name = "Stacey", DepartmentID = 2 },
            new Employee { ID = 10, Name = "Andy", DepartmentID = 1}
        };
    }
}
```

Example 1: Group **employees** by **Department**.

```
var employeesByDepartment = Department.GetAllDepartments()

.GroupJoin(Employee.GetAllEmployees(),
                                    d => d.ID,
                                    e => e.DepartmentID,
                                    (department, employees) => new
                                    {
                                        Department = department,
                                        Employees = employees
                                    });

foreach (var department in employeesByDepartment)
{
    Console.WriteLine(department.Department.Name);
    foreach (var employee in department.Employees)
    {
        Console.WriteLine(" " + employee.Name);
    }
    Console.WriteLine();
}
```

Output:



```
IT
 Mark
 Ben
 Philip
 John
 Pam
 Andy

HR
 Steve
 Mary
 Valarie
 Stacey

Payroll
```

Example 2: Rewrite **Example 1** using SQL like syntax.

```
var employeesByDepartment = from d in Department.GetAllDepartments()
                            join e in Employee.GetAllEmployees()
                            on d.ID equals e.DepartmentID into eGroup
                            select new
                            {
                               Department = d,
                               Employees = eGroup
                            };
```

Please note: Group Join uses the **join** operator and the **into** keyword to group the results of the join.

Part 22 – Inner Join

The following are the different types of joins in LINQ
Group Join - Discussed in Part 21
Inner Join - We will discuss in this video
Left Outer Join - Later Video
Cross Join - Later Video

In this video we will discuss implementing **INNER JOIN** in **LINQ**. If you have 2 collections, and when you perform an inner join, then only the matching elements between the 2 collections are included in the result set. Non - Matching elements are excluded from the result set.

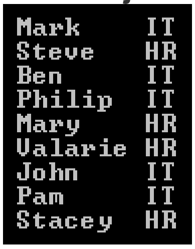
Let us understand Inner Join with an example. Consider the following **Department** and **Employee** classes. Notice that, **Employee** Andy does not have a department assigned. An inner join will not include his record in the result set.

```
public class Department
{
   public int ID { get; set; }
   public string Name { get; set; }

   public static List<Department> GetAllDepartments()
   {
      return new List<Department>()
      {
         new Department { ID = 1, Name = "IT"},
         new Department { ID = 2, Name = "HR"},
         new Department { ID = 3, Name = "Payroll"},
      };
   }
}

public class Employee
{
   public int ID { get; set; }
   public string Name { get; set; }
   public int DepartmentID { get; set; }

   public static List<Employee> GetAllEmployees()
   {
      return new List<Employee>()
      {
         new Employee { ID = 1, Name = "Mark", DepartmentID = 1 },
         new Employee { ID = 2, Name = "Steve", DepartmentID = 2 },
         new Employee { ID = 3, Name = "Ben", DepartmentID = 1 },
         new Employee { ID = 4, Name = "Philip", DepartmentID = 1 },
         new Employee { ID = 5, Name = "Mary", DepartmentID = 2 },
         new Employee { ID = 6, Name = "Valarie", DepartmentID = 2 },
         new Employee { ID = 7, Name = "John", DepartmentID = 1 },
         new Employee { ID = 8, Name = "Pam", DepartmentID = 1 },
         new Employee { ID = 9, Name = "Stacey", DepartmentID = 2 },
         new Employee { ID = 10, Name = "Andy"}
      };
   }
}
```

Example 1 : Join the **Employees** and **Department** collections and print all the Employees and their respective department names.

```
var result = Employee.GetAllEmployees().Join(Department.GetAllDepartments(),
                                        e => e.DepartmentID,
                                        d => d.ID, (employee, department) => new
                                        {
                                            EmployeeName = employee.Name,
                                            DepartmentName = department.Name
                                        });
foreach (var employee in result)
{
    Console.WriteLine(employee.EmployeeName + "\t" + employee.DepartmentName);
}
```

Output: Notice that, in the output we don't have **Andy** record. This is because, Andy does not have a matching department in Department collection. So this is effectively an **inner join**.



```
Mark     IT
Steve    HR
Ben      IT
Philip   IT
Mary     HR
Valarie  HR
John     IT
Pam      IT
Stacey   HR
```

Example 2 : Rewrite Example 1 using SQL like syntax.

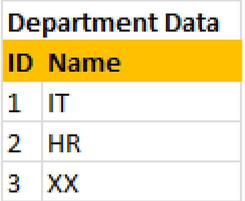
```
var result = from e in Employee.GetAllEmployees()
                    join d in Department.GetAllDepartments()
                    on e.DepartmentID equals d.ID
                    select new
                    {
                        EmployeeName = e.Name,
                        DepartmentName = d.Name
                    };

foreach (var employee in result)
{
    Console.WriteLine(employee.EmployeeName + "\t" + employee.DepartmentName);
}
```

Part 23 – Difference Between Group Join and Inner Join

In this video, we will discuss the **difference between Group Join and Inner Join in LINQ** with examples. We will be using the following Department and Employee classes in this video.

```
public class Department
{
    public int ID { get; set; }
    public string Name { get; set; }

    public static List<Department> GetAllDepartments()
    {
        return new List<Department>()
        {
            new Department { ID = 1, Name = "IT"},
            new Department { ID = 2, Name = "HR"},
            new Department { ID = 3, Name = "XX"},
        };
    }
}

public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int DepartmentID { get; set; }

    public static List<Employee> GetAllEmployees()
    {
        return new List<Employee>()
        {
            new Employee { ID = 1, Name = "Mark", DepartmentID = 1 },
            new Employee { ID = 2, Name = "Steve", DepartmentID = 2 },
            new Employee { ID = 3, Name = "Ben", DepartmentID = 1 },
            new Employee { ID = 4, Name = "Philip", DepartmentID = 1 },
            new Employee { ID = 5, Name = "Mary", DepartmentID = 2 }
        };
    }
}
```

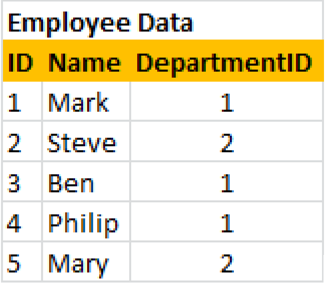
Department data returned by **GetAllDepartments()** method is shown below



Department Data

ID	Name
1	IT
2	HR
3	XX

Employee data returned by **GetAllEmployees()** method is shown below



Employee Data		
ID	Name	DepartmentID
1	Mark	1
2	Steve	2
3	Ben	1
4	Philip	1
5	Mary	2

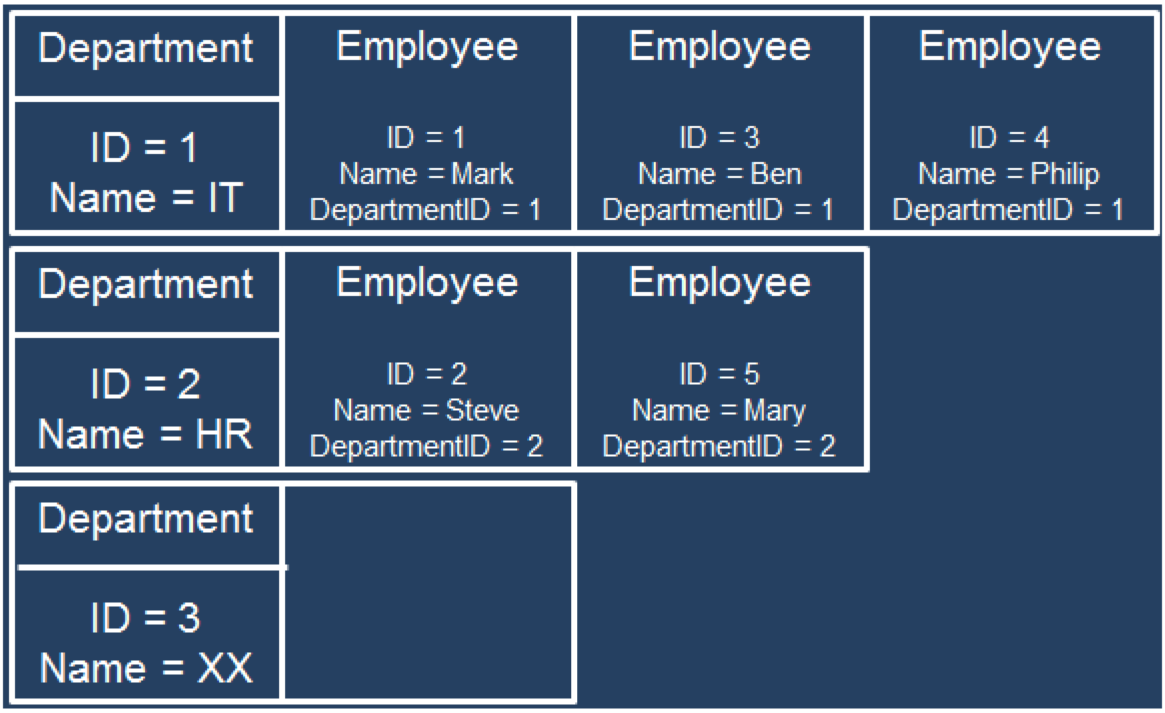
The following query performs a GroupJoin on the 2 lists

```
var result = from d in Department.GetAllDepartments()
             join e in Employee.GetAllEmployees()
             on d.ID equals e.DepartmentID into eGroup
             select new
             {
                Department = d,
                Employees = eGroup
             };
```

Notice that we are using the **join** operator and the into keyword to group the results of the join. To perform group join using extension method syntax, we use **GroupJoin()** Extension method as shown below.

```
var result = Department.GetAllDepartments()
                          .GroupJoin(Employee.GetAllEmployees(),
                           d => d.ID,
                           e => e.DepartmentID,
                           (department, employees) => new
                           {
                               Department = department,
                               Employees = employees
                           });
```

The above 2 queries **groups employees by department** and would produce the following groups.



To print the **Department** and **Employee** Names we use 2 foreach loops as shown below.

```
foreach (var department in result)
{
    Console.WriteLine(department.Department.Name);
    foreach (var employee in department.Employees)
    {
        Console.WriteLine(" " + employee.Name);
    }
    Console.WriteLine();
}
```

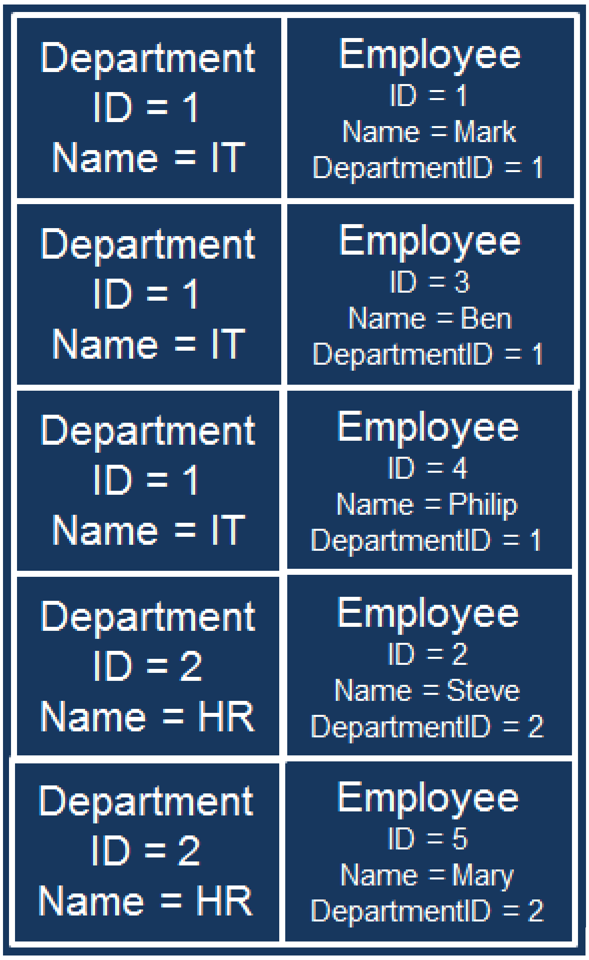
The following query performs an **Inner Join** on the 2 lists

```
var result = from e in Employee.GetAllEmployees()
                    join d in Department.GetAllDepartments()
                    on e.DepartmentID equals d.ID
                    select new { e, d };
```

To perform an **inner join** using extension method syntax, we use **Join()** Extension method as shown below.

```
var result = Employee.GetAllEmployees()
                                    .Join(Department.GetAllDepartments(),
                                     e => e.DepartmentID,
                                     d => d.ID, (employee, department) => new
                                     {
                                          e = employee,
                                          d = department
                                     });
```

The above 2 queries would produce a **flat result set** as shown below



Department
ID = 1
Name = IT
	Employee
	ID = 1
	Name = Mark
	DepartmentID = 1

Department
ID = 1
Name = IT
	Employee
	ID = 3
	Name = Ben
	DepartmentID = 1

Department
ID = 1
Name = IT
	Employee
	ID = 4
	Name = Philip
	DepartmentID = 1

Department
ID = 2
Name = HR
	Employee
	ID = 2
	Name = Steve
	DepartmentID = 2

Department
ID = 2
Name = HR
	Employee
	ID = 5
	Name = Mary
	DepartmentID = 2

To print the **Department** and **Employee** Names we use just 1 foreach loop as shown below.

```
foreach (var employee in result)
{
   Console.WriteLine(employee.e.Name + "\t" + employee.d.Name);

}
```

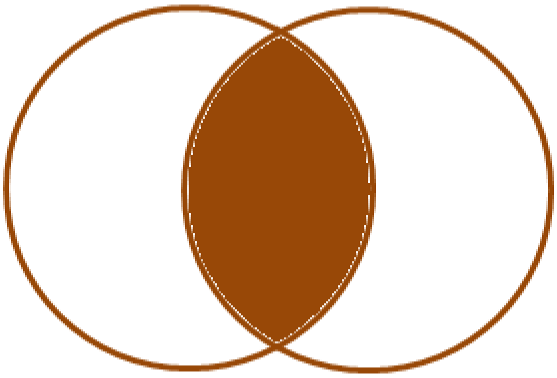
In short, **Join** is similar to **INNER JOIN** in SQL and **GroupJoin** is similar to **OUTER JOIN** in SQL

Part 24 – Left Outter Join

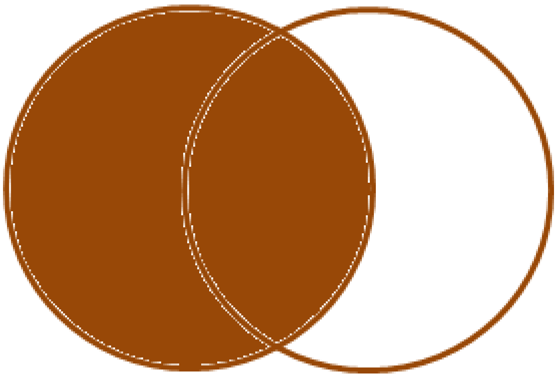
The following are the different types of joins in LINQ
Group Join - Discussed in Part 21
Inner Join - Discussed in Part 22
Left Outer Join - We will discuss in this video
Cross Join - Later Video

In this video we will discuss implementing **LEFT OUTER JOIN in LINQ**.

With **INNER JOIN** only the matching elements are included in the result set. Non-matching elements are excluded from the result set.



With **LEFT OUTER JOIN** all the matching elements + all the non matching elements from the left collection are included in the result set.



Let us understand implementing **Left Outer Join** with an example. Consider the following **Department** and **Employee** classes. Notice that, Employee Mary does not have a department assigned. An inner join will not include her record in the result set, where as a Left Outer Join will.

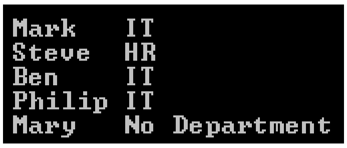
```
public class Department
{
   public int ID { get; set; }
   public string Name { get; set; }

   public static List<Department> GetAllDepartments()
   {
       return new List<Department>()
       {
           new Department { ID = 1, Name = "IT"},
           new Department { ID = 2, Name = "HR"},
       };
   }
}


public class Employee
{
   public int ID { get; set; }
   public string Name { get; set; }
   public int DepartmentID { get; set; }

   public static List<Employee> GetAllEmployees()
   {
       return new List<Employee>()
       {
           new Employee { ID = 1, Name = "Mark", DepartmentID = 1 },
           new Employee { ID = 2, Name = "Steve", DepartmentID = 2 },
           new Employee { ID = 3, Name = "Ben", DepartmentID = 1 },
           new Employee { ID = 4, Name = "Philip", DepartmentID = 1 },
           new Employee { ID = 5, Name = "Mary" }
       };
   }
}
```

Use **DefaultIfEmpty**() method on the results of a group join to implement **Left Outer Join**

Example 1 : Implement a **Left Outer Join** between **Employees** and **Department** collections and print all the Employees and their respective department names. Employees without a department, should display **"No Department"** against their name.

```
var result = from e in Employee.GetAllEmployees()
                    join d in Department.GetAllDepartments()
                    on e.DepartmentID equals d.ID into eGroup
                    from d in eGroup.DefaultIfEmpty()
                    select new
                    {
                         EmployeeName = e.Name,
                         DepartmentName = d == null ? "No Department" : d.Name
                    };

foreach (var v in result)
{
    Console.WriteLine(v.EmployeeName + "\t" + v.DepartmentName);
}
```

Output: Notice that, we also have **Mary** record in spite of she not having a department. So this is effectively a left outer join.



```
Mark   IT
Steve  HR
Ben    IT
Philip IT
Mary   No Department
```

Example 2 : Rewrite **Example 1** using extension method syntax.

```
var result = Employee.GetAllEmployees()
                        .GroupJoin(Department.GetAllDepartments(),
                                e => e.DepartmentID,
                                d => d.ID,
                                (emp, depts) => new { emp, depts })
                        .SelectMany(z => z.depts.DefaultIfEmpty(),
                                (a, b) => new
                                {
                                        EmployeeName = a.emp.Name,
                                        DepartmentName = b == null ? "No Department" : b.Name
                                });

foreach (var v in result)
{
    Console.WriteLine(" " + v.EmployeeName + "\t" + v.DepartmentName);
}
```

To implement **Left Outer Join**, with extension method syntax we use the **GroupJoin()** method along with **SelectMany()** and **DefaultIfEmpty()** methods.

Part 25 – Cross Join

The following are the different types of joins in LINQ
Group Join - Part 21
Inner Join - Part 22
Left Outer Join - Part 24
Cross Join - We will discuss in this video

In this video we will discuss implementing **CROSS JOIN in LINQ**.

Cross join produces a cartesian product i.e when we cross join two sequences, every element in the first collection is combined with every element in the second collection. The total number of elements in the resultant sequence will always be equal to the product of the elements in the two source sequences. The on keyword that specfies the JOIN KEY is not required.

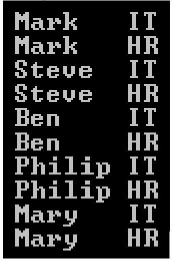
Let us understand implementing Cross Join with an example. Consider the following **Department** and **Employee** classes.

```
public class Department
{
    public int ID { get; set; }
    public string Name { get; set; }

    public static List<Department> GetAllDepartments()
    {
        return new List<Department>()
        {
            new Department { ID = 1, Name = "IT"},
            new Department { ID = 2, Name = "HR"},
        };
    }
}


public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }
    public int DepartmentID { get; set; }

    public static List<Employee> GetAllEmployees()
    {
        return new List<Employee>()
        {
            new Employee { ID = 1, Name = "Mark", DepartmentID = 1 },
            new Employee { ID = 2, Name = "Steve", DepartmentID = 2 },
            new Employee { ID = 3, Name = "Ben", DepartmentID = 1 },
            new Employee { ID = 4, Name = "Philip", DepartmentID = 1 },
            new Employee { ID = 5, Name = "Mary", DepartmentID = 2 },
        };
    }
}
```

Example 1 : Cross Join **Employees** collection with **Departments** collections.

```
var result = from e in Employee.GetAllEmployees()
              from d in Department.GetAllDepartments()
              select new { e, d };

foreach (var v in result)
{
   Console.WriteLine(v.e.Name + "\t" + v.d.Name);
}
```

Output: We have 5 elements in **Employees** collection and 2 elements in **Departments** collection. In the result we have 10 elements, i.e the cartesian product of the elements present in Employees and Departments collection. Notice that every element from the Employees collection is combined with every element in the Departments collection.

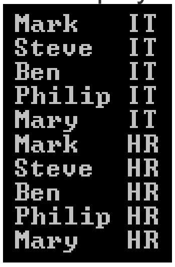


```
Mark    IT
Mark    HR
Steve   IT
Steve   HR
Ben     IT
Ben     HR
Philip  IT
Philip  HR
Mary    IT
Mary    HR
```

Example 2 : Cross Join **Departments** collections with **Employees** collection

```
var result = from d in Department.GetAllDepartments()
              from e in Employee.GetAllEmployees()
              select new { e, d };

foreach (var v in result)
{
   Console.WriteLine(v.e.Name + "\t" + v.d.Name);
}
```

Output: Notice that the output in this case is slightly different from **Example 1**. In this case, every element from the Departments collection is combined with every element in the Employees collection.



```
Mark    IT
Steve   IT
Ben     IT
Philip  IT
Mary    IT
Mark    HR
Steve   HR
Ben     HR
Philip  HR
Mary    HR
```

Example 3 : Rewrite **Example 1** using extension method syntax

To implement **Cross Join** using extension method syntax, we could either use SelectMany() method or Join() method

Implementing cross join using SelectMany()

```
var result = Employee.GetAllEmployees()
```

```
                        .SelectMany(e => Department.GetAllDepartments(), (e, d) => new { e, d
           });

foreach (var v in result)
{
    Console.WriteLine(v.e.Name + "\t" + v.d.Name);
}
```

Implementing cross join using Join()

```
var result = Employee.GetAllEmployees()
                                    .Join(Department.GetAllDepartments(),
                                              e => true,
                                              d => true,
                                              (e, d) => new { e, d });

foreach (var v in result)
{
    Console.WriteLine(v.e.Name + "\t" + v.d.Name);
}
```

Part 26 – Set Operators

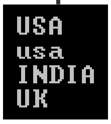
The following operators belong to Set operators category
Distinct
Union
Intersect
Except

In this video we will discuss **Distinct** operator. This operator returns distinct elements from a given collection.

Example 1: Return **distinct** country names. In this example the default comparer is being used and the comparison is case-sensitive, so in the output we see country USA 2 times.

```
string[] countries = { "USA", "usa", "INDIA", "UK", "UK" };

var result = countries.Distinct();

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

Output:

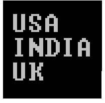


```
USA
usa
INDIA
UK
```

Example 2: For the **comparison to be case-insensitive**, use the other overloaded version of **Distinct()** method to which we can pass a class that implements **IEqualityComparer** as an argument. In this case we see country USA only once in the output.

```
string[] countries = { "USA", "usa", "INDIA", "UK", "UK" };

var result = countries.Distinct(StringComparer.OrdinalIgnoreCase);

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

Output:



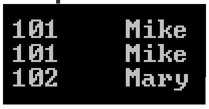
```
USA
INDIA
UK
```

When comparing elements, **Distinct**() works in a slightly different manner with **complex types** like Employee, Customer etc.

Example 3: Notice that in the output we don't get unique employees. This is because, the default comparer is being used which will just check for object references being equal and not the individual property values.

```
List<Employee> list = new List<Employee>()
{
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 102, Name = "Mary"}
};

var result = list.Distinct();

foreach (var v in result)
{
    Console.WriteLine(v.ID + "\t" + v.Name);
}
```

Output:



```
101     Mike
101     Mike
102     Mary
```

To solve the problem in Example 3, there are 3 ways
1. Use the other overloaded version of **Distinct()** method to which we can pass a custom class that implements **IEqualityComparer**
2. Override **Equals()** and **GetHashCode()** methods in **Employee** class
3. Project the properties into a **new anonymous type**, which overrides **Equals()** and **GetHashCode()** methods

Example 4 : Using the overloaded version of **Distinct()** method to which we can pass a custom class that implements **IEqualityComparer**

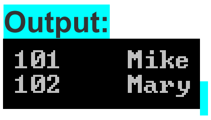
Step 1 : Create a custom class that implements **IEqualityComparer<T>** and implement **Equals()** and **GetHashCode()** methods

```
public class EmployeeComparer : IEqualityComparer<Employee>
{
    public bool Equals(Employee x, Employee y)
    {
        return x.ID == y.ID && x.Name == y.Name;
    }

    public int GetHashCode(Employee obj)
    {
        return obj.ID.GetHashCode() ^ obj.Name.GetHashCode();
    }
}
```

Step 2 : Pass an instance of **EmployeeComparer** as an argument to **Distinct()** method

```
List<Employee> list = new List<Employee>()
{
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 102, Name = "Mary"}
};

var result = list.Distinct(new EmployeeComparer());

foreach (var v in result)
{
    Console.WriteLine(v.ID + "\t" + v.Name);
}
```

Output:



```
101     Mike
102     Mary
```

Example 5 : Override **Equals()** and **GetHashCode()** methods in **Employee** class

```
public class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }

    public override bool Equals(object obj)
    {
        return this.ID == ((Employee)obj).ID && this.Name == ((Employee)obj).Name;
    }

    public override int GetHashCode()
    {
        return this.ID.GetHashCode() ^ this.Name.GetHashCode();
    }
}
```

Example 6 : Project the properties into a **new anonymous type**, which overrides **Equals()** and **GetHashCode()** methods

```
List<Employee> list = new List<Employee>()
{
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 102, Name = "Mary"}
};

var result = list.Select(x => new { x.ID, x.Name }).Distinct();

foreach (var v in result)
{
    Console.WriteLine(" " + v.ID + "\t" + v.Name);
}
```

Part 27 – Union, Intercept and Except (Set Operators)

The following operators belong to Set operators category

Distinct
Union
Intersect
Except

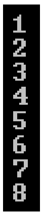
We discussed **Distinct** operator in Part 26. In this video we will discuss **Union**, **Intersect** and **Except** operators.

Union combines two collections into one collection while removing the duplicate elements.

Example 1: numbers1 and numbers2 collections are combined into a single collection. Notice that, the duplicate elements are removed.

```
int[] numbers1 = { 1, 2, 3, 4, 5 };
int[] numbers2 = { 1, 3, 6, 7, 8 };

var result = numbers1.Union(numbers2);

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

Output:



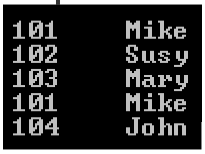
```
1
2
3
4
5
6
7
8
```

When **comparing elements**, just like **Distinct()** method, **Union(), Intersect()** and **Except()** methods work in a slightly different manner with **complex types** like **Employee, Customer** etc.

Example 2 : Notice that in the output the duplicate employee objects are not removed. This is because, the default comparer is being used which will **just check for object references being equal** and not the individual property values.

```
List<Employee> list1 = new List<Employee>()
{
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 102, Name = "Susy"},
    new Employee { ID = 103, Name = "Mary"}
};

List<Employee> list2 = new List<Employee>()
{
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 104, Name = "John"}
};

var result = list1.Union(list2);

foreach (var v in result)
{
    Console.WriteLine(v.ID + "\t" + v.Name);
}
```

Output :



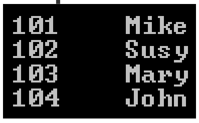
```
101     Mike
102     Susy
103     Mary
101     Mike
104     John
```

Example 3 : To solve the problem in **Example 2**, there are 3 ways
1. Use the other overloaded version of **Union()** method to which we can pass a custom class that implements **IEqualityComparer**
2. Override **Equals()** and **GetHashCode()** methods in **Employee** class
3. Project the properties into a new anonymous type, which overrides **Equals()** and **GetHashCode()** methods

Project the properties into a new anonymous type, which overrides **Equals()** and **GetHashCode()** methods

```
List<Employee> list1 = new List<Employee>()
{
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 102, Name = "Susy"},
    new Employee { ID = 103, Name = "Mary"}
};

List<Employee> list2 = new List<Employee>()
{
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 104, Name = "John"}
};

var result = list1.Select(x => new { x.ID, x.Name })
                  .Union(list2.Select(x => new { x.ID, x.Name }));

foreach (var v in result)
{
    Console.WriteLine(v.ID + "\t" + v.Name);
}
```

Output :



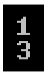
```
101     Mike
102     Susy
103     Mary
104     John
```

Intersect() returns the common elements between the 2 collections.

Example 4 : Return common elements in numbers1 and numbers2 collections.

```
int[] numbers1 = { 1, 2, 3, 4, 5 };
int[] numbers2 = { 1, 3, 6, 7, 8 };

var result = numbers1.Intersect(numbers2);

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

Output :



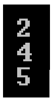
```
1
3
```

Except() returns the elements that are present in the first collection but not in the second collection.

Example 5: Return the elements that are present in the first collection but not in the second collection.

```
int[] numbers1 = { 1, 2, 3, 4, 5 };
int[] numbers2 = { 1, 3, 6, 7, 8 };

var result = numbers1.Except(numbers2);

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

Output :



```
2
4
5
```

Part 28 – Generator Operators

The following operators belong to **Generation Operators** category
Range
Repeat
Empty

Range operator generates a sequence of integers within a specified range. This method has 2 integer parameters. The start parameter specifies the integer to start with and the count parameter specifies the number of sequential integers to generate.

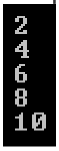
For example to print the first 10 even numbers without using LINQ, we would use a for loop as shown below.

```
for (int i = 1; i <= 10; i++)
{
    if (i % 2 == 0)
    {
        Console.WriteLine(i);
    }
}
```

To achieve the same using LINQ, we can use **Range** method as shown below.

```
var evenNumbers = Enumerable.Range(1, 10).Where(x => x % 2 == 0);

foreach (int i in evenNumbers)
{
    Console.WriteLine(i);
}
```

Output :



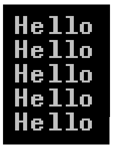
```
2
4
6
8
10
```

Repeat operator is used to generate a sequence that contains one repeated value.

For example the following code returns a string sequence that contains **5 "Hello" string objects** in it.

```
var result = Enumerable.Repeat("Hello", 5);

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

Output:



```
Hello
Hello
Hello
Hello
Hello
```

Empty operator returns an empty sequence of the specified type. For example
Enumerable.Empty<int>() - Returns an empty IEnumerable<int>
Enumerable.Empty<string>() - Returns an empty IEnumerable<string>

The question that comes to our mind is, **what is the use of Empty() method**. Here is an example where we could use Empty() method

There may be scenarios where our application calls a method in a third party application that returns **IEnumerable<int>**. There may be a situation where the third party method returns null. For the purpose of this example, let us assume the third party method is similar to **GetIntegerSequence().**

A NULL reference exception will be thrown if we run the following code

```
class Program
{
    public static void Main()
    {
        IEnumerable<int> result = GetIntegerSequence();

        foreach (var v in result)
        {
            Console.WriteLine(v);
        }
    }

    private static IEnumerable<int> GetIntegerSequence()
    {
        return null;
    }
}
```

One way to fix this is to **check for NULL** before looping thru the items in the result as shown below.

```
class Program
{
    public static void Main()
    {
        IEnumerable<int> result = GetIntegerSequence();

        if (result != null)
        {
            foreach (var v in result)
            {
                Console.WriteLine(v);
            }
        }
    }

    private static IEnumerable<int> GetIntegerSequence()
    {
        return null;
    }
}
```

The other way to fix it, is by using **Empty()** linq method as shown below. Here we are using **NULL-COALESCING operator** that checks if the **GetIntegerSequence()** method returns NULL, in which case the result variable is initialized with an empty **IEnumerable<int>.**

```
class Program
{
    public static void Main()
    {
        IEnumerable<int> result = GetIntegerSequence() ?? Enumerable.Empty<int>();

        foreach (var v in result)
        {
            Console.WriteLine(v);
        }
    }

    private static IEnumerable<int> GetIntegerSequence()
    {
        return null;
    }
}
```

Part 29 – Concat Operator

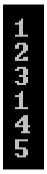
In this video we will discuss
1. The use of Concat operator
2. Difference between Concat and Union operators

Concat operator concatenates two sequences into one sequence.

The following code will **concatenate both the integer sequences** (numbers1 & numbers2) into one integer sequence. Notice that the duplicate elements ARE NOT REMOVED.

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 1, 4, 5 };

var result = numbers1.Concat(numbers2);

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

Output :

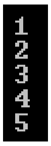


```
1
2
3
1
4
5
```

Now let us perform a **union** between the 2 integer sequences (numbers1 & numbers2). Just like concat operator, union operator also combines the 2 integer sequences (numbers1 & numbers2) into one integer sequence, but notice that the **duplicate elements ARE REMOVED.**

```
int[] numbers1 = { 1, 2, 3 };
int[] numbers2 = { 1, 4, 5 };

var result = numbers1.Union(numbers2);

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

Output :



```
1
2
3
4
5
```

What is the difference between Concat and Union operators?
Concat operator combines 2 sequences into 1 sequence. Duplicate elements are not removed. It simply returns the items from the first sequence followed by the items from the second sequence.

Union operator also combines 2 sequences into 1 sequence, but will remove the duplicate elements.

Part 30 – Sequence Equal Method

SequenceEqual() method is used to determine whether two sequences are equal. This method returns true if the sequences are equal otherwise false.

For 2 sequences to be equal
1. Both the sequences should have same number of elements and
2. Same values should be present in the same order in both the sequences

Example 1 : SequenceEqual() returns true.

```
string[] countries1 = { "USA", "India", "UK" };
string[] countries2 = { "USA", "India", "UK" };

var result = countries1.SequenceEqual(countries2);

Console.WriteLine("Are Equal = " + result);
```

Example 2 : In this case, **SequenceEqual()** returns false, as the default comparison is case sensitive.

```
string[] countries1 = { "USA", "INDIA", "UK" };
string[] countries2 = { "usa", "india", "uk" };

var result = countries1.SequenceEqual(countries2);

Console.WriteLine("Are Equal = " + result);
```

Example 3: If we want the comparison to be **case-insensitive**, then use the other overloaded version of SequenceEqual() method to which we can pass an alternate comparer.

```
string[] countries1 = { "USA", "INDIA", "UK" };
string[] countries2 = { "usa", "india", "uk" };

var result =
countries1.SequenceEqual(countries2, StringComparer.OrdinalIgnoreCase);

Console.WriteLine("Are Equal = " + result);
```

Example 4 : SequenceEqual() returns false. This is because, although both the sequences contain same data, the data is not present in the same order.

```
string[] countries1 = { "USA", "INDIA", "UK" };
string[] countries2 = { "UK", "INDIA", "USA" };

var result = countries1.SequenceEqual(countries2);

Console.WriteLine("Are Equal = " + result);
```

Example 5 : To fix the problem in Example 4, use **OrderBy()** to sort data in the source sequences.

```
string[] countries1 = { "USA", "INDIA", "UK" };
```

```
string[] countries2 = { "UK", "INDIA", "USA" };

var result = countries1.OrderBy(c => c).SequenceEqual(countries2.OrderBy(c => c));

Console.WriteLine("Are Equal = " + result);
```

Example 6 : When comparing complex types, the default comparer will only check if the object references are equal. So, in this case SequenceEqual() returns false.

```
List<Employee> list1 = new List<Employee>()
{
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 102, Name = "Susy"},
};

List<Employee> list2 = new List<Employee>()
{
    new Employee { ID = 101, Name = "Mike"},
    new Employee { ID = 102, Name = "Susy"},
};

var result = list1.SequenceEqual(list2);

Console.WriteLine("Are Equal = " + result);
```

To solve the problem in Example 6, there are 3 ways
1. Use the other overloaded version of SequenceEqual() method to which we can pass a custom class that implements IEqualityComparer
2. Override Equals() and GetHashCode() methods in Employee class
3. Project the properties into a new anonymous type, which overrides Equals() and GetHashCode() methods

We discussed implementing these 3 options for Distinct() method in Part 26 of LINQ Tutorial. In the same way these options can be implemented for SequenceEqual() method.

Part 31 – Quantifiers

The following methods belong to Quantifiers category
All
Any
Contains

All these methods return true or false depending on whether if some or all of the elements in a sequence satisfy a condition.

All() method returns true if all the elements in a sequence satisfy a given condition, otherwise false.

Example 1 : Returns true, as all the numbers are less than 10

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

var result = numbers.All(x => x < 10);

Console.WriteLine(result);
```

There are 2 overloaded versions of **Any()** method. The version without any parameters checks if the sequence contains at least one element. The other version with a predicate parameter checks if the sequence contains at least one element that satisfies a given condition.

Example 2 : Returns true as the sequence contains at least one element

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

var result = numbers.Any();

Console.WriteLine(result);
```

Example 3 : Returns false as the sequence does not contain any element that satisfies the given condition (No element in the sequence is greater than 10)

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

var result = numbers.Any(x => x > 10);

Console.WriteLine(result);
```

There are 2 overloaded versions of the **Contains()** method. One of the overloaded version checks if the sequence contains a specified element using the default equality comparer. The other overloaded version checks if the sequence contains a specified element using an alternate equality comparer.

Example 4 : Returns true as the sequence contains number 3. In this case the default equality comparer is used.

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

var result = numbers.Contains(3);
```

```
Console.WriteLine(result);
```

Example 5 : Returns true. In this case we are using an alternate equality comparer (StringComparer) for the comparison to be case-insensitive.

```
string[] countries = { "USA", "INDIA", "UK" };

var result = countries.Contains("india", StringComparer.OrdinalIgnoreCase);

Console.WriteLine(result);
```

When comparing complex types like **Employee, Customer** etc, the default comparer will only check if the object references are equal, and not the individual property values of the objects that are being compared.

Example 6 : Returns false, as the default comparer will only check if the object references are equal.

```
List<Employee> employees = new List<Employee>()
{
    new Employee { ID = 101, Name = "Rosy"},
    new Employee { ID = 102, Name = "Susy"}
};

var result = employees.Contains(new Employee { ID = 101, Name = "Rosy" });

Console.WriteLine(result);
```

To solve the problem in Example 6, there are 3 ways
1. Use the other overloaded version of **Contains()** method to which we can pass a custom class that implements **IEqualityComparer**
2. Override **Equals()** and **GetHashCode()** methods in **Employee** class
3. Project the properties into a new anonymous type, which overrides **Equals()** and **GetHashCode()** methods

We discussed implementing these **3 options for Distinct() method in Part 26 of LINQ Tutorial**. In the same way these options can be implemented for Contains() method.

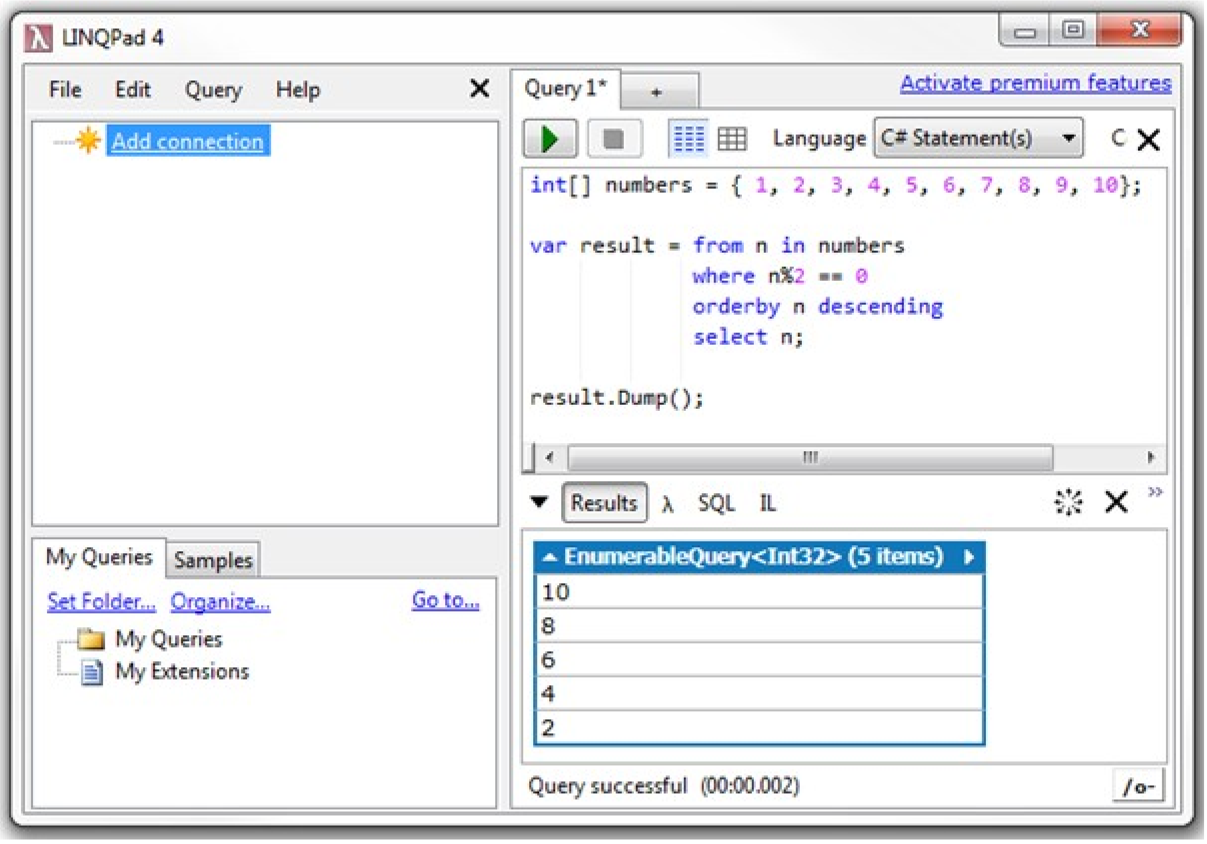
32 – LinqPad

What is LinqPad

LinqPad is a free tool that you can download from http://www.linqpad.net. It helps learn, write and test linq queries.

Copy and paste the following LINQ query in LinqPad. To execute the query, you can either press the **Green Execute button** on the LinqPad or press **F5**. **Dump()** method is similar to **Console.WriteLine()** in a console application.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

var result = from n in numbers
             where n % 2 == 0
             orderby n descending
             select n;
result.Dump();
```



```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

var result = from n in numbers
                    where n%2 == 0
                    orderby n descending
                    select n;

result.Dump();
```

```
EnumerableQuery<Int32> (5 items)
10
8
6
4
2
```

Query successful (00:00.002)

Notice that the results of the query are shown in the **Results** window. Next to the results window, you also have the following options

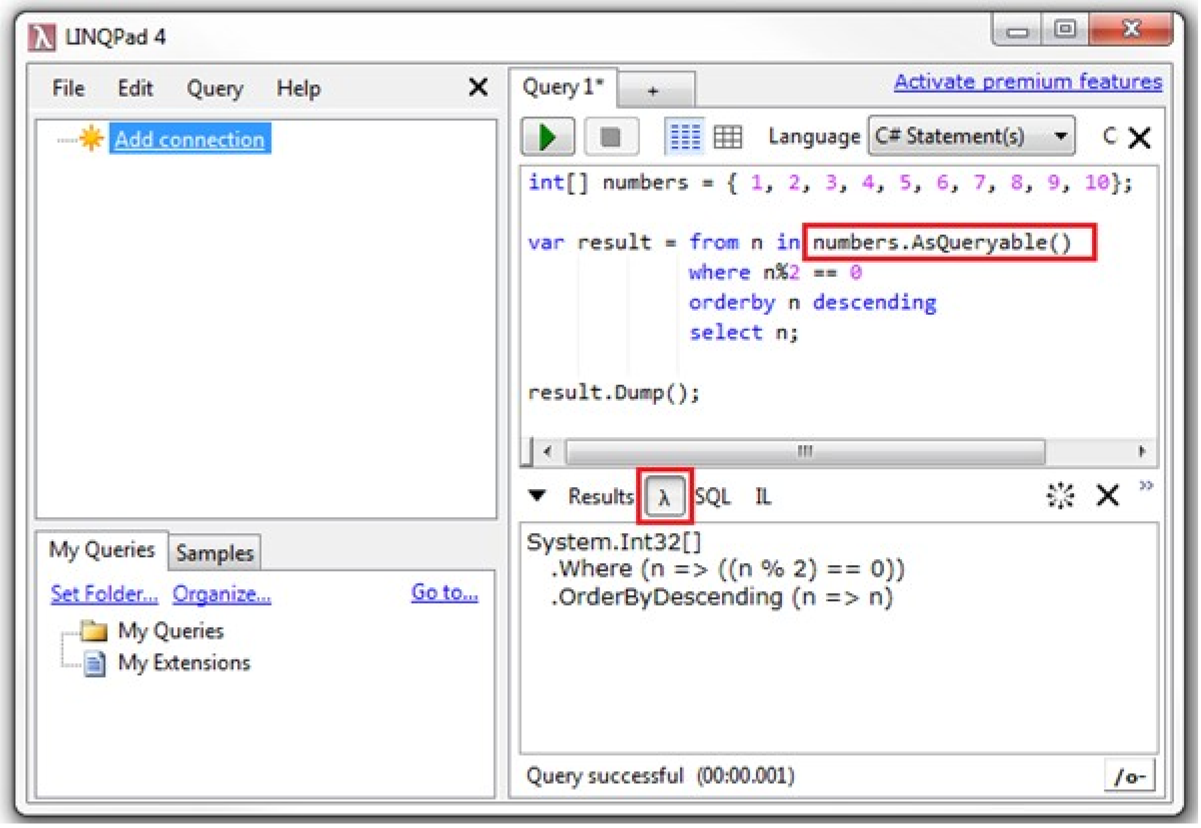
1. ? (lambda Symbol) - Use this button to get the lambda equivalent of a LINQ Query

2. SQL - Shows the generated SQL statement that will be executed against the underlying database

3. IL - Shows the Intermediate Language code

For the above query, Lambda and SQL windows will not show anything. To get the

Lambda equivalent of a LINQ query, use **.AsQueryable()** on the source collection as shown below.



```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

var result = from n in numbers.AsQueryable()
                    where n%2 == 0
                    orderby n descending
                    select n;

result.Dump();
```

```
System.Int32[]
  .Where (n => ((n % 2) == 0))
  .OrderByDescending (n => n)
```

Query successful (00:00.001)

AsQueryable() can also be used on the source collection as shown below.

```
var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }.AsQueryable();

var result = from n in numbers
             where n % 2 == 0
             orderby n descending
             select n;

result.Dump();
```

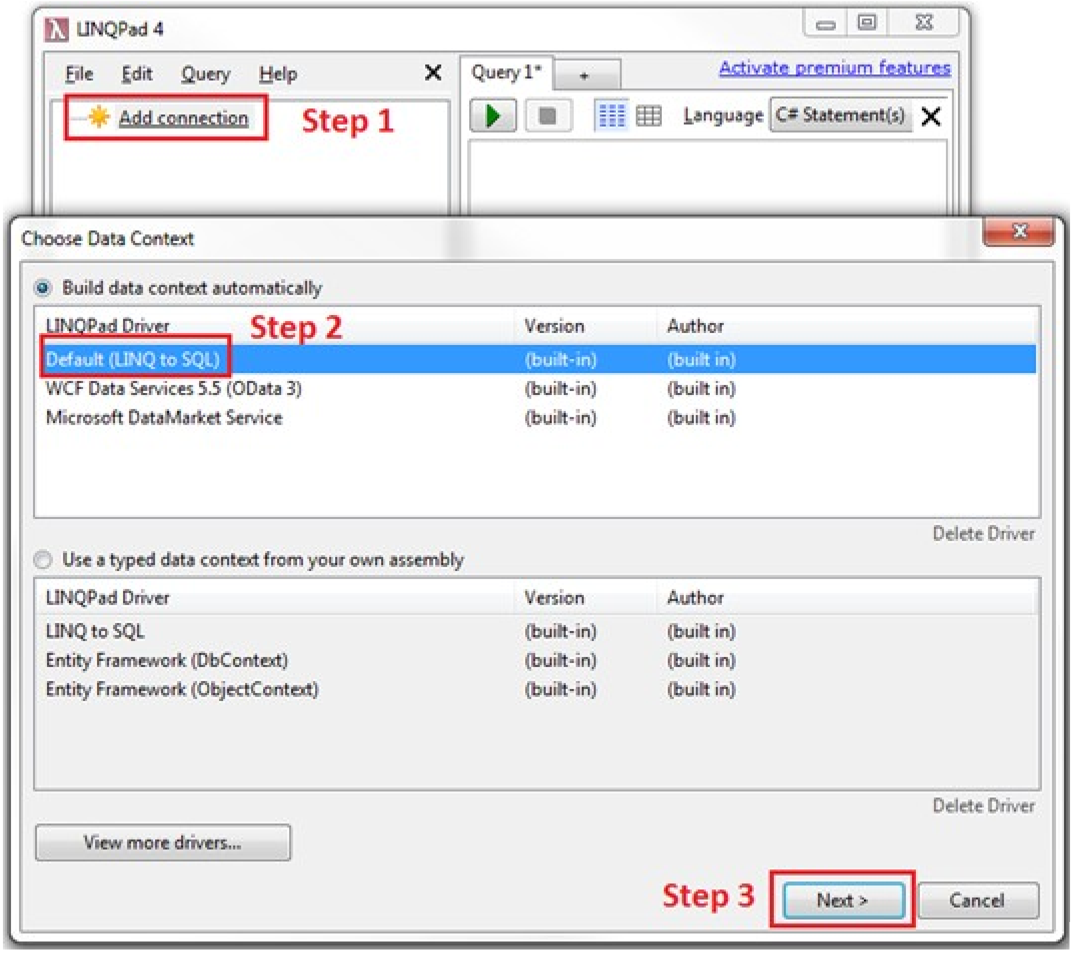
LinqPad can execute

1. Statements
2. Expressions
3. Program

LinqPad can also be used with databases and WCF Data Services.

Adding a database connection in LinqPad

Step 1 : Click "Add connection"
Step 2 : Under LinqPad Driver, select "Default (LINQ to SQL)"
Step 3 : Click Next



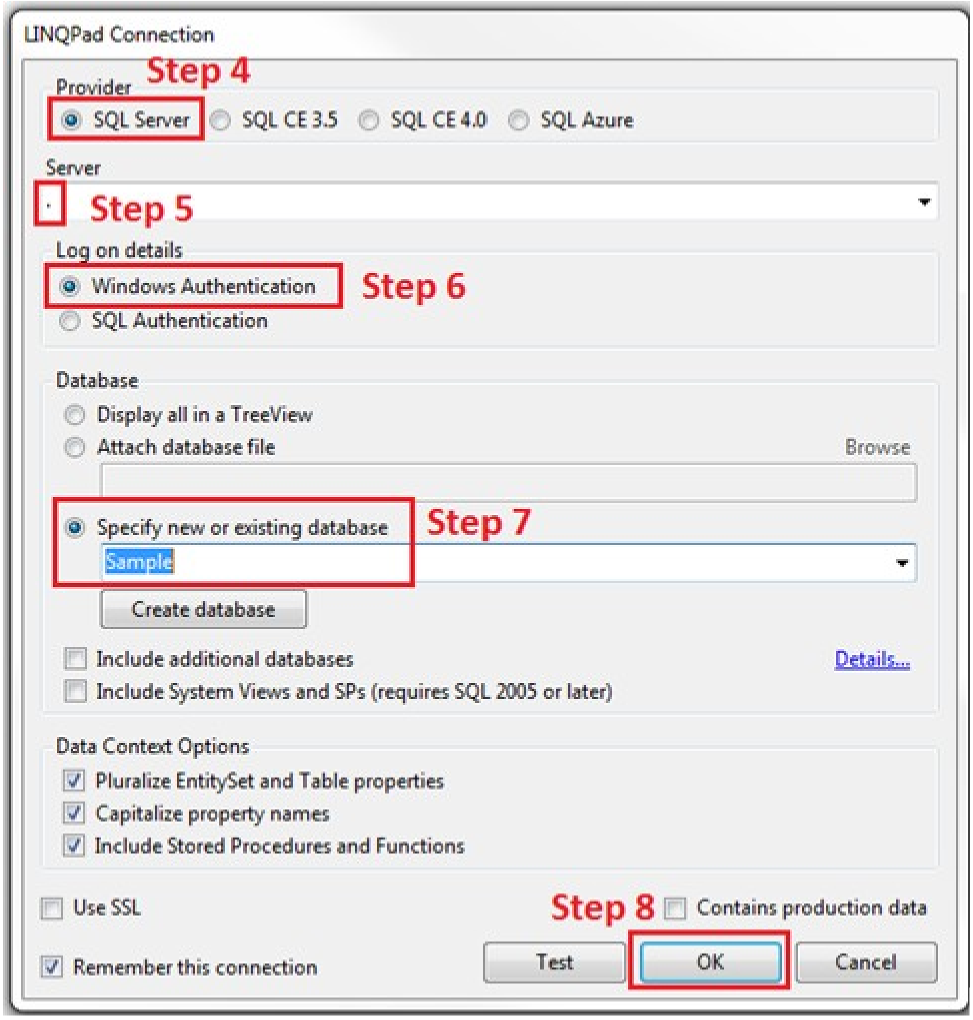
● Build data context automatically

LINQPad Driver	Version	Author
Default (LINQ to SQL)	(built-in)	(built in)
WCF Data Services 5.5 (OData 3)	(built-in)	(built in)
Microsoft DataMarket Service	(built-in)	(built in)

○ Use a typed data context from your own assembly

LINQPad Driver	Version	Author
LINQ to SQL	(built-in)	(built in)
Entity Framework (DbContext)	(built-in)	(built in)
Entity Framework (ObjectContext)	(built-in)	(built in)

Step 4 : Select "SQL Server" as the "Provider"
Step 5 : Specify the Server Name. In my case I am connecting to the local SQL Server. So I used . (DOT)
Step 6 : Select the Authentication
Step 7 : Select the Database
Step 8 : Click OK



LINQPad Connection

Provider Step 4

● SQL Server ○ SQL CE 3.5 ○ SQL CE 4.0 ○ SQL Azure

Server

. Step 5

Log on details

● Windows Authentication Step 6

○ SQL Authentication

Database

○ Display all in a TreeView

○ Attach database file Browse

● Specify new or existing database Step 7

Sample

□ Include additional databases Details...

□ Include System Views and SPs (requires SQL 2005 or later)

Data Context Options

☑ Pluralize EntitySet and Table properties

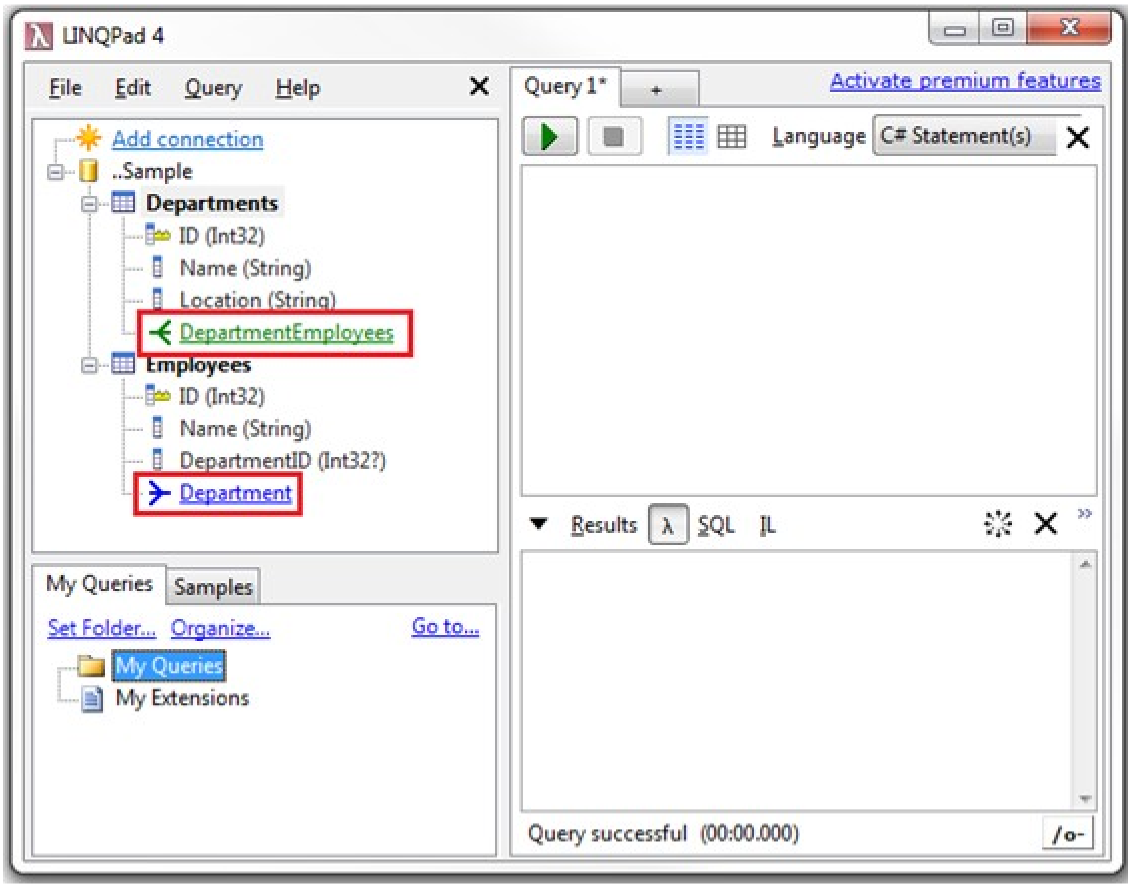
☑ Capitalize property names

☑ Include Stored Procedures and Functions

□ Use SSL Step 8 □ Contains production data

☑ Remember this connection

At this point LinqPad connects to the database, and shows all the table entities. The relationships between the entities are also shown. The Green Split arrow indicates One-to-Many relationship and the Blue Split Arrow indicates Many-to-One relationship.

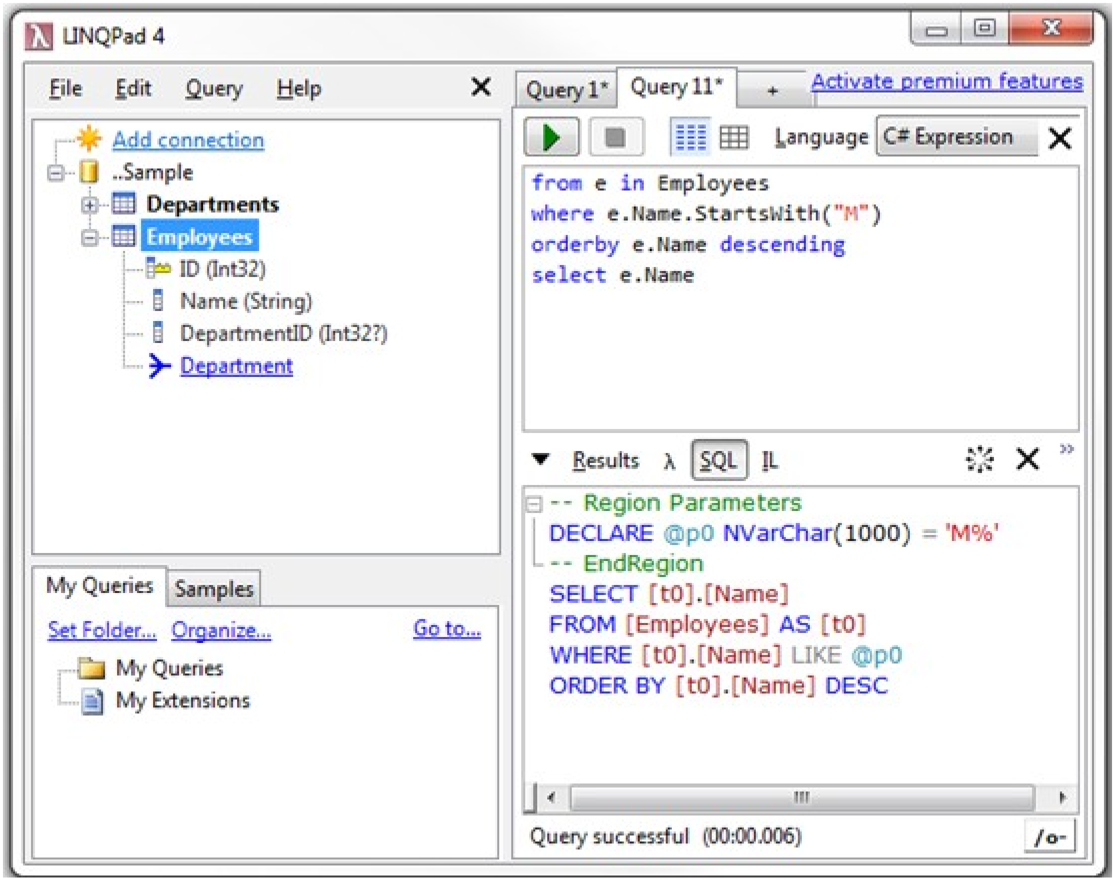


We can now start writing linq queries targeting the SQL Server database.

The following LINQ query fetches **all the employee names that start with letter 'M' and sorts them in descending order**

```
from e in Employees
where e.Name.StartsWith("M")
orderby e.Name descending
select e.Name
```

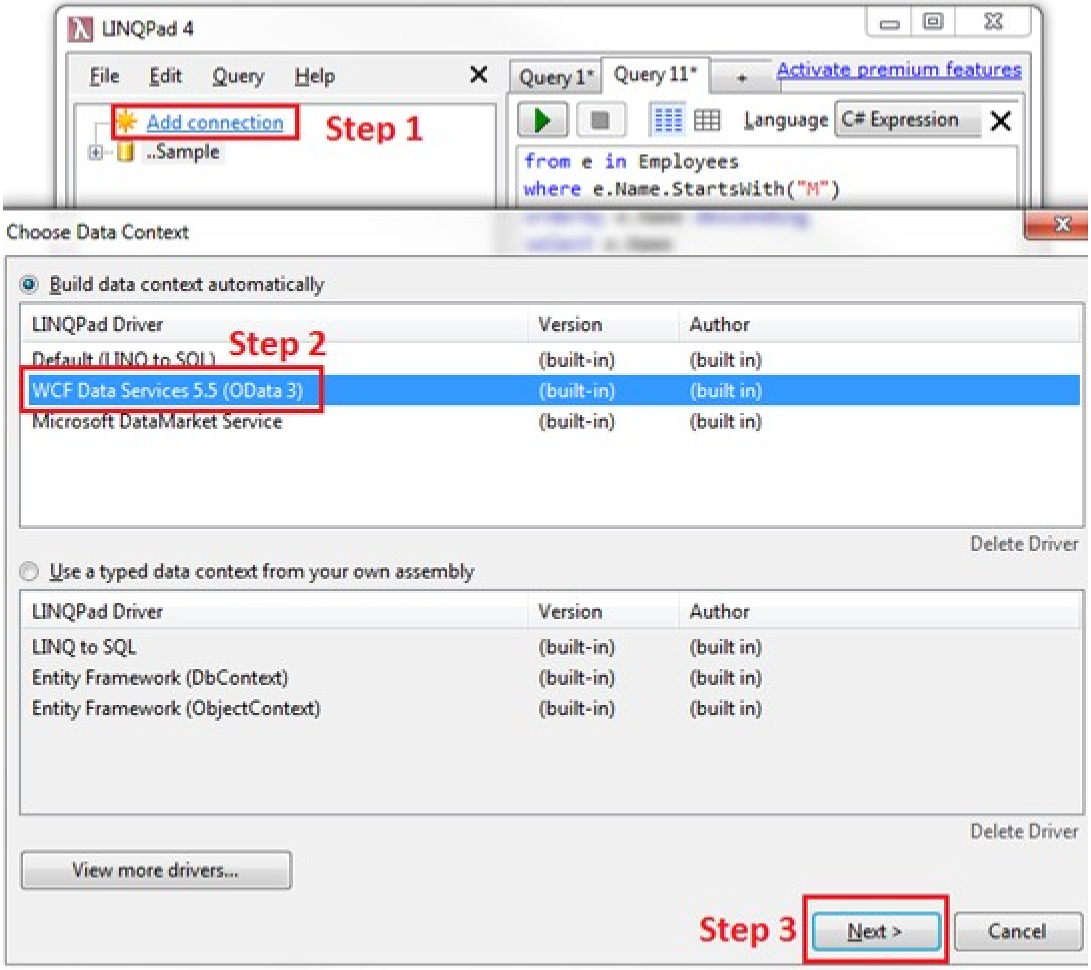
After executing the query, click on the SQL button to see the Transact-SQL that is generated.



```
from e in Employees
where e.Name.StartsWith("M")
orderby e.Name descending
select e.Name
```

```
-- Region Parameters
DECLARE @p0 NVarChar(1000) = 'M%'
-- EndRegion
SELECT [t0].[Name]
FROM [Employees] AS [t0]
WHERE [t0].[Name] LIKE @p0
ORDER BY [t0].[Name] DESC
```

Adding a WCF Data Services connection in LinqPad
Step 1 : Click "Add connection"
Step 2 : Under LinqPad Driver, select "WCF Data Services"
Step 3 : Click Next



```
from e in Employees
where e.Name.StartsWith("M")
```

Choose Data Context

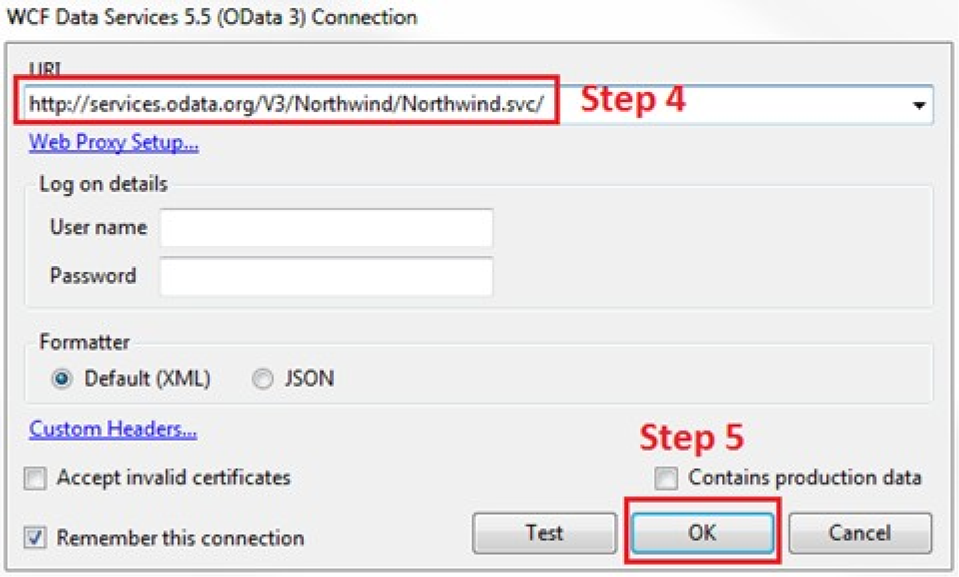
● Build data context automatically

LINQPad Driver	Version	Author
Default (LINQ to SQL)	(built-in)	(built in)
WCF Data Services 5.5 (OData 3)	(built-in)	(built in)
Microsoft DataMarket Service	(built-in)	(built in)

○ Use a typed data context from your own assembly

LINQPad Driver	Version	Author
LINQ to SQL	(built-in)	(built in)
Entity Framework (DbContext)	(built-in)	(built in)
Entity Framework (ObjectContext)	(built-in)	(built in)

Step 4 : Type the URI for the WCF Data Service.
http://services.odata.org/V3/Northwind/Northwind.svc/
Step 5 : Click OK



WCF Data Services 5.5 (OData 3) Connection

URI: http://services.odata.org/V3/Northwind/Northwind.svc/

Log on details
User name
Password

Formatter
● Default (XML) ○ JSON

□ Accept invalid certificates
□ Contains production data
☑ Remember this connection

We can now start writing linq queries targeting the WCF Data Service.

The following LINQ query fetches **all the product names that start with letter 'C' and**

sorts them in ascending order

```
from p in Products
where p.ProductName.StartsWith("C")
orderby p.ProductName ascending
select p
```