# StartWith

string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };

var selectedTeams = from t in teams // определяем каждый объект из teams как t

where t.ToUpper().StartsWith("Б") //фильтрация по критерию

orderby t // упорядочиваем по возрастанию

select t; // выбираем объект

foreach (string s in selectedTeams)

Console.WriteLine(s);

string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };

var selectedTeams = teams.Where(t => t.ToUpper().StartsWith("Б")).OrderBy(t => t);

foreach (string s in selectedTeams)

Console.WriteLine(s);

# All methods

### Список используемых методов расширения LINQ

* **Select**: определяет проекцию выбранных значений
* **Where**: определяет фильтр выборки
* **OrderBy**: упорядочивает элементы по возрастанию
* **OrderByDescending**: упорядочивает элементы по убыванию
* **ThenBy**: задает дополнительные критерии для упорядочивания элементов возрастанию
* **ThenByDescending**: задает дополнительные критерии для упорядочивания элементов по убыванию
* **Join**: соединяет две коллекции по определенному признаку
* **GroupBy**: группирует элементы по ключу
* **ToLookup**: группирует элементы по ключу, при этом все элементы добавляются в словарь
* **GroupJoin**: выполняет одновременно соединение коллекций и группировку элементов по ключу
* **Reverse**: располагает элементы в обратном порядке
* **All**: определяет, все ли элементы коллекции удовлятворяют определенному условию
* **Any**: определяет, удовлетворяет хотя бы один элемент коллекции определенному условию
* **Contains**: определяет, содержит ли коллекция определенный элемент
* **Distinct**: удаляет дублирующиеся элементы из коллекции
* **Except**: возвращает разность двух коллекцию, то есть те элементы, которые содератся только в одной коллекции
* **Union**: объединяет две однородные коллекции
* **Intersect**: возвращает пересечение двух коллекций, то есть те элементы, которые встречаются в обоих коллекциях
* **Count**: подсчитывает количество элементов коллекции, которые удовлетворяют определенному условию
* **Sum**: подсчитывает сумму числовых значений в коллекции
* **Average**: подсчитывает cреднее значение числовых значений в коллекции
* **Min**: находит минимальное значение
* **Max**: находит максимальное значение
* **Take**: выбирает определенное количество элементов
* **Skip**: пропускает определенное количество элементов
* **TakeWhile**: возвращает цепочку элементов последовательности, до тех пор, пока условие истинно
* **SkipWhile**: пропускает элементы в последовательности, пока они удовлетворяют заданному условию, и затем возвращает оставшиеся элементы
* **Concat**: объединяет две коллекции
* **Zip**: объединяет две коллекции в соответствии с определенным условием
* **First**: выбирает первый элемент коллекции
* **FirstOrDefault**: выбирает первый элемент коллекции или возвращает значение по умолчанию
* **Single**: выбирает единственный элемент коллекции, если коллекция содердит больше или меньше одного элемента, то генерируется исключение
* **SingleOrDefault**: выбирает первый элемент коллекции или возвращает значение по умолчанию
* **ElementAt**: выбирает элемент последовательности по определенному индексу
* **ElementAtOrDefault**: выбирает элемент коллекции по определенному индексу или возвращает значение по умолчанию, если индекс вне допустимого диапазона
* **Last**: выбирает последний элемент коллекции
* **LastOrDefault**: выбирает последний элемент коллекции или возвращает значение по умолчанию

# SelectMany

var r = users.SelectMany(u => u.Languages,

(u, l) => new { User = u, Lang = l })

.Where(u => u.Lang == "английский" && u.User.Age < 28)

.Select(u => u.User);

# Select

List<User> users = new List<User>();

users.Add(new User { Name = "Sam", Age = 43 });

users.Add(new User { Name = "Tom", Age = 33 });

var names = from u in users select u.Name;

# Select anonymous type

var items = users.Select(u => new

{

FirstName = u.Name,

DateOfBirth = DateTime.Now.Year - u.Age

});

# Let

List<User> users = new List<User>()

{

new User { Name = "Sam", Age = 43 },

new User { Name = "Tom", Age = 33 }

};

var people = from u in users

let name = "Mr. " + u.Name

select new

{

Name = name,

Age = u.Age

};

# OrderBy

int[] numbers = { 3, 12, 4, 10, 34, 20, 55, -66, 77, 88, 4 };

var orderedNumbers = from i in numbers

orderby i

select i;

foreach (int i in orderedNumbers)

Console.WriteLine(i);

# OrderBy by class property

var sortedUsers = from u in users

orderby u.Name

select u;

# Two list selecting

List<User> users = new List<User>()

{

new User { Name = "Sam", Age = 43 },

new User { Name = "Tom", Age = 33 }

};

List<Phone> phones = new List<Phone>()

{

new Phone {Name="Lumia 630", Company="Microsoft" },

new Phone {Name="iPhone 6", Company="Apple"},

};

var people = from user in users

from phone in phones

select new { Name = user.Name, Phone = phone.Name };

foreach (var p in people)

Console.WriteLine("{0} - {1}", p.Name, p.Phone);

};

# ThenBy

var result = users.OrderBy(u => u.Name).ThenBy(u => u.Age).ThenBy(u => u.Name.Length);

var result = from user in users

orderby user.Name, user.Age, user.Name.Length

select user;

# Except

string[] soft = { "Microsoft", "Google", "Apple" };

string[] hard = { "Apple", "IBM", "Samsung" };

// разность множеств

var result = soft.Except(hard);

# Intersect

string[] soft = { "Microsoft", "Google", "Apple" };

string[] hard = { "Apple", "IBM", "Samsung" };

// пересечение множеств

var result = soft.Intersect(hard);

# Union|Concat|Distinct

string[] soft = { "Microsoft", "Google", "Apple" };

string[] hard = { "Apple", "IBM", "Samsung" };

// объединение множеств

var result = soft.Union(hard);

var result = soft.Concat(hard).Distinct();

# Aggragate

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

int query = numbers.Aggregate((x, y) => x - y);

int query = 1 - 2 - 3 - 4 – 5

# Count

int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };

int size = (from i in numbers where i % 2 == 0 && i > 10 select i).Count();

Console.WriteLine(size);

# Sum

int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };

List<User> users = new List<User>()

{

    new User { Name = "Tom", Age = 23 },

    new User { Name = "Sam", Age = 43 },

    new User { Name = "Bill", Age = 35 }

};

int sum1 = numbers.Sum();

decimal sum2 = users.Sum(n => n.Age);

# Skip and Take

int[] numbers = { -3, -2, -1, 0, 1, 2, 3 };

var result = numbers.Take(3);

foreach (int i in result)

Console.WriteLine(i);

var result = numbers.Skip(3);

# TakeWhile

string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };

foreach (var t in teams.TakeWhile(x => x.StartsWith("Б")))

Console.WriteLine(t);

# SkipWhile

string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };

foreach (var t in teams.SkipWhile(x => x.StartsWith("Б")))

Console.WriteLine(t);

# Group By

var phoneGroups = from phone in phones

group phone by phone.Company;

foreach (IGrouping<string, Phone> g in phoneGroups)

{

Console.WriteLine(g.Key);

foreach (var t in g)

Console.WriteLine(t.Name);

Console.WriteLine();

}

var phoneGroups = phones.GroupBy(p => p.Company)

                        .Select(g => new { Name = g.Key, Count = g.Count() });

# Вложенный запрос

var phoneGroups = phones.GroupBy(p => p.Company)

                        .Select(g => new

                        {

                            Name = g.Key,

                            Count = g.Count(),

                            Phones = g.Select(p =>p)

                        });

# Join

var result = players.Join(teams, // второй набор

p => p.Team, // свойство-селектор объекта из первого набора

t => t.Name, // свойство-селектор объекта из второго набора

(p, t) => new { Name = p.Name, Team = p.Team, Country = t.Country }); // результат

var result = from pl in players

join t in teams on pl.Team equals t.Name

select new { Name = pl.Name, Team = pl.Team, Country = t.Country };

# GroupJoin

var result2 = teams.GroupJoin(

players, // второй набор

t => t.Name, // свойство-селектор объекта из первого набора

pl => pl.Team, // свойство-селектор объекта из второго набора

(team, pls) => new // результирующий объект

{

Name = team.Name,

Country = team.Country,

Players = pls.Select(p => p.Name)

});

foreach (var team in result2)

{

Console.WriteLine(team.Name);

foreach (string player in team.Players)

{

Console.WriteLine(player);

}

Console.WriteLine();

}

# Zip

var result2 = players.Zip(teams,

(player, team) => new

{

Name = player.Name,

Team = team.Name,

Country = team.Country

});

foreach (var player in result2)

{

Console.WriteLine("{0} - {1} ({2})", player.Name, player.Team, player.Country);

Console.WriteLine();

}

# All|Any

List<User> users = new List<User>()

{

new User { Name = "Tom", Age = 23 },

new User { Name = "Sam", Age = 43 },

new User { Name = "Bill", Age = 35 }

};

bool result1 = users.All(u => u.Age > 20); // true

bool result1 = users.Any(u => u.Age < 20); //false хотя бы один

# Отложенное и немедленное выполнение

var selectedTeams = from t in teams where t.ToUpper().StartsWith("Б") orderby t select t;

// выполнение LINQ-запроса

foreach (string s in selectedTeams)

Console.WriteLine(s);

То есть фактическое выполнение запроса происходит не в строке определения: var selectedTeams = from t..., а при переборе в цикле foreach.

После определения запроса он может выполняться множество раз. И до выполнения запроса источник данных может изменяться. Чтобы более наглядно увидеть это, мы можем изменить какой-либо элемент до перебора выборки:

var selectedTeams = from t in teams where t.ToUpper().StartsWith("Б") orderby t select t;

// изменение массива после определения LINQ-запроса

teams[1] = "Ювентус";

// выполнение LINQ-запроса

Немедленное выполнение запроса

С помощью ряда методов мы можем применить немедленное выполнение запроса.Это методы, которые возвращают одно атомарное значение или один элемент.Например, Count(), Average(), First() / FirstOrDefault(), Min(), Max() и т.д.Например, метод Count() возвращает числовое значение, которое представляет количество элементов в полученной последовательности.А метод First() возвращает первый элемент последовательности. Но чтобы выполнить эти методы, вначале надо получить саму последовательность, то есть результат запроса, и пройтись по ней циклом foreach, который вызывается неявно внутри структуры запроса.

Рассмотрим пример с методом Count(), который возвращает число элементов последовательности:

string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };

// определение и выполнение LINQ-запроса

int i = (from t in teams

where t.ToUpper().StartsWith("Б")

orderby t

select t).Count();

Console.WriteLine(i); //3

teams[1] = "Ювентус";

Console.WriteLine(i); //3

--------

string[] teams = { "Бавария", "Боруссия", "Реал Мадрид", "Манчестер Сити", "ПСЖ", "Барселона" };

// выполнение LINQ-запроса

var selectedTeams = (from t in teams

where t.ToUpper().StartsWith("Б")

orderby t

select t).ToList<string>();

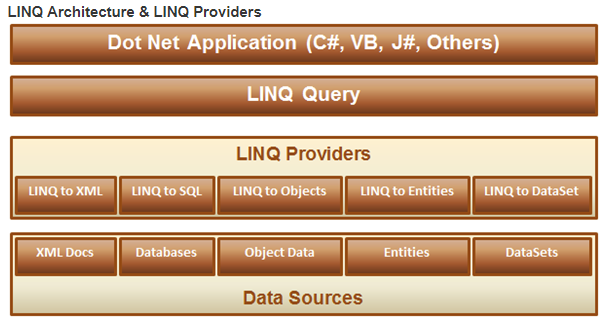
// изменение массива никак не затронет список selectedTeams

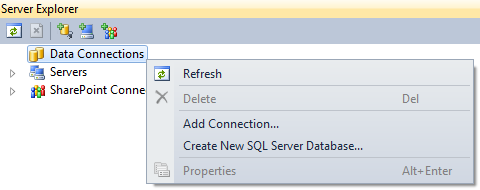
teams[1] = "Ювентус";

foreach (string s in selectedTeams)

Console.WriteLine(s);

# Linq providers



**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

# SampleDataContext

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();

        }

    }

# **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=fromstudentinStudent.GetAllStudents**()**

wherestudent.Gender=="Male"

selectstudent**;**  
  
**To bind the results of this LINQ query to a GridView**  
GridView1.DataSource = students;  
GridView1.DataBind();

# **Aggregate Functions**

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();

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

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

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

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

            string result = countries.Aggregate((a, b) => a + ", " + b);

  int[] Numbers = { 2, 3, 4, 5 };

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

# 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.

**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);

 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;

            }

        }

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);

            }

# Model from Database

Add-> Items->ADO EDM

# EmployeeDBContext

 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();

            }

# Projections

**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

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

var result = Employee.GetAllEmployees().Select(emp => new

                    {

                        FirstName = emp.FirstName,

                        Gender = emp.Gender

                    });

omputes **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

});

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

                 });

# SelectMany

public static List<Student> GetAllStudetns()

    {

        List<Student> listStudents = new List<Student>

        {

            new Student

            {

                Name = "Tom",

                Gender = "Male",

                Subjects = new List<string> { "ASP.NET", "C#" }

            }

IEnumerable<string> allSubjects = Student.GetAllStudetns().SelectMany(s => s.Subjects);

foreach (string subject in allSubjects)

{

    Console.WriteLine(subject);

}

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;

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);

}

ewrite **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);

}

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);

}

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);

}

# **Difference between Select and SelectMany in LINQ**

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);

}

# **Ordering Operators in LINQ**

List<Student> listStudents = new List<Student>

        {

            new Student

            {

                StudentID= 101,

                Name = "Tom",

                TotalMarks = 800

            }

IEnumerable<Student> result = from student in Student.GetAllStudents()

                                                      orderby student.Name

                                                      select student;

IEnumerable<Student> result = Student.GetAllStudents().OrderByDescending(s => s.Name);

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

**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);

}

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);

}

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);

}

# **Partitioning Operators in LINQ**

Take  
Skip  
TakeWhile  
SkipWhile

**ake**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

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);

}

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);

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);

}

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);

}

# **Implement paging using skip and take operators**

List<Student> listStudents = new List<Student>

        {

            new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },

            new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },

  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);

# **LINQ query deferred 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

**LINQ Deferred Execution Example**

 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);

            }

**LINQ Immediate Execution Example 1**

 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);

            }

# **Conversion Operators in LINQ**

Convert int array to List<int>

  int[] numbers = { 1, 2, 3, 4, 5 };

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

            foreach (int i in result)

            {

                Console.WriteLine(i);

            }

Convert int array to List<int>

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

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

            foreach (int i in result)

            {

                Console.WriteLine(i);

            }

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

      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);

            }

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

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);

            }

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

  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);

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

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);

                }

            }

# **Cast and OfType operators in LINQ**

**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.

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);

            }

**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.

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);

            }

**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

# **AsEnumerable and AsQueryable in LINQ**

**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.

 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**

varresult=dbContext.Employees.Where**(**x=>x.Gender=="Male"**)**

.OrderByDescending**(**x=>x.Salary**)**.Take**(**5**);**  
  
**TO**

varresult=dbContext.Employees.AsEnumerable**()**

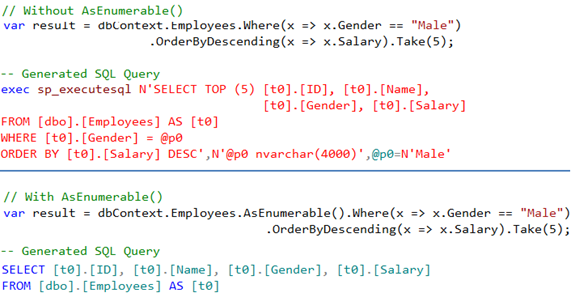
.Where**(**x=>x.Gender=="Male"**)**

.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:**   
   
  
**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]

# **GroupBy in LINQ**

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 }

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:**   
linq group by

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:**   


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)

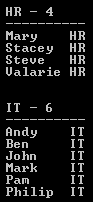
    {

        Console.WriteLine(employee.Name + "\t" + employee.Department);

    }

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

}

**Output:**   


Employee.GetAllEmployees().GroupBy(x => x.Department).OrderBy(c => c.Key)  
.Select(x => new  
{  
Key=x.Key,  
employee = x.OrderBy(c => c.Name)  
});

# **Group by multiple keys in linq**

  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" },

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();

}

# **Element Operators in LINQ**

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);

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

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);

**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.

**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);

**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

**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

# **Group Join in LINQ**

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}

        };

    }

}

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)

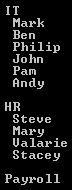
    {

        Console.WriteLine(" " + employee.Name);

    }

    Console.WriteLine();

}

**Output:**   
   
  
**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

                                                       };

# **Difference between group join and inner join in linq**

**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,

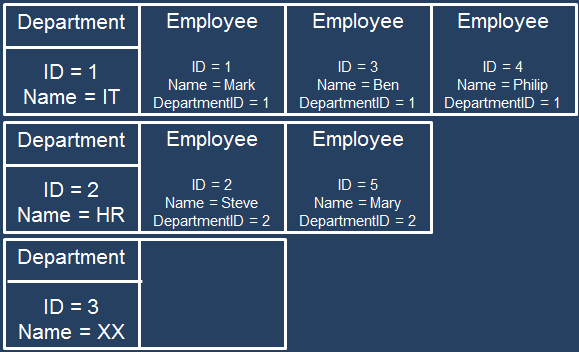
                                         (department, employees) => new

                                         {

                                              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,

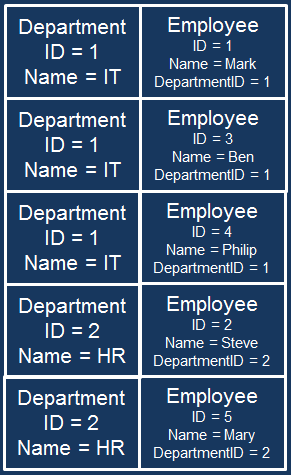
                                      d => d.ID, (employee, department) => new

                                      {

                                           e = employee,

                                           d = department

                                      });

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

}

# **Left Outer Join in LINQ**

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.   
   
  
**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);

}

# **Cross Join in LINQ**

 return new List<Department>()

        {

            new Department { ID = 1, Name = "IT"},

            new Department { ID = 2, Name = "HR"},

        };

  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 },

        };

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.   
   
  
**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.   
   
  
**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);

}

# **Set operators in LINQ**

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);

}

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);

}

**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:**   
linq distinct c# example   
  
**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:**   
iequalitycomparer example   
  
**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);

}

# **Union, Intersect and Except operators in LINQ**

**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:**   
union in linq

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 :**   
   
  
**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

**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);

}

**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 :**   
except in linq

# **Generation Operators in LINQ**

**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 :    
linq range example

**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:**   
linq repeat n times

**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>

  IEnumerable<int> result = GetIntegerSequence() ?? Enumerable.Empty<int>();

        foreach (var v in result)

        {

            Console.WriteLine(v);

        }

# **Concat operator in LINQ**

**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 :**   
linq concat example

**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.

# **SequenceEqual Operator in LINQ**

**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

# **Quantifiers in LINQ**

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