**Основы LINQ**

**LINQ (Language-Integrated Query)** представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable (например, стандартные коллекции, массивы), набор данных DataSet, документ XML. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

**Пример 1.** Выберем из массива строки, начинающиеся на определенную букву и отсортируем полученный список:

static void Main(string[] args)

{

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

var selectedTeams = new List<string>();

foreach (string s in teams)

{

if (s.ToUpper().StartsWith("Б"))

selectedTeams.Add(s);

}

selectedTeams.Sort();

foreach (string s in selectedTeams)

Console.WriteLine(s);

}

**Пример 2.** Вариант с LINQ:

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);

**Пример 3.** Вариант LINQ с методами

var selectedTeams2 = teams.Where(t => t.ToUpper().StartsWith("Б")).OrderBy(t => t);

foreach (string s in selectedTeams2)

Console.WriteLine(s);

**Пример 4.** Использование интерфейса IEnumerable<…>

string[] names = { "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland", "Clinton", "Coolidge", "Eisenhower", "Fillmore", "Ford", "Garfield", "Grant", "Harding", "Harrison", "Hayes", "Hoover", "Jackson",

"Jefferson", "Johnson", "Kennedy", "Lincoln", "Madison", "McKinley",

"Monroe", "Nixon", "Obama", "Pierce", "Polk", "Reagan", "Roosevelt",

"Taft", "Taylor", "Truman", "Tyler", "Van Buren", "Washington", "Wilson"};

// Использование синтаксиса выражения запроса

IEnumerable<string> sequence1 = from n in names

where n.Length < 6

select n;

foreach (string name in sequence)

{

Console.WriteLine("{0}", name);

}

// Использование точечной нотации

IEnumerable<string> sequence = names.Where(n => n.Length < 6).Select(n => n);

**Пример 5.**

string[] numbers = { "40", "2012", "176", "5" };

// Преобразуем массив строк в массив типа int используя LINQ

int[] nums = numbers.Select(s => Int32.Parse(s)).ToArray();

foreach (int n in nums)

Console.Write(n + " ");

**Пример 6.**

string[] numbers = { "40", "2012", "176", "5" };

// Преобразуем массив строк в массив типа int и сортируем по возрастанию используя LINQ

int[] nums2 = numbers.Select(s => Int32.Parse(s)).OrderBy(s => s).ToArray();

foreach (int n in nums2)

Console.Write(n + " ");

**Пример 7.**

Dictionary<int, string> obj = new Dictionary<int, string>();

obj[5] = "11111";

obj[2] = "55555";

obj[3] = "33333";

var obj2 = from a in obj

where a.Key>3

select a;

foreach (KeyValuePair<int, string> p in obj2)

{

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

}

**Пример 8.** Отложенный запрос, где результат не кэшируется и может изменяться от одного перечисления к другому

int[] intArray = new int[] { 1, 2, 3 };

IEnumerable<int> ints = intArray.Select(i => i);

foreach (int i in ints)

Console.WriteLine(i);

// Изменить элемент, в источнике данных

intArray[0] = 5;

foreach (int i in ints)

Console.WriteLine(i);

**Пример 9.** Выборка сложных объектов при помощи метода фильтрации

class User

{

public string Name { get;set; }

public int Age { get; set; }

public List<string> Languages { get; set; }

public User()

{

Languages = new List<string>();

}

}

List<User> users = new List<User>

{

new User {Name="Том", Age=23, Languages = new List<string> {"английский", "немецкий" }},

new User {Name="Боб", Age=27, Languages = new List<string> {"английский", "французский" }},

new User {Name="Джон", Age=29, Languages = new List<string> {"английский", "испанский" }},

new User {Name="Элис", Age=24, Languages = new List<string> {"испанский", "немецкий" }}

};

var selectedUsers = from user in users

where user.Age > 25

select user;

foreach (User user in selectedUsers)

или при помощи метода фильтрации **Where**

var selectedUsers = users.Where(u => u.Age > 25);

**Сложные фильтры**

Теперь рассмотрим более сложные фильтры. Например, в классе пользователя есть список языков, которыми владеет пользователь. Что если нам надо отфильтровать пользователей по языку:

**Пример 10.** Фильтрация пользователей по языку

var selectedUsers = from user in users

from lang in user.Languages

where user.Age < 28

where lang == "немецкий"

select user;

Для создания аналогичного запроса с помощью методов расширения применяется метод **SelectMany**:

var selectedUsers = users.SelectMany(u => u.Languages, (u, l) => new { User = u, Lang = l }).Where(u => u.Lang == "немецкий" && u.User.Age < 28).Select(u => u.User);

Метод **SelectMany()** в качестве первого параметра принимает последовательность, которую надо проецировать, а в качестве второго параметра - функцию преобразования, которая применяется к каждому элементу. На выходе она возвращает 8 пар "пользователь - язык" (new { User = u, Lang = l }), к которым потом применяетс фильтр с помощью **Where**.

**Проекция**

Проекция позволяет спроектировать из текущего типа выборки какой-то другой тип. Для проекции используется оператор **select**. Допустим, у нас есть набор объектов следующего класса, представляющего пользователя:

class User

{

public string Name { get; set; }

public int Age { get; set; }

}

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;

Console.WriteLine(names.GetType());

foreach (string n in names)

Console.WriteLine(n);

Результат выражения LINQ будет представлять набор строк, поскольку выражение select u.Name выбирают в результирующую выборку только значения свойства Name.

Аналогично можно создать объекты другого типа, в том числе анонимного:

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

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

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

var items = from u in users

select new

{

FirstName = u.Name,

DateOfBirth = DateTime.Now.Year - u.Age

};

Console.WriteLine(items.GetType());

foreach (var n in items)

Console.WriteLine("{0} - {1}", n.FirstName, n.DateOfBirth);

Здесь оператор select создает объект анонимного типа, используя текущий объект User. И теперь результат будет содержать набор объектов данного анонимного типа, в котором определены два свойства: FirstName и DateOfBirth.

В качестве альтернативы мы могли бы использовать метод расширения **Select():**

// выборка объектов анонимного типа

var items = users.Select(u => new

{

FirstName = u.Name,

DateOfBirth = DateTime.Now.Year - u.Age

});

Переменные в запросах и оператор **let**

Иногда возникает необходимость произвести в запросах **LINQ** какие-то дополнительные промежуточные вычисления. Для этих целей мы можем задать в запросах свои переменные с помощью оператора **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

};

**Выборка из нескольких источников**

В LINQ можно выбирать объекты не только из одного, но и из большего количества источников:

**Пример 11**

class Phone

{

public string Name { get; set; }

public string Company { get; set; }

}

class User

{

public string Name { get; set; }

public int Age { get; set; }

}

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"},

};

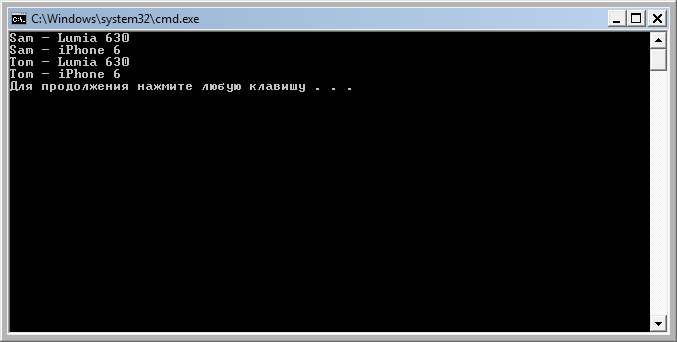
var people = from user in users

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);



**Сортировка**

Для сортировки набора данных по возрастанию используется оператор **orderby**. Оператор **orderby** принимает критерий сортировки. По умолчанию оператор **orderby** производит сортировку по возрастанию.

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

var orderedNumbers = from i in numbers

orderby i

select i;

Допустим, надо отсортировать выборку сложных объектов. Тогда в качестве критерия мы можем указать свойство класса объекта:

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

{

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

new User { Name = "Bob", Age = 30 },

new User { Name = "Tom", Age = 21 },

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

};

var sortedUsers = from u in users

orderby u.Name

select u;

Сортировка по убыванию с помощью ключевого слова и **descending**

var sortedUsers = from u in users

orderby u.Name descending

select u;

Вместо оператора **orderby** можно использовать методы расширения **OrderBy (сортирование по возрастанию)**:

var sortedUsers = users.OrderBy(u => u.Name);

или сортировка по убыванию **OrderByDescending**

var sortedUsers = users.OrderByDescending(u => u.Name);

**Множественные критерии сортировки**

В наборах сложных объектов иногда встает ситуация, когда надо отсортировать не по одному, а сразу по нескольким полям. Для этого в запросе **LINQ** все критерии указываются в порядке приоритета через запятую:

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

{

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

new User { Name = "Bob", Age = 30 },

new User { Name = "Tom", Age = 21 },

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

new User { Name = "Alice", Age = 28 },

};

var result = from user in users

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

select user;

С помощью методов расширения то же самое можно сделать через метод **ThenBy()**(для сортировки по возрастанию) и **ThenByDescending()** (для сортировки по убыванию):

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

**Работа с множествами**

Кроме методов выборки **LINQ** имеет несколько методов для работы с множествами: разность, объединение и пересечение.

**Разность множеств**

С помощью метода **Except** можно получить разность двух множеств:

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

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

// разность множеств

var result = soft.Except(hard);

foreach (string s in result)

Console.WriteLine(s);

В данном случае из массива **soft** убираются все элементы, которые есть в массиве **hard**.

**Пересечение множеств**

Для получения пересечения множеств, то есть общих для обоих наборов элементов, применяется метод **Intersect**:

// пересечение множеств

var result = soft.Intersect(hard);

Так как оба набора имеют только один общий элемент, то соответственно только он и попадет в результирующую выборку - **Apple**

**Объединение множеств**

Для объединения двух множеств используется метод **Union**. Его результатом является новый набор, в котором имеются элементы, как из одного, так и из второго множества. Повторяющиеся элементы добавляются в результат только один раз:

// объединение множеств

var result = soft.Union(hard);

Объединение двух наборов, то мы можем использовать метод **Concat**:

var result = soft.Concat(hard);

string[] str = result.ToArray<string>();

**Удаление дубликатов**

Для удаления дублей в наборе используется метод **Distinct**:

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

**Агрегатные операции**

К агрегатным операциям относят различные операции над выборкой, например, получение числа элементов, получение минимального, максимального и среднего значения в выборке, а также суммирование значений.

**Метод** **Aggregate**

Метод **Aggregate** выполняет общую агрегацию элементов коллекции в зависимости от указанного выражения.

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

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

Переменная query будет представлять результат агрегации массива. В качестве условия агрегации используется выражение (x,y)=> x - y, то есть вначале из первого элемента вычитается второй, потом из получившегося значения вычитается третий и так далее.

**Получение размера выборки. Метод Count**

Для получения числа элементов в выборке используется метод **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);

Метод **Count()** в одной из версий также может принимать лямбда-выражение, которое устанавливает условие выборки. Поэтому мы можем в данном случае не использовать выражение **Where**:

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

**Получение суммы**

Для получения суммы значений применяется метод **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);

**Максимальное, минимальное и среднее значения**

Для нахождения минимального значения применяется метод **Min()**, для получения максимального - метод **Max()**, а для нахождения среднего значения - метод **Average().** Их действие похоже на методы **Sum** и **Count**:

int min1 = numbers.Min();

int min2 = users.Min(n => n.Age); // минимальный возраст

int max1 = numbers.Max();

int max2 = users.Max(n => n.Age); // максимальный возраст

double avr1 = numbers.Average();

double avr2 = users.Average(n => n.Age); //средний возраст

**Методы Skip и Take**

Метод **Skip()** пропускает определенное количество элементов, а метод **Take()** извлекает определенное число элементов. Нередко данные методы применяются вместе для создания постраничного вывода.

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

var result = numbers.Take(3);

foreach (int i in result)

Console.WriteLine(i);

var result2 = numbers.Skip(3);

foreach (int i in result2)

Console.WriteLine(i);

var result3 = numbers.Skip(3).Take(3);

foreach (int i in result3)

Console.WriteLine(i);

Похожим образом работают методы **TakeWhile** и **SkipWhil**e.

Метод **TakeWhile** выбирает цепочку элементов, начиная с первого элемента, пока они удовлетворяют определенному условию

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

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

Console.WriteLine(t);

**SkipWhile** пропускает цепочку элементов, начиная с первого элемента, пока они удовлетворяют определенному условию.

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

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

Console.WriteLine(t);

**Группировка**

Для группировки данных по определенным параметрам применяется оператор **group by** или метод **GroupBy(**). Если в выражении **LINQ** последним оператором, выполняющим операции над выборкой, является **group**, то оператор **select** не применяется.

Допустим, у нас есть набор из объектов следующего типа:

class Phone

{

public string Name { get; set; }

public string Company { get; set; }

}

List<Phone> phones = new List<Phone>

{

new Phone {Name="Lumia 430", Company="Microsoft" },

new Phone {Name="Mi 5", Company="Xiaomi" },

new Phone {Name="LG G 3", Company="LG" },

new Phone {Name="iPhone 5", Company="Apple" },

new Phone {Name="Lumia 930", Company="Microsoft" },

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

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

new Phone {Name="LG G 4", Company="LG" }

};

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();

}

Оператор **group** принимает критерий по которому проводится группировка: **group phone by phone.Company** - в данном случае группировка по свойству **Company**.

Результатом оператора group является выборка, которая состоит из групп. Каждая группа представляет объект **IGrouping<string, Phone>.**

Аналогичный запрос можно построить с помощью метода расширения **GroupBy:**

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

**Создание из группы нового объекта**

var phoneGroups2 = from phone in phones

group phone by phone.Company into g

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

foreach (var group in phoneGroups2)

Console.WriteLine("{0} : {1}", group.Name, group.Count);

Аналогичная операция с помощью метода **GroupBy():**

var phoneGroups2 = phones.GroupBy(p => p.Company).Select(g => new { Name = g.Key, Count = g.Count() });

**Вложенные запросы:**

var phoneGroups2 = from phone in phones

group phone by phone.Company into g

select new

{

Name = g.Key,

Count = g.Count(),

Phones = from p in g select p

};

foreach (var group in phoneGroups2)

{

Console.WriteLine("{0} : {1}", group.Name, group.Count);

foreach (Phone phone in group.Phones)

Console.WriteLine(phone.Name);

Console.WriteLine();

}

Аналогичный запрос с помощью метода **GroupBy:**

var phoneGroups2 = phones.GroupBy(p => p.Company).Select(g => new

{

Name = g.Key,

Count = g.Count(),

Phones = g.Select(p => p)

});