**№ 11 LINQ to Object**

**Задания**

1. Задайте массив типа string, содержащий 12 месяцев (June, July, May, December, January ….). Используя LINQ to Object напишите запрос выбирающий последовательность месяцев с длиной строки равной n, запрос возвращающий только летние и зимние месяцы, запрос вывода месяцев в алфавитном порядке, запрос считающий месяцы содержащие букву «u» и длиной имени не менее 4-х..

2. Создайте коллекцию List<T> и параметризируйте ее типом (классом) из лабораторной №3 (при необходимости реализуйте нужные интерфейсы). Заполните ее минимум 8 элементами.

Если в задании указано свойство, которым ваш класс не обладает, то его нужно расширить, чтобы класс соответствовал условию. Один из запросов реализуйте используя язык LINQ и используя методы расширения LINQ.

3. На основе LINQ сформируйте следующие запросы по вариантам. При необходимости добавьте в класс T (тип параметра) свойства.

4. Придумайте и напишите свой собственный запрос, в котором было бы не менее 5 операторов из разных категорий: условия, проекций, упорядочивания, группировки, агрегирования, кванторов и разбиения.

5. Придумайте запрос с оператором Join

|  |  |
| --- | --- |
| **Вариант 8** | *сведения об абонентах, у которых время внутригородских разговоров превышает заданное;*  *сведения об абонентах, которые пользовались междугородной связью;*  *количество абонентов с заданным значением дебета*  *максимального абонента (по вашему критерию)*  *упорядоченный список абонентов по фамилии* |

**► LINQ**

**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.

Существует несколько разновидностей LINQ:

* **LINQ to Objects**: применяется для работы с массивами и коллекциями
* **LINQ to Entities**: используется при обращении к базам данных через технологию Entity Framework
* **LINQ to Sql**: технология доступа к данным в MS SQL Server
* **LINQ to XML**: применяется при работе с файлами XML
* **LINQ to DataSet**: применяется при работе с объектом DataSet
* **Parallel LINQ (PLINQ)**: используется для выполнения параллельных запросов

В этой главе речь пойдет прежде всего о **LINQ to Objects**.

В чем же удобство LINQ? Посмотрим на простейшем примере. Выберем из массива строки, начинающиеся на определенную букву, и отсортируем полученный список:

using System;

using System.Collections.Generic;

namespace Laba11

{

class Program

{

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();

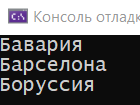
foreach (string s in selectedTeams)

Console.WriteLine(s);

}

}

}



Теперь проведем те же действия с помощью 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);

Чтобы использовать функциональность LINQ, убедимся, что в файле подключено пространство имен **System.Linq**.

Итак, код стал меньше и проще. В принципе все выражение можно было бы записать в одну строку: var selectedTeams = from t in teams where t.ToUpper().StartsWith("Б") orderby t select t. Но для более понятной логической разбивки я поместил каждое отдельное подвыражение на отдельной строке.

Простейшее определение запроса LINQ выглядит следующим образом:

***from переменная in набор\_объектов***

***select переменная;***

Итак, что делает этот запрос LINQ? Выражение from t in teams проходит по всем элементам массива teams и определяет каждый элемент как t. Используя переменную t мы можем проводить над ней разные операции.

Несмотря на то, что мы не указываем тип переменной t, выражения LINQ являются строго типизированными. То есть среда автоматически распознает, что набор teams состоит из объектов string, поэтому переменная t будет рассматриваться в качестве строки.

Далее с помощью оператора where проводится фильтрация объектов, и если объект соответствует критерию (в данном случае начальная буква должна быть "Б"), то этот объект передается дальше.

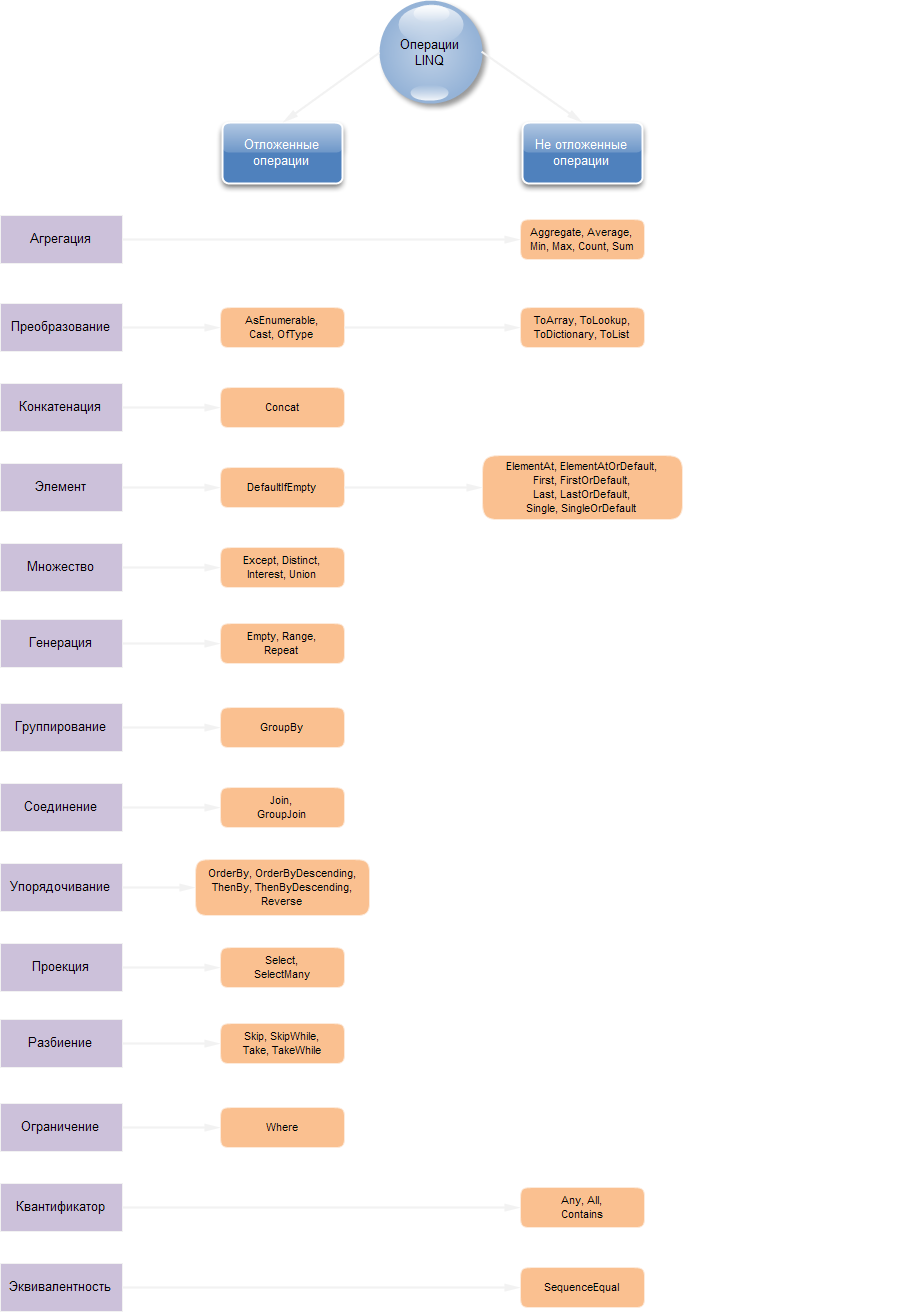
Оператор orderby упорядочивает по возрастанию, то есть сортирует выбранные объекты.

Оператор select передает выбранные значения в результирующую выборку, которая возвращается LINQ-выражением.

В данном случае результатом выражения LINQ является объект IEnumerable<T>. Нередко результирующая выборка определяется с помощью ключевого слова var, тогда компилятор на этапе компиляции сам выводит тип.

Преимуществом подобных запросов также является и то, что они интуитивно похожи на запросы языка SQL, хотя и имеют некоторые отличия.

LINQ включает в себя около 50 стандартных операций запросов, разделяемых на 2 большие группы - отложенные операции (выполняются не во время инициализации, а только при их вызове) и неотложенные операции (выполняются сразу). На рисунке ниже наглядно показана "градация" операций LINQ:



По большей части LINQ ориентирован на запросы — будь то запросы, возвращающие набор подходящих объектов, единственный объект или подмножество полей из объекта либо набора объектов. В LINQ этот возвращенный набор называется последовательностью (sequence). Большинство последовательностей LINQ имеют тип IEnumerable<T>, где T — тип данных объектов, находящихся в последовательности. Например, если есть последовательность целых чисел, они должны храниться в переменной типа IEnumerable<int>. Вы увидите, что IEnumerable<T> буквально господствует в LINQ и очень многие методы LINQ возвращают IEnumerable<T>.

Может показаться, что LINQ — это нечто, связанное только с запросами, поскольку расшифровывается как язык интегрированных запросов (Language Integrated Query). Однако не думайте о нем лишь в этом контексте. Предпочтительнее воспринимать LINQ как механизм итерации данных (data iteration engine), но возможно в Microsoft не захотели обозначать эту технологию аббревиатурой DIE ("умереть").

Приходилось ли вам когда-нибудь вызывать метод, возвращающий данные в структуре, которую затем приходилось преобразовывать в еще одну структуру данных, прежде чем передать другому методу? Предположим, например, что вызывается метод А, и этот метод возвращает массив типа string, содержащий числовые значения в виде строк. Затем нужно вызвать метод В, но метод В требует массива целых чисел. Обычно приходится организовывать цикл для прохода по массиву строк и наполнения вновь сконструированного массива целых чисел. Давайте рассмотрим краткий пример мощи Microsoft LINQ.

Предположим, что имеется массив строк, которые приняты от метода А, как показано ниже:

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 + " ");



### Методы расширения LINQ

Кроме стандартного синтаксиса from .. in .. select для создания запроса LINQ мы можем применять специальные методы расширения, которые определены для интерфейса IEnumerable. Как правило, эти методы реализуют ту же функциональность, что и операторы LINQ типа where или orderby.

Например:

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);

Запрос teams.Where(t=>t.ToUpper().StartsWith("Б")).OrderBy(t => t) будет аналогичен предыдущему. Он состоит из цепочки методов Where и OrderBy. В качестве аргумента эти методы принимают делегат или лямбда-выражение.

Не каждый метод расширения имеет аналог среди операторов LINQ, но в этом случае можно сочетать оба подхода. Например, используем стандартный синтаксис linq и метод расширения Count(), возвращающий количество элементов в выборке:

int number = (from t in teams where t.ToUpper().StartsWith("Б") select t).Count();

Запрос с использованием методов расширения называется **точечной нотацией**.

### 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 удовлетворяют 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 создаются только в одной коллекции
* **Union**: объединяет две однородные коллекции
* **Intersect**: возвращает пересечение двух коллекций, то есть те элементы, которые встречаются в обеих 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 содержит 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

## ► Фильтрация выборки и проекция

### Фильтрация

Для выбора элементов из некоторого набора по условию используется метод **Where**. Например, выберем все четные элементы, которые больше 10.

Фильтрация с помощью операторов LINQ:

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

var evens = from i in numbers

where i % 2 == 0 && i > 10

select i;

foreach(int i in evens)

{

Console.WriteLine(i);

}

Здесь используется конструкция from: from i in numbers

Тот же запрос с помощью метода расширения:

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

var evens = numbers.Where(i => i % 2 == 0 && i > 10);

Если выражение в методе Where для определенного элемента будет равно true (в данном случае выражение i % 2 == 0 && i > 10), то данный элемент попадает в результирующую выборку.

### Выборка сложных объектов

Допустим, у нас есть класс пользователя:

class User

{

public string Name { get; set; }

public int Age { get; set; }

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

public User()

{

Languages = new List<string>();

}

}

Создадим набор пользователей и выберем из них тех, которым больше 25 лет:

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)

Console.WriteLine($"{user.Name} - {user.Age}");

Консольный вывод:

Боб - 27

Джон - 29

Аналогичный запрос с помощью метода расширения 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:

var selectedUsers = from user in users

from lang in user.Languages

where user.Age < 28

where lang == "английский"

select user;

Результат:

Том - 23

Боб - 27

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 == "английский" && 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 применяется фильтр с помощью Where.

### Проекция

##### **Что такое проекция?**

Проекция - это не что иное, как механизм, который используется для выбора данных из источника данных. Вы можете выбрать данные в той же форме (т.е. исходные данные в исходном состоянии). Также возможно создать новую форму данных, выполнив над ней некоторые операции.

В проецировании доступны два метода. Они следующие:

1. Select
2. SelectMany

Оператор LINQ Select позволяет нам указать, какие свойства мы хотим получить, возможно все свойства или некоторые свойства, которые нужно указать в операторе выбора. Стандартный оператор LINQ Select также позволяет нам выполнять некоторые вычисления.

Допустим, у нас есть набор объектов следующего класса, представляющего пользователя:

class User

{

public string Name { get; set; }

public int Age { get; set; }

}

Но нам нужен не весь объект, а только его свойство Name:

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;

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

};

foreach (var n in items)

Console.WriteLine($"{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 имен

var names = users.Select(u => u.Name);

// 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

};

В данном случае создается переменная name, значение которой равно "Mr. " + u.Name.

Возможность определения переменных, наверное, одно из главных преимуществ операторов LINQ по сравнению с методами расширения.

### Выборка из нескольких источников

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

Например, возьмем классы:

class Phone

{

public string Name { get; set; }

public string Company { get; set; }

}

class User

{

public string Name { get; set; }

public int Age { get; set; }

}

Создадим два разных источника данных и произведем выборку:

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($"{p.Name} - {p.Phone}");

Консольный вывод:

Sam - Lumia 630

Sam - iPhone 6

Tom - Lumia 630

Tom - iPhone 6

Таким образом, при выборке из двух источников каждый элемент из первого источника будет сопоставляться с каждым элементом из второго источника. То есть получиться 4 пары.

## ► Сортировка

Для сортировки набора данных по возрастанию используется оператор **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);

Оператор orderby принимает критерий сортировки. В данном случае в качестве критерия выступает само число.

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

using System;

using System.Linq;

using System;

using System.Collections.Generic;

using System.Linq;

namespace pract

{

class Phone

{

public string Name { get; set; }

public string Company { get; set; }

}

class User

{

public string Name { get; set; }

public int Age { get; set; }

}

class Program

{

static void Main(string[] args)

{

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;

foreach (User u in sortedUsers)

Console.WriteLine(u.Name);

}

}

}



По умолчанию оператор orderby производит сортировку по возрастанию. Однако с помощью ключевых слов **ascending** (сортировка по возрастанию) и **descending** (сортировка по убыванию) можно явным образом указать направление сортировки:

var sortedUsers = from u in users

orderby u.Name descending

select u;

Вместо оператора orderby можно использовать методы расширения **OrderBy**:

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

IEnumerable<int> sortedNumbers = numbers.OrderBy(i => i);

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

Метод OrderBy()сортирует по возрастанию. Для сортировки по убыванию используется метод:

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 }

};

var result = from user in users

orderby user.Name, user.Age

select user;

foreach (User u in result)

Console.WriteLine($"{u.Name} - {u.Age}");

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);

Результат будет аналогичен предыдущему.

## ► Объединение, пересечение и разность коллекций

Кроме методов выборки LINQ имеет несколько методов, который позволяют из двух последовательностей объектов сгенерировать множество (то есть некий набор уникальных элементов): разность, объединение и пересечение.

### Разность последовательностей

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

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

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

// разность последовательностей

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. Результатом операции будут два элемента:

Microsoft

Google

### Пересечение последовательностей

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

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

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

// пересечение последовательностей

var result = soft.Intersect(hard);

foreach (string s in result)

Console.WriteLine(s);

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

Apple

### Объединение последовательностей

Для объединения двух последовательностей используется метод **Union**. Его результатом является новый набор, в котором имеются элементы, как из первой, так и из второй последовательности. Повторяющиеся элементы добавляются в результат только один раз:

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

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

// объединение последовательностей

var result = soft.Union(hard);

foreach (string s in result)

Console.WriteLine(s);

Если же нам нужно простое объединение двух наборов, то мы можем использовать метод **Concat**:

|  |
| --- |
| var result = soft.Concat(hard); |

Те элементы, которые встречаются в обоих наборах, дублируются.

### Удаление дубликатов

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

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

Последовательное применение методов Concat и Distinct будет подобно действию метода Union.

## ► 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. Например:

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. То есть будет эквивалентно выражению:

|  |
| --- |
| int query = 1 - 2 - 3 - 4 - 5 |

В итоге мы получим число -13. Соответственно мы бы могли использовать любые другие операции, например, сложение:

int query = numbers.Aggregate((x,y)=> x + y); // аналогично 1 + 2 + 3 + 4 + 5

### 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);

Console.WriteLine(size);

### 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);

Метод Sum() имеет ряд перегрузок. В частности, если у нас набор сложных объектов, как в примере выше, то мы можем указать свойство, значения которого будут суммироваться: 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[] 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 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

**Операции First, FirstOrDefault, Last и LastOrDefault**

Операции элементов позволяют извлекать отдельные элементы из входной последовательности.

**First**

Операция First возвращает первый элемент последовательности или первый элемент последовательности, соответствующий предикату — в зависимости от использованного прототипа.

Эта операция имеет два прототипа, описанные ниже:

Первый прототип First

*public static T First<T>(*

*this IEnumerable<T> source);*

При использовании этого прототипа операции First выполняется перечисление входной последовательности source и возвращается ее первый элемент.

Второй прототип First

Второй прототип операции First позволяет передать ему предикат:

*public static T First<T>(*

*this IEnumerable<T> source,*

*Func<T, bool> predicate);*

Эта версия операции First возвращает первый найденный ею элемент, для которого predicate дал true. Если ни один из элементов не заставил predicate вернуть true, то операция First генерирует исключение InvalidOperationException.

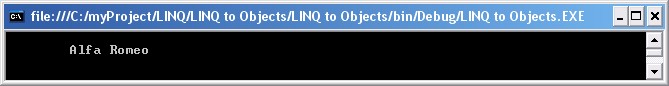
Пример первого прототипа First приведен ниже:

string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW",

"Ferrari", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

string auto = cars.First();

Console.WriteLine("\n\t" + auto);



Может возникнуть вопрос: чем отличается эта операция от вызова операции Take с параметром 1? Отличие в том, что Take возвращает последовательность элементов, даже если она состоит всего из одного элемента. Операция First всегда возвращает в точности один элемент либо генерирует исключение, если возвращать нечего.

Ниже приведен пример использования второго прототипа операции First:

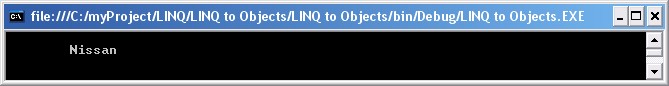
string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW",

"Ferrari", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

string auto = cars.First(p => p.StartsWith("N"));

Console.WriteLine("\n\t" + auto);

Он должен вернуть первый элемент из входной последовательности, начинающийся со строки "N". Вот результат:



Помните, что если любой из прототипов операции First не находит элемент, который нужно возвратить, **генерируется исключение InvalidOperationException**. Чтобы избежать этого, используйте операцию FirstOrDefault.

**FirstOrDefault**

Операция FirstOrDefault подобна First во всем, кроме поведения, когда элемент не найден. Эта операция имеет два прототипа, описанные ниже:

Первый прототип FirstOrDefault

*public static T FirstOrDefault<T> (*

*this IEnumerable<T> source);*

Эта версия прототипа FirstOrDefault возвращает первый элемент, найденный во входной последовательности. Если последовательность пуста, возвращается default(Т). Для ссылочных и допускающих null типов значением по умолчанию является null.

Второй прототип FirstOrDefault

Второй прототип операции FirstOrDefault позволяет передать predicate, определяющий, какой элемент следует возвратить.

*public static T FirstOrDefault<T>(*

*this IEnumerable<T> source,*

*Func<T, bool> predicate);*

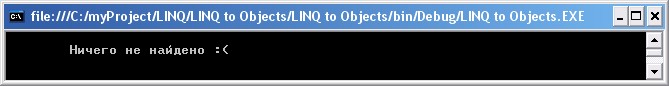
Код ниже, содержит пример использования первого прототипа FirstOrDefault, где элемент не найден. Для этого понадобилась пустая последовательность, которая была создана с помощью Take(0):

string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW",

"Ferrari", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

string auto = cars.Take(0).FirstOrDefault();

Console.WriteLine(auto == null ? "\n\tНичего не найдено :(" : auto);



**Last**

Операция Last возвращает последний элемент последовательности или последний элемент, соответствующий предикату — в зависимости от используемого прототипа. Эта операция имеет два прототипа, которые описаны ниже:

Первый прототип Last

*public static T Last<T>(*

*this IEnumerable<T> source);*

В случае этого прототипа операция Last перечисляет входную последовательность по имени source и возвращает ее последний элемент.

Второй прототип Last

Второй прототип Last позволяет передать predicate и рассматривается ниже:

*public static T Last<T>(*

*this IEnumerable<T> source,*

*Func<T, bool> predicate);*

Эта версия операции Last возвращает последний из найденных элементов, для которых predicate вернет true. Ниже показан пример использования этих прототипов:

string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW",

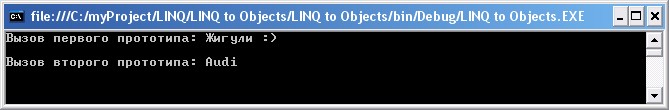
"Ferrari", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

string auto = cars.Last();

Console.WriteLine("Вызов первого прототипа: " + auto);

string auto1 = cars.Last(p => p.StartsWith("A"));

Console.WriteLine("\nВызов второго прототипа: " + auto1);



Помните, что если любому из двух прототипов операции Last возвращать нечего, генерируется исключение InvalidOperationException. Чтобы избежать этого, используйте операцию LastOrDefault.

**LastOrDefault**

Операция LastOrDefault подобна Last во всем, за исключением поведения в случае, когда элемент не найден. Эта операция имеет два прототипа, анологичные операции Last.

Ниже показан пример использования обоих прототипов LastOrDefault:

string[] cars = { "Alfa Romeo", "Aston Martin", "Audi", "Nissan", "Chevrolet", "Chrysler", "Dodge", "BMW",

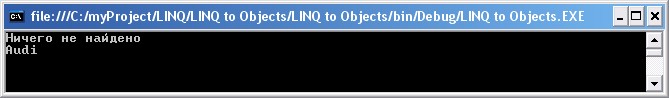
"Ferrari", "Bentley", "Ford", "Lexus", "Mercedes", "Toyota", "Volvo", "Subaru", "Жигули :)"};

string auto = cars.Take(0).LastOrDefault();

Console.WriteLine(auto == null ? "Ничего не найдено" : auto);

string auto1 = cars.LastOrDefault(p => p.StartsWith("A"));

Console.WriteLine(auto1 == null ? "Ничего не найдено" : auto1);



## 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.

Извлечем три первых элемента:

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

var result = numbers.Take(3);

foreach (int i in result)

Console.WriteLine(i);

Выберем все элементы, кроме первых трех:

var result = numbers.Skip(3);

Совмещая оба метода, мы можем выбрать определенное количество элементов начиная с определенного элемента. Например, выберем три элемента, начиная с пятого (то есть пропустив четыре элемента):

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

var result = numbers.Skip(4).Take(3);

foreach (int i in result)

Console.WriteLine(i);

Похожим образом работают методы **TakeWhile** и **SkipWhile**.

Метод TakeWhile выбирает цепочку элементов, начиная с первого элемента, пока они удовлетворяют определенному условию. Например:

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

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

Console.WriteLine(t);

Согласно условию мы выбираем те команды, которые начинаются с буквы Б. В массиве есть три таких команды. Однако в цикле будут выведены только две первых:

Бавария

Боруссия

Потому что цепочка обрывается на третьей команде - "Реал Мадрид" - она не соответствует условию, и после этого выборка уже не идет.

Если бы первой командой в массиве стояла бы команда, начинающаяся не с буквы Б, например, "Реал Мадрид", то в этом случае метод возвратил бы нам 0 элементов.

В подобном русле действует метод SkipWhile. Он пропускает цепочку элементов, начиная с первого элемента, пока они удовлетворяют определенному условию. Например:

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

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

Console.WriteLine(t);

Первые две команды, которые начинаются с буквы Б и соответствуют условию, будут пропущены. На третьей команде цепочка обрывается, поэтому последняя команда, начинающаяся с буквы Б, будет включена в выходной список:

Реал Мадрид

Манчестер Сити

ПСЖ

Барселона

И опять же если в массиве первый элемент не начинался бы с буквы Б, то цепочка пропускаемых элементов прервалась бы уже на первом элементе, и поэтому метод SkipWhile возвратил бы все элементы массива.

## ► Группировка

Для группировки данных по определенным параметрам применяется оператор **group by** или метод **GroupBy()**. Допустим, у нас есть набор из объектов следующего типа:

class Phone

{

public string Name { get; set; }

public string Company { get; set; }

}

Данный класс представляет модель телефона, в которой определены свойства для названия и компании-производителя. Сгруппируем набор телефонов по производителю:

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();

}

Если в выражении LINQ последним оператором, выполняющим операции над выборкой, является group, то оператор select не применяется.

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>: параметр string указывает на тип ключа, а параметр Phone - на тип сгруппированных объектов.

Каждая группа имеет ключ, который мы можем получить через свойство Key: g.Key

Все элементы группы можно получить с помощью дополнительной итерации. Элементы группы имеют тот же тип, что и тип объектов, которые передавались оператору group, то есть в данном случае объекты типа Phone.

В итоге мы получим следующий вывод:

Microsoft

Lumia 430

Lumia 930

Lumia 630

Xiaomi

Mi 5

LG

LG G 3

LG G4

Apple

iPhone 5

iPhone 6

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

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

Теперь изменим запрос, сгруппируем объекты в переменную *g* и создадим из этой группы новый объект:

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($"{group.Name} : {group.Count}");

Выражение group phone by phone.Company into g определяет переменную g, которая будет содержать группу. С помощью этой переменной мы можем затем создать новый объект анонимного типа: select new { Name = g.Key, Count = g.Count() } Теперь результат запроса LINQ будет представлять набор объектов таких анонимных типов, у которых два свойства Name и Count.

Результат программы:

Microsoft : 3

Xiaomi : 1

LG : 2

Apple : 2

Аналогичная операция с помощью метода GroupBy():

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

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

Также мы можем осуществлять 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($"{group.Name} : {group.Count}");

foreach (Phone phone in group.Phones)

Console.WriteLine(phone.Name);

Console.WriteLine();

}

Здесь свойство Phones каждой группы формируется с помощью дополнительного запроса, выбирающего все телефоны в этой группе. Консольный вывод программы:

Microsoft : 3

Lumia 430

Lumia 930

Lumia 630

Xiaomi : 1

Mi 5

LG : 2

LG G 3

LG G4

Apple : 2

iPhone 5

iPhone 6

Аналогичный запрос с помощью метода GroupBy:

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

.Select(g => new

{

Name = g.Key,

Count = g.Count(),

Phones = g.Select(p => p)

});

## ► Соединение коллекций. Метод Join, GroupJoin и Zip

## Соединение в LINQ используется для объединения двух разнотипных наборов в один. Для соединения используется оператор join или метод Join(). Как правило, данная операция применяется к двум наборам, которые имеют один общий критерий. Например, у нас есть два класса:

class Player

{

public string Name { get; set; }

public string Team { get; set; }

}

class Team

{

public string Name { get; set; }

public string Country { get; set; }

}

Объекты обоих классов будет иметь один общий критерий - название команды. Соединим по этому критерию два набора этих классов:

List<Team> teams = new List<Team>()

{

new Team { Name = "Бавария", Country ="Германия" },

new Team { Name = "Барселона", Country ="Испания" }

};

List<Player> players = new List<Player>()

{

new Player {Name="Месси", Team="Барселона"},

new Player {Name="Неймар", Team="Барселона"},

new Player {Name="Роббен", Team="Бавария"}

};

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 };

foreach (var item in result)

Console.WriteLine($"{item.Name} - {item.Team} ({item.Country})");

С помощью выражения join t in teams on pl.Team equals t.Name объект pl из списка players соединяется с объектом t из списка teams, если значение свойства pl.Team совпадает со значением свойства t.Name. Результатом соединения будет объект анонимного типа, который будет содержать три свойства. В итоге мы получим следующий вывод:

Месси - Барселона (Испания)

Неймар - Барселона (Испания)

Роббен - Бавария (Германия)

То же самое действие можно было бы выполнить с помощью метода 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

Метод Join() принимает четыре параметра:

* второй список, который соединяем с текущим
* свойство объекта из текущего списка, по которому идет соединение
* свойство объекта из второго списка, по которому идет соединение
* новый объект, который получается в результате соединения

### GroupJoin

Метод GroupJoin кроме соединения последовательностей также выполняет и группировку. Например, возьмем вышеопределенные списки teams и players и сгуппируем всех игроков по командам:

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();

}

Результатом выполнения программы будет следующий вывод:

Бавария

Роббен

Барселона

Месси

Неймар

Метод GroupJoin, также как и метод Join, принимает все те же параметры. Только теперь во последний параметр - делегат передаются объект команды и набор игроков этой команды.

### Метод Zip

Метод Zip позволяет объединять две последовательности таким образом, что первый элемент из первой последовательности объединяется с первым элементом из второй последовательности, второй элемент из первой последовательности соединяется со вторым элементом из второй последовательности и так далее:

List<Team> teams = new List<Team>()

{

new Team { Name = "Бавария", Country ="Германия" },

new Team { Name = "Барселона", Country ="Испания" },

new Team { Name = "Ювентус", Country ="Италия" }

};

List<Player> players = new List<Player>()

{

new Player {Name="Роббен", Team="Бавария"},

new Player {Name="Неймар", Team="Барселона"},

new Player {Name="Буффон", Team="Ювентус"}

};

var result2 = players.Zip(teams,

(player, team) => new

{

Name = player.Name,

Team = team.Name,

Country = team.Country

});

foreach (var player in result2)

{

Console.WriteLine($"{player.Name} - {player.Team} ({player.Country})");

Console.WriteLine();

}

Метод Zip в качестве первого параметра принимает вторую последовательность, с которой надо соединяться, а в качестве второго параметра - делегат для создания нового объекта.

Консольный вывод программы:

Роббен - Бавария (Германия)

Неймар - Барселона (Испания)

Буффон - Ювентус (Италия)