

Итераторы и LINQ

Интерфейс IEnumerable и IEnumerator

Любая коллекция реализует интерфейс IEnumerable.

```
public interface IEnumerable<out T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

Объект, возвращаемый методом GetEnumerator(), называется итератор. Итератор реализует интерфейс IEnumerator<T>:

```
public interface IEnumerator<out T> : IDisposable, IEnumerator
{
    T Current { get; }
    bool MoveNext(); // false, когда дальше ехать некуда
    void Reset();
}
```

Замечание. Интерфейс IEnumerator<T> не связан с какой-то коллекцией.

Ключевое слово yield

Главным в интерфейсе `IEnumerator<T>` является метод `MoveNext()`, поэтому итератор можно объявить, запрограммировав одну эту функцию, остальное компилятор доделает сам.

Для этого в C# существует слово `yield`.

```
IEnumerator<int> M() {  
    yield return 1;  
    yield return 2;  
    yield return 3;  
}
```

```
for (var iter = M(); iter.MoveNext(); )  
    Console.WriteLine(iter.Current);
```

Более практично возвращать не `IEnumerator` а `IEnumerable`.

```
IEnumerable<int> M() {  
    yield return 1;  
    yield return 2;  
    yield return 3;  
}
```

```
foreach (var i in M())  
    Console.WriteLine(i);
```

Здесь компилятор реализует сразу два интерфейса – `IEnumerator` и `IEnumerable`.

Примеры итераторов

Итератор может сам вырабатывать последовательность значений.

```
static IEnumerable<int> Range(int a, int b)
{
    for (int i = a; i < b; i++)
        yield return i;
}
```

Итератор может основываться на входной последовательности.

```
static IEnumerable<int> Filter(IEnumerable<int> it)
{
    foreach (var x in it)
        if (x % 2 == 0)
            yield return x;
}
```

...или на нескольких входных последовательностях.

Задание 0

1. Реализовать итератор, который позволит проходить любой целый массив в обратном порядке.
2. Реализовать обобщенный итератор, который позволит проходить любой массив в обратном порядке.
3. Сделать обобщенный итератор, который обходит двумерный массив по столбцам.
4. Объявить итератор, который получает один целочисленный массив в качестве аргумента и вырабатывает все уникальные значения из входного массива.
 $\{ 2, 3, 4, 3, 3, 1, 2 \} \Rightarrow \{ 2, 3, 4, 1 \}$
5. Объявить итератор, который получает два целочисленных массива в качестве аргументов и вырабатывает только те значения, которые есть в первом массиве и в то же время отсутствуют во втором массиве. $\{ 1, 2, 3, 5, 6 \}, \{ 4, 5, 1 \} \Rightarrow \{ 2, 3, 6 \}$

Что такое LINQ to Objects

Итератор `IEnumerable<T>` можно преобразовать в другой итератор, в число, в объект, в массив, в коллекцию, во что угодно.

Набор готовых методов-преобразователей, который содержится в статическом классе `System.Linq.Enumerable`, и составляет основное содержание LINQ to Objects.

Большинство преобразователей имеют форму методов, расширяющих тип `IEnumerable<T>`, например

```
public static int Count<TSource>(this IEnumerable<TSource> source);
```

Это нужно, чтобы была возможность составлять цепочечные выражения.

```
int n = Enumerable.Range(1, 10).Where(x => x % 2 == 0).Count();
```

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

Выражения запросов

Верхний уровень LINQ составляют выражения запросов.

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

class Program
{
    static void Main() {

        string[] names = { "Burke", "Connor", "Frank",
            "Everett", "Albert", "George", "Harris", "David" };

        var query = from name in names
                    where name.Length == 5
                    orderby name
                    select name.ToUpper();

        foreach (string item in query)
            Console.WriteLine(item);
    }
}
```

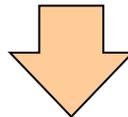
```
BURKE
DAVID
FRANK
```

Компиляция выражений

Компилятор переводит выражения запроса в цепочку вызовов расширяющих методов.

Аргументами методов являются лямбда-выражения.

```
var query = from name in names
             where name.Length == 5
             orderby name
             select name.ToUpper();
```



```
var query = names.Where (name => name.Length == 5)
                  .OrderBy (name => name)
                  .Select (name => name.ToUpper());
```

Делегаты Func

Методы LINQ, принимающие делегаты в качестве параметров, должны быть объявлены с указанием типа параметров-делегатов.

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

Для указания типа параметров используется обобщенный делегат Func, объявленный в пространстве имен System.

```
public delegate TResult Func<in T, out TResult>(T arg);
```

Делегат перегружен и может принимать от 0 до 9 входных параметров-типов (T1, T2, T3,...).

Запросы Where() и Select()

Where() – фильтр

Пример: отфильтровать все четные числа

```
IEnumerable<int> result = m.Where(a => a % 2 == 0);
```

Пример: отфильтровать все числа, стоящие на четных местах

```
IEnumerable<int> result = m.Where((a, i) => i % 2 == 0);
```

Select() – проекция

Пример: увеличить все числа в 1000 раз

```
IEnumerable<int> result = m.Select(a => a * 1000 );
```

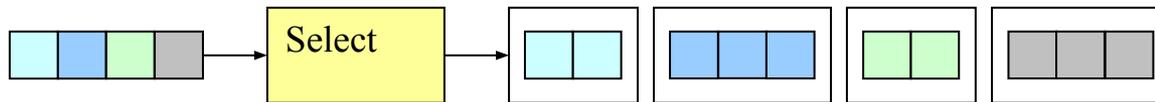
Пример: цепочка вызовов.

```
result = m.Where(a => a % 2 == 0).Select(a => a * 1000 );
```

Запрос SelectMany()

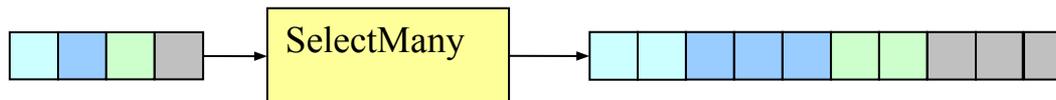
```
string[] input = { "SELECT", "FROM", "WHERE", "ORDER BY" };
```

```
var output = input.Select(s => s.ToCharArray());
```



```
foreach (var array in output) {  
    foreach (var v in array) {  
        Console.Write(v + " ");  
    }  
}
```

```
var output = input.SelectMany(s => s.ToCharArray());
```



```
foreach (var v in output) {  
    Console.Write(v + " ");  
}
```

Упорядочение – OrderBy(), ThenBy()

```
string[] input = { "Select", "Where", "OrderBy", "GroupBy" };
```

Так можно упорядочить строки по алфавиту:

```
var output = input.OrderBy(s => s);
```

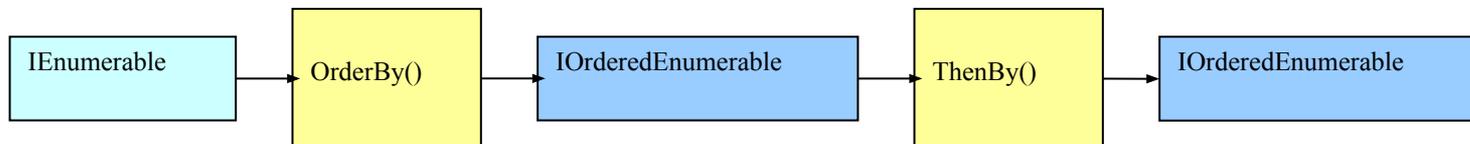
по длине:

```
var output = input.OrderBy(s => s.Length);
```

по последней букве слова:

```
var output = input.OrderBy(s => s[s.Length - 1]);
```

После упорядочения тип итератора меняется.



Пример. Упорядочим книги в первую очередь по фамилии автора, во вторую очередь – по названию.

```
var result = books.OrderBy(b => b.Author).ThenBy(b => b.Title);
```

Группирование – запрос GroupBy()

```
string[] input = { "Select", "Where", "OrderBy", "GroupBy" };
```

Пример: сгруппировать строки по длине.

```
var result = input.GroupBy(s => s.Length);
```



```
foreach (IGrouping<int, string> group in result) {  
    Console.WriteLine("Strings of length {0}",  
group.Key);  
    foreach (string value in group)  
        Console.WriteLine("  {0}", value);  
}
```

Запрос GroupBy() может не только сгруппировать данные, но и выполнить их проекцию. Выражение для проекции задается во втором параметре запроса.

```
var result = input.GroupBy(s => s.Length, s => s.ToUpper());
```

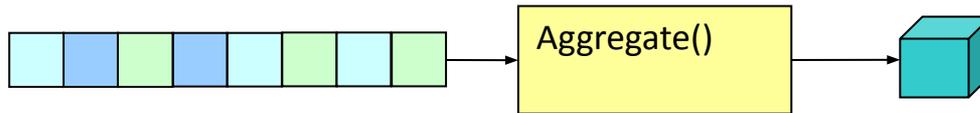
Задание 1

Имеется список книг – объектов класса Book:

```
class Book
{
    public string Name;
    public int Year;
}
```

1. Выберите все книги, в названии которых есть слово LINQ, а год издания високосный.
2. Дана последовательность русских слов. Сколько букв алфавита понадобилось для написания этих слов?
3. Дан массив целых двузначных чисел. Упорядочить их по возрастанию старшего разряда, а затем по убыванию младшего, например, { 14, 12, 23, 20, 33, 32 }.
4. Дан массив книг books. Сколько книг написал каждый автор (распечатать в два столбца: Автор, Количество)?

Агрегация – Agregate, Sum, Average, Max, Min



Суммирование:

```
int[] m = { 2, 6, 4, 9, 1 };  
int result = m.Sum(s => s);
```

Усреднение:

```
var result = m.Average(s =>  
s);
```

Наибольший и наименьший элемент:

```
var result = m.Max(s => s);  
var result = m.Min(s => s);
```

Параметр у всех запросов
означает преобразование перед
агрегированием
`books.Max(s => s.Year)`

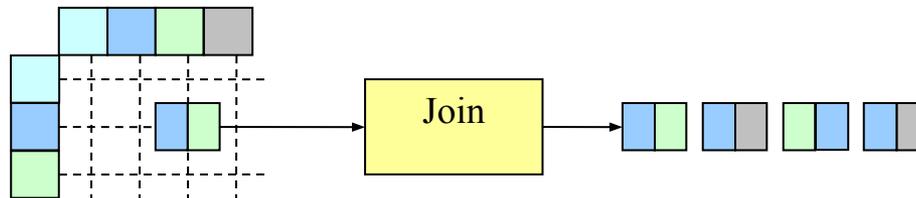
Обобщенный запрос на агрегацию

```
public static U Aggregate<T, U>(this IEnumerable<T> source,  
                                U seed,  
                                Func<U, T, U> func)
```

```
var sum = m.Aggregate(0, (s, i) => s += i);
```

Соединение – Join

Все элементы одной последовательности попарно сравниваются с элементами второй последовательности, и для каждой пары проверяется условие соединения.



Параметры запроса:

- вторая последовательность,
- функция для построения первого ключа,
- функция для построения второго ключа,
- функция для образования проекции.

```
string[] boys = { "Alex", "Bob", "Charley", "Dick"};  
string[] girls = { "Caroline", "Barbara", "Ann", "Adel"};
```

Образуем пары с именами на одну букву.

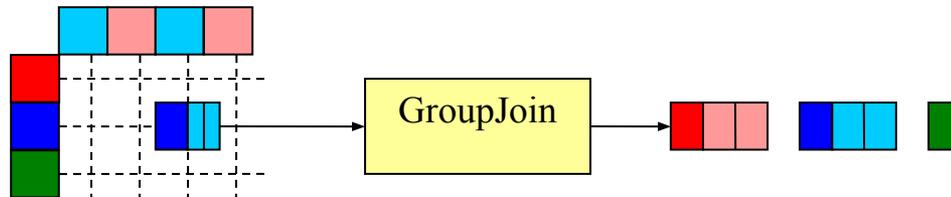
```
var result = boys.Join(girls,  
    x => x[0],  
    y => y[0],  
    (x, y) => new {Boy = x, Girl = y}  
);
```

```
foreach (var r in result)  
    Console.WriteLine("{0} + {1}", r.Boy, r.Girl);
```

Alex + Ann
Alex + Adel
Bob + Barbara
Charley + Caroline

Групповое соединение - GroupJoin

Пару составляют элемент первой последовательности и все соответствующие ему элементы второй последовательности.



Образует групповое соединение имен на одну букву.

```
string[] boys = { "Alex", "Bob", "Charley", "Dick"};
string[] girls = { "Caroline", "Barbara", "Ann", "Adel"};

var result = boys.GroupJoin(girls, x => x[0], y => y[0],
    (x, matches) => new { Boy = x, Girls = matches });
```

```
foreach (var r in result) {
    Console.WriteLine(r.Boy);
    foreach (var g in r.Girls) {
        Console.WriteLine("    " + g);
    }
}
```

```
Alex
    Ann
    Adel
Bob
    Barbara
Charley
    Caroline
Dick
```

Задание 2

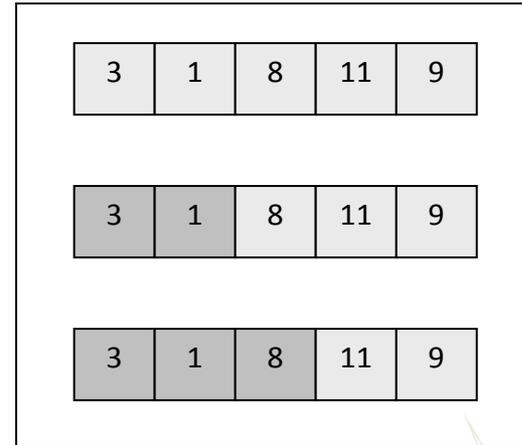
1. Выразите через запрос Agregate() запросы Count(), Max() и Average().
2. Дан массив книг books. Сколько книг написал каждый автор (получить последовательность пар: автор – число книг)?
3. Дан массив чисел. Составьте все пары ...
 - а) из всевозможных чисел;
 - б) из чисел, не равных между собой;
 - в) исключив пару (a, b), если пара (b, a) уже есть в результате.

Остальные запросы

Группа	Методы
Разбиение	<i>Take, Skip, TakeWhile, SkipWhile</i>
Множества	<i>Union, Intersect, Except, Distinct</i>
Отдельные элементы	<i>First, Last, ElementAt, Single (-OrDefault)</i>
Генераторы	<i>Repeat, Range, Empty</i>
Преобразования	<i>ToArray, ToList, ToDictionary, ToLookup, AsEnumerable</i>
Преобразования элементов	<i>OfType, Cast</i>
Кванторы	<i>All, Any, Contains</i>
Разное	<i>Concat, Reverse, SequenceEqual, DefaultIfEmpty</i>

Разбиение: Take, Skip, TakeWhile, SkipWhile

```
int[] m = { 3, 1, 8, 11, 9 }  
};  
  
var result = m.Take(2);  
  
var result = m.TakeWhile(a => a <  
10);
```



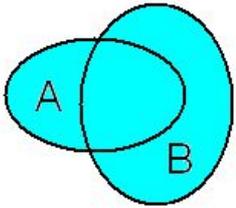
Запросы Skip и SkipWhile – противоположность запросам Take и TakeWhile



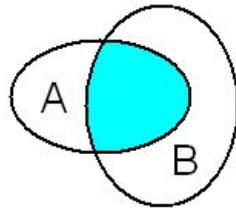
ПОЗИТИВ

НЕГАТИВ

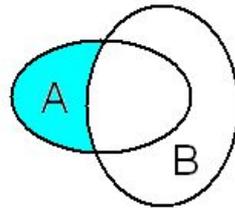
Union, Intersect, Except, Distinct



A.Union(B)



A.Intersect(B)



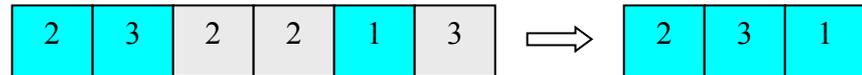
A.Except(B)

Дубликаты
игнорируются

```
int[] m = { 2, 1, 3, 2 }, m1 = { 2, 12, 13, 12};  
var result = m.Except(m1); // result = { 1, 3}
```

Distinct() устраняет дубликаты.

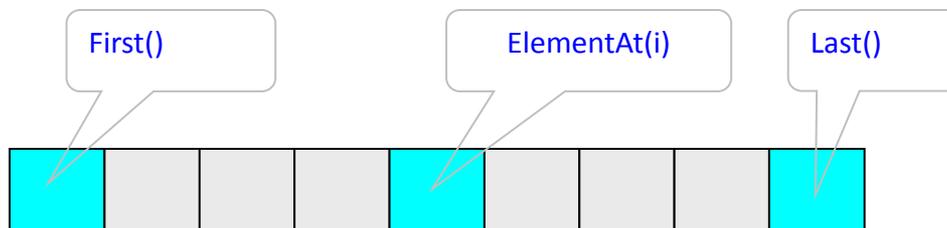
```
int[] m = {2, 3, 2, 2, 1, 3};  
var result = m.Distinct();
```



```
class MyComparer: EqualityComparer<int> {  
    public override bool Equals(int x, int y)  
    { return x % 10 == y % 10; }  
    public override int GetHashCode(int obj)  
    { return obj % 10; }  
}
```

```
int[] m = { 12, 13, 12, 2, 1, 3 };  
var result = m.Distinct(new MyComparer());  
// result = { 2, 3, 1}
```

Отдельные элементы: First, Last, ElementAt, Single



У всех запросов есть условный вариант:

FirstOrDefault

LastOrDefault

SingleOrDefault

ElementAtOrDefault

Генераторы: Repeat, Range, Empty

Repeat() – последовательность из повторений.

```
IEnumerable<string> strings = Enumerable.Repeat("GL", 10);
```

Range() – отрезок целых чисел.

```
IEnumerable<int> squares = Enumerable.Range(1, 10);
```

Empty() – пустая последовательность.

```
IEnumerable<decimal> empty = Enumerable.Empty<decimal>();
```

Пустую последовательность можно использовать как начальное значение агрегата в запросе Aggregate().

ToArray, ToList, ToDictionary, ToLookup, AsEnumerable

Запросы `ToArray()` и `ToList()` просто возвращают массив или коллекцию, составленную из элементов исходной последовательности.

Запрос `ToDictionary()` преобразует последовательность в словарь – коллекцию пар (ключ, значение). При этом можно задать функцию выбора ключа, функцию выбора значения и функцию-компаратор, которая принимает решение об эквивалентности ключей.

Запрос `ToLookup()` преобразует последовательность в мультисловарь. От словаря он отличается тем, что одному ключу может соответствовать любое количество значений, в том числе и ни одного.

Метод `AsEnumerable()` позволяет привести тип, наследующий интерфейс `IEnumerable<T>` (например, `IQueryable<T>`), к типу `IEnumerable<T>`. Это может понадобиться для того, чтобы компилятор выбрал реализацию запросов из класса `Enumerable`, а не из класса `Queryable`.

Преобразования элементов: OfType, Cast

Запрос `OfType<TResult>()` выбирает из входной последовательности элементы, которые могут быть приведены к типу `TResult`. В результате на выходе получается последовательность типа `IEnumerable<TResult>`.

Запрос `OfType<TResult>()` применим не только к `IEnumerable<T>`, но и к `IEnumerable`. Благодаря этому LINQ способен работать с коллекциями старого типа: `ArrayList`, `SortedList`, `Hashtable`.

```
static void Main() {
    Book[] books = {
        new Book {Author = "Албахари", Title = "LINQ: карманный справочник", Year = 2009 },
        new Book {Author = "Раттц", Title = "LINQ: язык запросов", Year = 2008 },
        new Book {Author = "Kimmel", Title = "LINQ Unleashed", Year = 2008 }
    };
    ArrayList bookList = new ArrayList(books);

    var result1 = bookList.OfType<Book>().Take(2);
    var result2 = books.OfType<Book>().Take(2);
    var result3 = books.Take(2);
}
```

Запрос `Cast<TResult>()` отличается от запроса `OfType` тем, что пытается привести к типу `TResult` все элементы входной последовательности. Если с каким-то элементом это не получается, выбрасывается исключение `InvalidCastException` (запрос `OfType` исключения не выбрасывает, он просто пропускает неподходящий элемент)

Кванторы: All, Any, Contains

Эти запросы возвращают логическое значение и называются кванторами, как аналогичные функции на множествах.

Квантор **All** возвращает true, если все элементы последовательности удовлетворяют заданному условию. Условие задается при помощи функции – параметра запроса.

Квантор **Any** возвращает true, если в последовательности есть хотя бы один элемент, удовлетворяющий заданному условию.

Квантор **Contains** возвращает true, если последовательность содержит заданный элемент. При этом можно задать собственное правило сравнения в виде объекта-компаратора.

Разное: Concat, Reverse, SequenceEqual, DefaultIfEmpty

Последовательности можно не только разрезать, но и сшивать запросом `Concat()`.

```
int[] m = {3, 1, 8,};  
var result = m.Concat(m); // result = { 3, 1, 8, 3, 1, 8, }
```

Запрос `Reverse()` придает элементам последовательности обратный порядок.

```
int[] m = {10, 20, 30};  
var result = m.Reverse(); // result = { 30, 20, 10}
```

Запрос `SequenceEqual()` сравнивает две последовательности поэлементно. Сравнение элементов может выполняться при помощи пользовательского объекта – компаратора.

Запрос `DefaultIfEmpty<T>()` подменяет пустую последовательность нулевым элементом базового типа `T`. При желании вместо нулевого может быть подставлено любое другое фиксированное значение базового типа.