

# Language Integrated Query – LINQ



# LINQ

## Language Integrated Query – LINQ

Набор языковых и платформенных средств для написания структурированных и безопасных в отношении типов запросов к локальным коллекциям объектов и удаленным источникам данным (базы данных, документы XML и т.д.)

*По типу обра*

*LINQ to Objects – библиотеки для обработки коллекций объектов в памяти,*

*LINQ to SQL – библиотеки для работы с базами данных,*

*LINQ to XML*

*LINQ to Entity*

- 1) LINQ-запрос похож на SQL
- 2) гибче и способен управлять широким диапазоном логических структур данных
- 3) может обрабатывать данные с иерархической организацией

# LINQ to Objects

## ► Операторы запросов

- отложенные операторы
- не отложенные операторы

### Возврат

- `IEnumerable<T>` или `var`

### Код

- именованные методы
- анонимные методы
- лямбда-выражения

### Форма

- Выражения запросов (синтаксис запросов)
- Стандартная точечная нотация C# с вызовом методов на объектах и классах (синтаксис методов)

набор классов, содержащих  
типичные методы обработки  
коллекций

# Операторы:

Агрегация (Count, Min, Max)

Преобразование (Cast, ofType, ToArray, ToList, ToDictionary)

Конкатенация (Concat)

Элемент (Last, First, Single, ElementAt+ Default)

Множество (Except, Distinct, Union)

Генерация (Empty, Range, Repeat)

Соединение (Join, GroupJoin)

Упорядочивание (OrderBy, ThenBy, Reverse,....)

Проекция (Select, SelectMany)

Разбиение (Skip, Take, +While)

Ограничение (Where)

Квантификатор (Any, All, Contains)

Эквивалентность (SequenceEqual)

# СИНТАКСИС

```
string[] names = {"Анна", "Станислав", "Ольга", "Сева"};
```

```
// Использование точечной нотации (операции)  
IEnumerable<string> result1 = names  
    .Where(n => n.Length < 6)  
    .Select(n => n);
```

```
// Использование синтаксиса выражения запроса  
IEnumerable<string> result2 = from n in names  
                               where n.Length < 6  
                               select n;
```

Синтаксис выражений запросов поддерживается : Where, Select, SelectMany, Join, GroupJoin, GroupBy, OrderBy, ThenBy, OrderByDescending и ThenByDescending.

# Грамматика выражений запросов

- ▶ 1) Начало - from
- ▶ 2) [ from, let или where]
- ▶ 3) [orderby, *ascending* или *descending*]
- ▶ 4) [select или group]
- ▶ 5) [конструкции into, join, или повторение с п.2. ]

Выражение → в методы расширения

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/standard-query-operators-overview>

# Отложенные операторы

## Оператор условия - Where

- Фильтрация элементов в последовательность

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

указывает на метод-обобщение,  
идентифицирующий  
извлекаемые поля

ссылается на тип,  
подвергшийся расширению

метод  
расширения  
класса  
Enumerable,  
находится в  
пространстве  
имен  
System. Linq

```
string[] names = {"Анна", "Станислав", "ольга", "Сева"};
```

```
IEnumerable<string> qwe =  
    names.Where(p => p.StartsWith("А"));
```

псевдонимом для строки в массиве

# Как написаны операторы?

```
static public class Some
{
    static public IEnumerable<string> FindL(this IEnumerable<string> values,
                                             Func<string, bool> test)
    {
        var result = new List<string>();
        foreach (var str in values)
        {
            if (test(str))
            {
                result.Add(str);
            }
        }
        return result;
    }
}
```

```
string[] names = new string[] { "Ольга", "Станислав", "Ольга", "Сева", "Ольга" };

var rez = names.FindL(n=>n.StartsWith("О"));
```



# Оператор проекции - Select

- Для создания выходной последовательности одного типа из входной последовательности элементов другого типа

```
public static IEnumerable<S> Select<T, S>(
    this IEnumerable<T> source,
    Func<T, S> selector);
```

```
string[] names = {"Анна", "Станислав", "Ольга", "Сева"};
IEnumerable<int> nameLen =
    names.Select(p => p.Length);
```

```
IEnumerable<int> nameLen2 = from p in names
                           select p.Length;
```

```
var obj = names.Select(p => new { p, p.Length });
```

## Создание нового типа

```
class NewType
{
    public string Name{get; set;}
    public int Leng { get; set; }
}
```

```
string[] names = { "Анна", "Станислав", "ольга", "Сева" };
```

```
IEnumerable<NewType> nameLen = names.Select
    (p => new NewType { Name = p, Leng =p.Length });
```

## Выборка данных

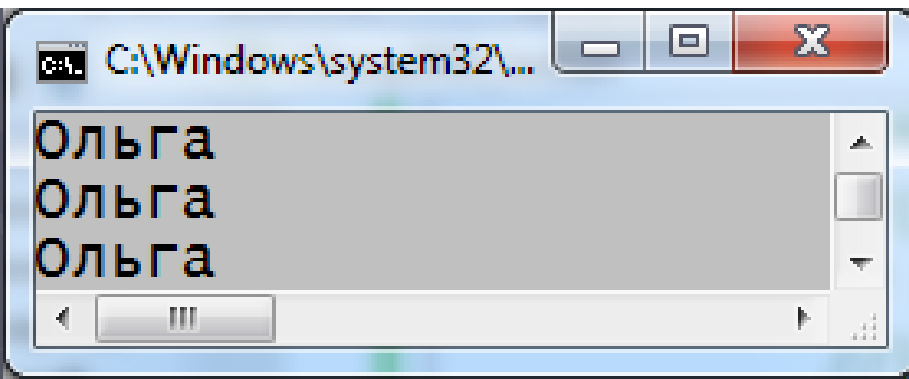
```
string[] names = { "Ольга", "Станислав", "Ольга", "Сева" ,  
"Ольга"};
```

```
IEnumerable<string> aNames =  
    names.Where(n => String.Equals(n, "Ольга"))  
        .Select(n => n);
```

фильтрует данные в  
соответствии с  
указанным критерием


```
foreach (string name in aNames)  
{  
    Console.WriteLine(name);  
}
```

```
IEnumerable<string> aNames3 =  
    from n in names  
    where  
        String.Equals(n, "Ольга")  
    select n;
```



```
var students = new[] {  
    new { studentID = 1, FirstName = "Anna", Country = "Belarus",  
          Spec = "Poit" },  
    new { studentID = 2, FirstName = "Helena", Country = "Bulgaria",  
          Spec = "Poit" },  
    new { studentID = 3, FirstName = "Aex", Country = "Germany",  
          Spec = "Isit" }  
};
```

```
IEnumerable<string> aStud =  
    students.Where(s => s.Country.StartsWith("B"))  
             .Where(c=>c.Spec.Equals("Poit"))  
             .Select(n => n.FirstName);
```



передает из этой  
перечисляемой коллекции  
только одно поле FirstName

## ► Отложенные вычисления

приложение не создает коллекцию в ходе выполнения метода расширения LINQ — коллекция перечисляется, только когда выполняется ее обход

```
string[] names = { "Ольга", "Станислав", "Ольга", "Сева", "Ольга" };
```

```
    IEnumerable<int> nameLen2 = from p in names  
                                select p.Length;
```

```
names[2] = "D";
```

Данные из массива names не извлекаются, не вычисляются, пока не будет выполняться сквозной обход элементов коллекции

```
foreach (int name in nameLen2){  
    Console.WriteLine(name);  
}
```

**отложенные операции** (выполняются не во время инициализации, а только при их вызове) и **не отложенные операции** (выполняются сразу).

# Опертор SelectMany

- Создание выходной последовательности с проекцией "один ко многим"

```
public static IEnumerable<S> SelectMany<T, S>(
    this IEnumerable<T> source,
    Func<T, IEnumerable<S>> selector);
```

```
string[] names = {"Анна", "Станислав", "Ольга", "Сева"};

IEnumerable<char> letters =
    names.SelectMany(p => p.ToArray());
```

```
А н н а С т а н и с л а в О л ь г а С е в а
```

# Оператор разбиения Take

- Возвращает указанное количество элементов из входной последовательности, начиная с ее начала

```
public static IEnumerable<T> Take<T>(
    this IEnumerable<T> source,
    int count);
```

```
string[] names = {"Анна", "Станислав",
                  "Ольга", "Сева"};
```

```
IEnumerable<string> group = names.Take(2);
```

# Оператор TakeWhile

- Возвращает элементы из входной последовательности, пока истинно некоторое условие, начиная с начала последовательности

```
string[] names = {"Анна", "Станислав",  
                  "Ольга", "Сева"};
```

```
IEnumerable<string> shortNames =  
    names.TakeWhile(p => p.Length < 5);
```



# Оператор Skip

- Пропускает указанное количество элементов из входной последовательности, начиная с ее начала, и выводит остальные

```
string[] names = {"Анна", "Станислав",  
                  "Ольга", "Сева"};
```

```
IEnumerable<string> names2 = names.Skip(2);
```

Ольга

Сева

Для продол

# Оператор Concat

- Соединяет две входные последовательности и выдает одну выходную последовательность

```
string[] names = {"Анна", "Станислав",  
                  "Ольга", "Сева"};
```

```
IEnumerable<string> names4 =  
    names.Take(1).Concat(names.Skip(3));
```

Анна

Сева

Для продолжения наж

# OrderBy и OrderByDescending


- Позволяют выстраивать входные последовательности в определенном порядке

```
public static IOrderedEnumerable<T> OrderBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector)
    where K : IComparable<K>;
```

```
string[] names = {"Анна", "Станислав", "Ольга",
"Сева"};
```

```
IEnumerable<string> names5
    names.OrderBy(s => s.Length);
```

определяет выражения, которые  
нужно использовать для сортировки  
данных



```
var students = new[] {  
    new { studentID = 1, FirstName = "Anna", Country = "Belarus",  
        Spec = "Poit" },  
    new { studentID = 2, FirstName = "Melena", Country = "Bulgaria",  
        Spec = "Poit" },  
    new { studentID = 3, FirstName = "Lena", Country = "Germany",  
        Spec = "Isit" }  
};
```

```
IEnumerable<string> aSpecStud =  
    students.OrderBy(s => s.Spec)  
        .OrderBy(s=>s.FirstName)  
        .Select(n => n.Spec + " " + n.FirstName);
```

```
Poit Anna  
Isit Lena  
Poit Melena
```

```
IEnumerable<string> aSpecStud2 =  
    from s in students  
    orderby s.Spec  
    orderby s.FirstName  
    select s.Spec + " " + s.FirstName;
```

# ThenBy и ThenByDescending

- Позволяет упорядочивать последовательно по нескольким критериям, вызывается после OrderBy

```
string[] names = {"Анна", "Станислав", "Ольга",  
"Сева"};
```

```
IEnumerable<string> names6 =  
    names.OrderBy(s => s.Length)  
        .ThenBy(s => s);
```

```
IEnumerable<string> names7 =  
    names.OrderBy(s => s.Length)  
        .ThenByDescending(s => s);
```

```
Анна  
Сева  
Ольга  
Станислав  
Сева  
Анна  
Ольга  
Станислав
```

# Оператор соединения - Join

- ▶ выполняет внутреннее соединение по эквивалентности двух последовательностей на основе ключей

```
public static IEnumerable<V> Join<T, U, K, V>(
    this IEnumerable<T> outer,
    IEnumerable<U> inner,
    Func<T, K> outerKeySelector,
    Func<U, K> innerKeySelector,
    Func<T, U, V> resultSelector);
```

```
string[] names = {"Анна", "Станислав", "Ольга", "Сева"};
int[] key = { 1, 4, 5, 7 };
var sometype = names
    .Join(
        key, // внутренняя
        w => w.Length, // внешний ключ выбора
        q => q, // внутренний ключ выбора
        (w, q) => new // результат
        {
            id = w,
            name = string.Format("{0} ", q),
        });

foreach (var item in sometype)
    Console.WriteLine(item);
```

```
{ id = Анна, name = 4 }
{ id = Ольга, name = 5 }
{ id = Сева, name = 4 }
```

Для продолжения нажмите любую клавишу .

# Оператор группировки GroupBy

- Используется для группирования элементов входной последовательности.

```
public static IEnumerable<IGrouping<K, T>> GroupBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector);
```

```
string[] names = {"Анна", "Станислав", "Ольга", "Сева"};
```

```
IEnumerable<IGrouping<int, string>> outerSequence =
    names.GroupBy(o => o.Length );
```

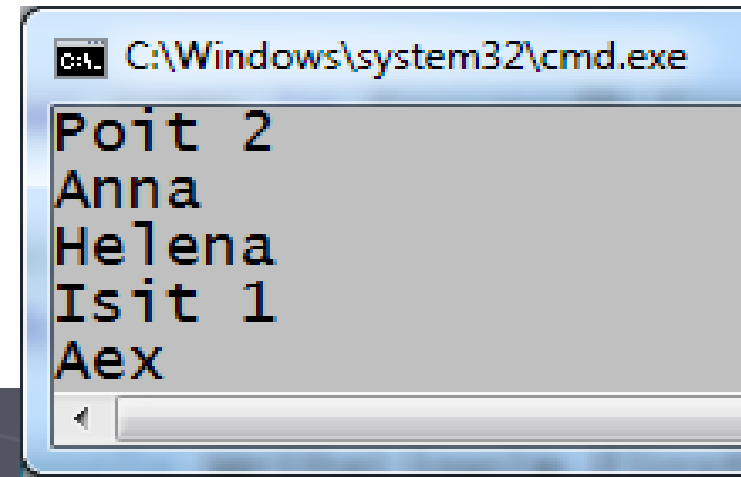
```
foreach (var item in outerSequence)
{
    Console.WriteLine(item.Key);
    foreach (var element in item)
        Console.WriteLine(element);
}
```

```
4
Анна
Сева
9
Станислав
5
Ольга
```

результатом работы  
метода GroupBy  
является  
перечисляемый набор  
групп, каждая из  
которых представляет  
собой перечисляемый  
набор строк



```
var students = new[] {  
    new { studentID = 1, FirstName = "Anna", Country = "Belarus",  
          Spec = "Poit" },  
    new { studentID = 2, FirstName = "Helena", Country = "Bulgaria",  
          Spec = "Poit" },  
    new { studentID = 3, FirstName = "Aex", Country = "Germany",  
          Spec = "Isit" }  
};  
  
var GroupedBySpec = students.GroupBy(s => s.Spec);  
  
foreach (var name in GroupedBySpec)  
{  
    Console.WriteLine(name.Key + " " + name.Count());  
    foreach (var m in name)  
    {  
        Console.WriteLine(m.FirstName);  
    }  
}
```



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the output of the C# code: "Poit 2", "Anna", "Helena", "Isit 1", and "Aex". The text is displayed in a monospaced font, with the first line "Poit 2" on a new line, followed by "Anna" and "Helena" on the same line, then "Isit 1" on a new line, and finally "Aex" on the same line as "Isit 1".

```
C:\Windows\system32\cmd.exe  
Poit 2  
Anna  
Helena  
Isit 1  
Aex
```

# Оператор Distinct

- Удаляет дублированные элементы из входной последовательности

```
int[] key = { 1, 4, 5, 5,5,7,7,7,7 };  
  
IEnumerable<int> nums = key.Distinct();  
  
foreach (var item in nums)  
    Console.WriteLine(item);
```



```
1  
4  
5  
7
```

# Оператор объединения Union

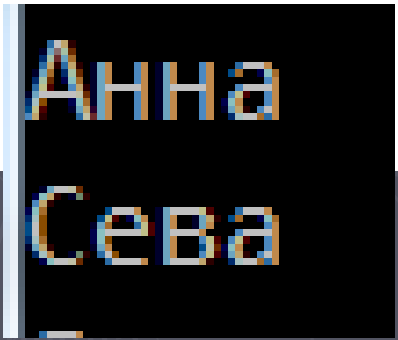
- ▶ Возвращает объединение множеств из двух исходных последовательностей

```
string[] names = {"Анна", "Станислав", "Ольга", "Сева"};
```

```
IEnumerable<string> names9 = names.Take(1);
```

```
IEnumerable<string> names10 = names.Skip(3);
```

```
IEnumerable<string> union =  
    names9.Union<string>(names10);
```



Анна

Сева

# Оператор Intersect

- Возвращает пересечение множеств из двух исходных последовательностей

```
string[] names = {"Анна", "Станислав", "Ольга", "Сева"};
```

```
IEnumerable<string> names9 = names.Take(2);
```

```
IEnumerable<string> names10 = names.Skip(1);
```

```
IEnumerable<string> inter =  
    names9.Intersect<string>(names10);
```

```
foreach (var item in inter)  
    Console.WriteLine(item);
```

Станислав

# Оператор Except

- ▶ Возвращает последовательность, содержащую все элементы первой последовательности, которых нет во второй последовательности



# Оператор Cast

- Используется для приведения каждого элемента входной последовательности в выходную последовательность указанного типа

```
string[] names = {"Анна", "Станислав", "Ольга",  
"Сева"};
```

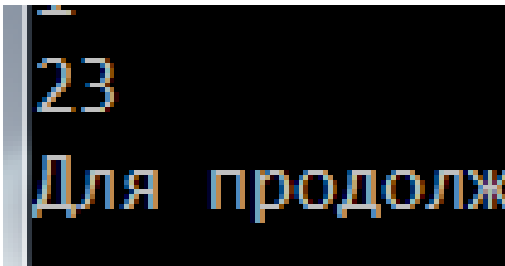
```
var seq = names.Cast<int>();  
Console.WriteLine("Тип данных seq: " +  
seq.GetType());
```

```
данных seq: System.Linq.Enumerable+<CastIterator>d__1`1[System.Int32]
```

# Оператор OfType

- Используется для построения выходной последовательности, содержащей только те элементы, которые могут быть успешно преобразованы к указанному типу.

```
ArrayList ala = new ArrayList();  
    ala.Add(new SByte());  
    ala.Add(new Decimal (23));  
    ala.Add(new String('0',8));
```



```
var seq = ala.OfType<Decimal>();  
foreach (var item in seq)  
    Console.WriteLine(item);
```

- ▶ Оператор `DefaultIfEmpty` возвращает последовательность, содержащую элемент по умолчанию, если входная последовательность пуста.
- ▶ Оператор `Range` генерирует последовательность целых чисел.

```
public static IEnumerable<int> Range(  
    int start,  
    int count);
```

```
IEnumerable<int> numberss = Enumerable.Range(34, 15);
```

```
34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
```

```
foreach (int i in numberss)  
    Console.Write(i + " ");
```



- ▶ Оператор Repeat генерирует последовательность, повторяя указанный элемент заданное количество раз.

```
IEnumerable<int> nqq = Enumerable.Repeat(10, 5);
```

- ▶ Оператор Empty генерирует пустую последовательность заданного типа.

# Не отложенные операторы

## Оператор преобразования ToArray

- ▶ создает массив типа T из входной последовательности типа T

```
int[] key = { 1, 4, 5, 5,5,7,7,7,7 };  
int[] arr = key.ToArray();
```

Сохранятся кэшированную  
коллекцию в массиве

# Оператор ToList

- Создает List типа T из входной последовательности типа T.

```
string[] names = {"Анна", "Станислав",  
"Ольга", "Сева"};
```

```
List<string> auto = names.ToList();
```

## ► Оператор ToDictionary создает Dictionary

```
string[] namer = { "Анна", "Станислав", "Ольга" };  
Dictionary<int, string> eDictionary =  
    namer.ToDictionary(k => k.Length);  
  
foreach (var i in eDictionary)  
    Console.Write(i.Key + " " + i.Value);
```

```
4  Анна9  Станислав5  Ольга
```

# Оператор ToLookup

- Создает объект Lookup типа  $\langle K, T \rangle$  или, возможно,  $\langle K, E \rangle$  из входной последовательности типа  $T$ , где  $K$  — тип ключа, а  $T$  — тип хранимых значений.

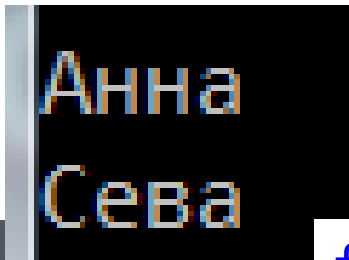
```
string[] actor = { "Анна", "Станислав", "Ольга",  
"Сева", "Николай" };
```

```
ILookup<int, string> lookup =  
    actor.ToLookup(y => y.Length);
```

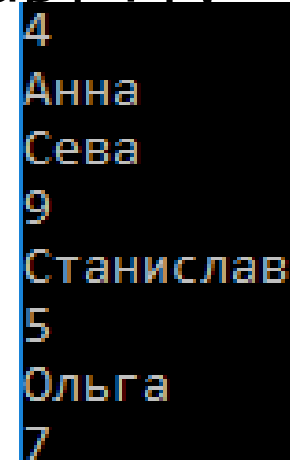
```
IEnumerable<string> actors = lookup[4];
```

```
foreach (var u in actors)  
    Console.WriteLine( u);
```

```
foreach (var u in lookup)  
    { Console.WriteLine(u.Key);  
      foreach (var y in u)  
          Console.WriteLine(y);
```



Анна  
Сева



4  
Анна  
Сева  
9  
Станислав  
5  
Ольга  
7

- ▶ Оператор SequenceEqual определяет, эквивалентны ли две входные последовательности.

```
public static bool SequenceEqual<T>(
    this IEnumerable<T> first,
    IEnumerable<T> second);
```

- ▶ Оператор First возвращает первый элемент последовательности или первый элемент последовательности, соответствующий предикату

```
string[] names = {"Анна", "Станислав", "Ольга", "Сева"};
string fnam = names.First(p => p.StartsWith("С"));
```

- ▶ Оператор FirstOrDefault подобна First во всем, кроме поведения, когда элемент не найден.
- ▶ Оператор Last возвращает последний элемент последовательности или последний элемент, соответствующий предикату
- ▶ Оператор LastOrDefault подобна Last во всем, за исключением поведения в случае, когда элемент не найден.

- ▶ Оператор `Single` возвращает единственный элемент последовательности или единственный элемент последовательности, соответствующий предикату

```
string[] names = {"Анна", "Станислав", "Ольга",  
"Сева"};  
string sst =  
names.Where(s => s.Length == 5).Single();
```

- ▶ Оператор `SingleOrDefault` подобна `Single`, но отличается поведением в случае, когда элемент не найден



- ▶ Оператор `ElementAt` возвращает элемент из исходной последовательности по указанному индексу.
- ▶ Оператор `Any` возвращает `true`, если любой из элементов входной последовательности отвечает условию.

```
string[] names = {"Анна", "Станислав", "Ольга",  
"Сева"};
```

```
bool rex = names.Any(s => s.StartsWith("О"));
```

- ▶ Оператор All возвращает true, если каждый элемент входной последовательности отвечает условию.

```
string[] names = {"Анна", "Станислав", "Ольга",  
"Сева"};  
rex = names.All(s => s.Length > 2);
```

- ▶ Оператор Contains возвращает true, если любой элемент входной последовательности соответствует указанному значению.

```
string[] names = {"Анна", "Станислав", "Ольга",  
"Сева"};  
bool contains = names.Contains("Ольга");
```

- ▶ Оператор Count возвращает количество элементов во входной последовательности.
- ▶ Оператор LongCount - значение типа long.

```
long ccount = Enumerable.Range(8, 98)  
    .Concat(Enumerable.Range(1, int.MaxValue))  
    .LongCount(s => s > 67);
```

- ▶ Оператор Sum возвращает сумму числовых значений, содержащихся в элементах последовательности.

```
2147483618
```

```
long oSum = key.Sum();
```

- ▶ Оператор Min Max возвращает минимальное максимальное значение входной последовательности.

```
IEnumerable<string> aStud =  
    students.Where(s => s.Country.StartsWith("B"))  
        .Where(c => c.Spec.Equals("Poit"))  
        .Select(n => n.FirstName);  
  
string aMax = aStud.Max();
```

- ▶ Оператор Average возвращает среднее арифметическое числовых значений элементов входной последовательности.

# Итераторы

- метод, оператор или аксессор, возвращающий по очереди члены совокупности объектов и имеет оператор `yield`.

```
IEnumerator IEnumerable.GetEnumerator()  
{  
    for (int i = 0; i < figure.Length; i++)  
    {  
        yield return figure[i];  
    }  
}
```

`yield return figure[3];`  
`yield return figure[4];`  
`yield return figure[5];`

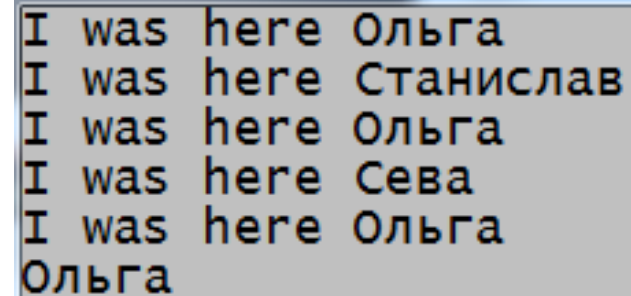
При обращении к оператору `yield return` будет сохраняться текущее местоположение и при переходе к следующей итерации для получения нового объекта, итератор начнет выполнения с этого местоположения.

# Отложенная инициализация

создание объекта откладывается до первого использования

```
static public IEnumerable<string> FindL(this IEnumerable<string>
values, Func<string, bool> test)
{
    var result = new List<string>();
    foreach (var str in values)
    {
        Console.WriteLine("I was here {0}", str);
        if (test(str))
        {
            result.Add(str);
        }
    }
    return result;
}

string[] names = new string[] { "Ольга", "Станислав", "Ольга",
"Сева", "Ольга" };
```



```
I was here Ольга
I was here Станислав
I was here Ольга
I was here Сева
I was here Ольга
Ольга
```

```
var rez = names.FindL(n=>n.StartsWith("O")).Take(1);
```

# yield - КОНТЕКСТНОЕ КЛЮЧЕВОЕ СЛОВО

*Именованный итератор*

```
static public IEnumerable<string> FindL(this  
IEnumerable<string> values, Func<string, bool> test)  
{  
  
    foreach (var str in values)  
    {  
        Console.WriteLine("I was here {0}", str);  
        if (test(str))  
        {  
            yield return str;  
        }  
    }  
}
```

I was here Ольга  
Ольга

позволяет передавать аргументы  
итератору, управляющему процессом  
получения конкретных элементов из  
коллекции

следующий объект, возвращаемый итератором  
Имеет спец. назначение только в блоке  
итератора

```
string[] names = new string[] { "Ольга", "Станислав", "Ольга",  
"Сева", "Ольга" };
```

```
var rez = names.FindL(n=>n.StartsWith("O")).Take(1);
```

# yield

```
public class Range
{
    public int Low { get; set; }
    public int High { get; set; }

    public IEnumerable<int> GetNumbers()
    {
        for (var counter = Low; counter <= High; counter++)
        {
            yield return counter;
        }
    }
}

var range = new Range { low = 0, high = 10 };
var enumerator = range.GetNumbers();
range.High = 5; // изменяем свойство объекта range
foreach (var number in enumerator)
{
    console.WriteLine(number);
}
```



```
public static class Helper
{
    public static IEnumerable<int> GetNumbers()
    {
        var i = 0;
        while (true)
        {
            yield return i++;
        }
    }
}
```

```
foreach (var number in Helper.GetNumbers())
{
    Console.WriteLine(number);
    if (number == 20)
    {
        break;
    }
}
```

```
public static class Helper2
{
    public static IEnumerable<int> GetNumbers()
    {
        var i = 0;
        while (true)
        {
            yield return i++;
            if (i == 21)
            {
                yield break;
            }
        }
    }
}

foreach (var number in Helper2.GetNumbers())
{
    Console.WriteLine(number);
}
```

# PLINQ (Parallel LINQ)

- ▶ позволяет выполнять обращения к коллекции в параллельном режиме (скорость на многоядерных машинах)
  - По умолчанию, если невозможно использует последовательную обработку
  - Параллельно для больших объемов и сложных операциях
  - Источник делится на сегменты и каждый обрабатывается отдельно

# AsParallel()

- ▶ распараллеливает запрос к источнику данных

```
var source = Enumerable.Range(10, 20000);  
var parallelQuery = from num in source.AsParallel()  
                    where num % 100 == 0 && num%3==0  
                    select num;  
  
parallelQuery.ForAll((e) => Console.WriteLine(e));
```

```
1100  
300  
1200  
900  
600  
2400  
2700  
3000  
1500  
3300  
3600  
1800  
4200  
2100  
5100
```

```
var list = Enumerable.Range(10, 20000);
var sw = new Stopwatch();

sw.Restart();
var result = (from l in list.AsParallel()
              where l > 14536 select l).ToList();
sw.Stop();
```

```
Console.WriteLine($"call .AsParallel() before:
                  {sw.ElapsedMilliseconds}");
```

```
sw.Restart();
result = (from l in list
          where l > 14536 select l).AsParallel().ToList();
sw.Stop();
```

```
Console.WriteLine($"call .AsParallel() after:
                  {sw.ElapsedMilliseconds}");
```

```
call .AsParallel() before: 10
call .AsParallel() after: 5
```

```
var result = (from l in list.AsParallel()  
              where l > 14536  
              select l).ToList();
```

```
call .AsParallel() before: 9  
call .AsParallel() after: 3  
Для продолжения нажмите любую к
```

```
result = (from l in list  
          where l > 14536  
          select l). ToList();
```

# ForAll ()

```
(from num in source.AsParallel()  
where num % 100 == 0 && num % 3 == 0  
select num).  
ForAll((n)=>Console.WriteLine(n));
```

Выводит данные в том же потоке, в котором они обрабатываются  
Быстрее цикла

# класс ParallelEnumerable

AsSequential()	конвертирует объект ParallelQuery<T> в коллекцию IEnumerable<T> так, что все запросы выполняются последовательно.
AsOrdered()	при параллельной обработке заставляет сохранять в ParallelQuery<T> порядок элементов (это замедляет обработку).
AsUnordered()	при параллельной обработке позволяет игнорировать в ParallelQuery<T> порядок элементов (отмена вызова AsOrdered()).
WithCancellation()	устанавливает для ParallelQuery<T> указанное значение токена отмены.
WithDegreeOfParallelism()	указывает для ParallelQuery<T>, на сколько параллельных частей нужно разбивать коллекцию для обработки.
WithExecutionMode()	задаёт опции выполнения параллельных запросов в виде перечисления ParallelExecutionMode.



# AsOrdered()

- ▶ данные склеиваются в общий набор неупорядоченно

```
var source = Enumerable.Range(10, 100);
```

```
parallelQuery.ForAll((e) => Console.WriteLine(e));  
    (from num in source.AsParallel()  
     where num % 2 == 0  
     select num)  
     .ForAll((n)=>Console.WriteLine(n));
```

```
10  
14  
12  
16  
24  
26  
28  
30  
32  
34  
18
```

```
parallelQuery.ForAll((e) => Console.WriteLine(e));  
    (from num in source.AsParallel().AsOrdered()  
     where num % 2 == 0  
     select num)  
        .ForAll((n)=>Console.WriteLine(n));
```

приводит к увеличению издержек,  
поэтому подобный запрос будет выполняться  
медленнее, чем неупорядоченный.

# Обработка ошибок в Parallel

- ▶ если возникнет ошибка в одном из потоков, то система прерывает выполнение всех потоков
- исключение **AggregateException**



?

```
var studentIT = new[] {  
    new { studentID = 1, FirstName = "Anna", Marks = new []{4,8,4,9}},  
    new { studentID = 2, FirstName = "Melena", Marks = new []{ 10,8,6,9 }},  
    new { studentID = 3, FirstName = "Lena", Marks = new []{7,2,4,5 }}  
};  
  
var rezult98 = studentIT.Where(n => n.Marks.Average() > 8)  
    .Where(s => s.Marks[0] > 8 && s.studentID > 1)  
    .SelectMany(n => n.Marks)  
    .Take(2)  
    .Union(  
        studentIT.Where(n => n.Marks.Contains(2))  
        .SelectMany(n => n.Marks)  
    );
```

10 8 7 2 4 5