

# Коллекции LINQ



# Типы коллекций

## ▶ необобщенные

- ▶ наличие разнотипных данных
- ▶ ссылки на данные типа object (не обеспечивают типовую безопасность)
- ▶ **System.Collections**

коллекции, в которых элемент коллекции представлен как object (слаботипизированные коллекции)

## ▶ обобщенные

- ▶ обеспечивают типовую безопасность
- ▶ **System.Collections.Generic**

## ▶ специальные

- ▶ **System.Collections.Specialized**

## ▶ с поразрядной организацией

- ▶ BitArray

## ▶ параллельные

- ▶ многопоточный доступ к коллекции
- ▶ **System.Collections.Concurrent**

*Каждый класс коллекции оптимизирован под конкретную форму хранения данных и доступа к ним,*

*и каждый из них предоставляет специализированные методы*

# *Интерфейсы, используемые в коллекциях C#*

- ▶ **IEnumerable<T>**
  - для foreach
  - GetEnumerator()

перечислитель, с помощью которого становится возможен последовательный перебор коллекции

- ▶ **IEnumerator<>**

позволяет перебирать элементы коллекции

- ▶ **ICollection<T>**

- ▶ Count
- ▶ CopyTo()
- ▶ Add(), Remove(), Clear()

- ▶ **IList<T>**

позволяет получать элементы коллекции по порядку

- ▶ Индексатор
- ▶ Insert()
- ▶ Remove()

▶ ISet<T>

▶ IDictionary<TKey, TValue>

▶ IComparer<T> сравнения двух объектов

---

▶ ICollection

- определяет элементы

▶ IComparer

- Compare()

# Классы необобщенных коллекций

- ▶ **ArrayList** - IList, ICollection, IEnumerable, ICloneable  
Определяет динамический массив
- ▶ **BitArray** - ICollection, IEnumerable, ICloneable
- ▶ **Hashtable** Определяет хеш-таблицу для пар "ключ-значение"
- ▶ **Queue** Определяет очередь
- ▶ **SortedList** - класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу
- ▶ **Stack** Определяет стек

- 1) хранят ссылки на объекты →  
при сохранении или извлечении элементов требуется приведение типов  
(исключение BitArray)
- 2) включены в библиотеку с целью обратной совместимости  
с существующими приложениями  
→ применять не рекомендуется
- 3) В UWP эти классы недоступны

# Класс ArrayList

определяется массив переменной длины, который состоит из ссылок на объекты и может динамически увеличивать и уменьшать свой размер

```
ArrayList arr1 = new ArrayList(); // 16
ArrayList arr2 = new ArrayList(1000); // 1000
ArrayList arr3 = new ArrayList();
```

- ▶ Свойства – Capacity, Count, Item
- ▶ Метод - Add , AddRange, BinarySearch, Clear, Clone, CopyTo, GetRange, Sort, RemoveRange, Reverse, IndexOf ....

# пример

```
ArrayList list = new ArrayList();  
list.Add(2.3);  
list.Add(55);  
list.AddRange(new string[] { "one", "two" });  
list.RemoveAt(0);  
list.Reverse();
```

# Обобщенные коллекции

Dictionary <Tkey, TValue>

▶ LinkedList<T>

▶ List<T>

▶ Queue<T>

▶ SortedDictionary<Tkey, TValue>

▶ SortedList<T> (использовании памяти и в скорости вставки и удаления)

▶ HashSet<T> и SortedSet<T>

▶ Stack<T>

преимущества: повышение производительности (не надо тратить время на упаковку и распаковку объекта) и повышенная типобезопасность.



# Классы обобщенных коллекций

System.Collections.Generic

| Тип коллекции                  | Особенности                                                                                                          |
|--------------------------------|----------------------------------------------------------------------------------------------------------------------|
| Dictionary <Tkey, TValue>      | Идентификация и извлечение с помощью ключей                                                                          |
| LinkedList<T>                  | Двусторонний упорядоченный список, оптимизация - вставка и удаление с любого конца, поддерживает произвольный доступ |
| List<T>                        | доступ по индексу, поиск и сортировка                                                                                |
| Queue<T> и Stack<T>            |                                                                                                                      |
| SortedList<T>                  | Отсортированный список пар «ключ–значение», ключи должны реализовывать IComparable<T>, не дублир.                    |
| SortedDictionary<Tkey, TValue> | Вставка медленнее, извлечение быстрее, использует больше памяти чем                                                  |
| HashSet<T>                     | Неупорядоченный набор значений, оптимизация - быстрое извлечение данных, объединений и пересечений наборов.          |

```
Stack<int> numbs = new Stack<int>();
```

```
numbs.Push(3); // в стеке 3
```

```
numbs.Push(5); // в стеке 5, 3
```

```
int stackElement = numbs.Pop();
```

```
Stack<Point> figure = new Stack<Point>();
```

```
figure.Push(new Point() );
```

```
foreach (Point p in figure)
```

```
{
```

```
    Console.WriteLine(p.x);
```

```
}
```

```
Queue<int> numbers = new Queue<int>();  
numbers.Enqueue(3);  
int queueElement = numbers.Dequeue();
```

```
Queue<Point> points = new Queue<Point>();  
points.Enqueue(new Point());  
Point pp = points.Peek();  
Console.WriteLine(pp.x);
```

# Пример: связный список → класс `LinkedList<T>`

```
public static void Main()
{
    LinkedList<int> spisok = new LinkedList<int>();
    spisok.AddFirst(23);
    spisok.AddLast(234);
    LinkedListNode<int> node = spisok.First;
    node = node.Next;
    node = node.Previous;
    spisok.AddAfter(node, 111);
    spisok.RemoveFirst();
    spisok.AddLast(4563);
    for (node = spisok.First; node != null;
        node = node.Next)
        Console.WriteLine(node.Value + "\t");
    node = spisok.Find(111);
}
```

ICollection,  
ICollection<T>,  
IEnumerable,  
IEnumerable<T>,  
ISerializable и  
IDeserializationCallback

# Пример: Dictionary<T, R>

```
Dictionary<string, int> student = new Dictionary<string, int>();
```

```
student.Add("Анна", 8);  
student.Add("Никита", 3);  
student["Алексей"] = 1;  
student["Елена"] = 3;  
student.Remove("Никита");
```

```
student.First(( n) => (n.Value==3));
```

```
Console.WriteLine("The Dictionary contains:");  
foreach (KeyValuePair<string, int> element in student)  
{  
    Console.WriteLine($"Name: { element.Key},  
                        Age: {element.Value}");  
}
```

```
The Dictionary contains:  
Name: Анна, Age: 8  
Name: Алексей, Age: 1  
Name: Елена, Age: 3
```

# Инициализация словаря

```
Dictionary<string, string> fit =  
    new Dictionary<string, string>  
    {  
        [ "ИСИТ" ] = "Понедельник",  
        [ "ДЭВИ" ] = "Вторник",  
        [ "ПОИТ" ] = "Среда",  
        [ "ПОБМС" ] = "Четверг"  
    };
```

# System.Collections.Specialized

- ▶ **CollectionsUtil**

содержит фабричные методы для создания коллекций

- ▶ **HybridDictionary**

Для небольшого ListDictionary  
Для большого количества Hashtable

- ▶ **ListDictionary**

для хранения пар "ключ-значение" используется связный список (небольшое количество)

- ▶ **NameValueCollection**

пары "ключ-значение" относятся к типу string

- ▶ **OrderedDictionary**

индексируемые пары "ключ-значение"

- ▶ **StringCollection**

оптимизация для хранения символьных строк

- ▶ **StringDictionary**

пары ключ-значение типа string

# Битовые коллекции

## ► Класс BitArray

- Изменяемый размер
- ICollection, IEnumerable, ICloneable

System.Collections.Specialized

*And()  
Get  
Not  
Or  
Xor  
Set*

## ► Структура BitVector32

- 32 бита (целое) - хранение – стек → тип значения  
→ выше скорость работы

```
byte[] d = { 12, 100 };
```

```
    BitArray bits = new BitArray(d);
```

```
    bits.SetAll(false);
```

```
    bits.Set(2, true);
```


```
    bits[2] = true;
```

```
    bits[8] = true;
```

```
    foreach (bool b in bits)
```

```
        Console.Write(b ? 1 : 0);
```

```
    Console.WriteLine("\n");
```





# Наблюдаемые коллекции

**System.Collections.ObjectModel**

## ► **ObservableCollection<T>**

- пользовательский интерфейс получает информацию об изменениях коллекции
- унаследован от `Collection<T>`, использует внутри себя `List<T>`, `INotifyCollectionChanged`

```
var obsev = new ObservableCollection<int>();
            obsev.CollectionChanged += CollectionChanged;
            obsev.Add(23);
            obsev.Add(675);
            obsev.Insert(1,78);
        }
        private static void CollectionChanged(object sender,
System.Collections.Specialized.NotifyCollectionChangedEventArgs e)
        {}
```

# Параллельные коллекции

System.Collections.Concurrent

коллекции классов, предназначенные для безопасной работы в многопоточной среде, которыми можно воспользоваться при создании многопоточных приложений

► TryAdd() и TryTake()

*ConcurrentStack<T>*

*ConcurrentBag<T>*

*ConcurrentDictionary<TKey, TValue>*

*BlockingCollection<T>*

....

# Реализация интерфейса

## ► Comparable

- Для сортировки и сравнения объектов (SortedList)
- Требуется реализации
  - `int CompareTo(object obj)`

## ► Comparer

- **`int Compare(object x, object y)`**

```
class Air : IComparable<Air>
{
    public int Number { set; get; }
    public int CompareTo(Air obj)
    {
        if (this.Number > obj.Number)
            return 1;
        if (this.Number < obj.Number)
            return -1;
        else
            return 0;
    }
}

static class Run
{
    public static void Main()
    {
        List<Air> minsk2 = new List<Air>();
        minsk2.Add(new Air());
        minsk2.Sort();
    }
}
```

# Интерфейс ICollection

```
public interface ICollection : IEnumerable
{
    // метод
    void CopyTo(Array array, int index);
    // свойства
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
}
```

# Универсальный интерфейс ICollection<T>

```
public interface ICollection<T> : IEnumerable<T>
{
    // методы
    void Add(T item);
    void Clear();
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
    bool Remove(T item);

    // свойства
    int Count { get; }
    bool IsReadOnly { get; }
}
```

# Интерфейс IList

- описывает набор данных, которые проецируются на массив

```
public interface IList : ICollection
{
    // методы
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);

    // свойства
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[int index] { get; set; }
}
```



# Интерфейс IDictionary

- ▶ протокол взаимодействия для коллекций-словарей  
(KeyValuePair<TKey, TValue> – это вспомогательная структура, у которой определены свойства Key и Value)

```
public interface IDictionary : ICollection
{
    // методы
    void Add(object key, object value);
    void Clear();
    bool Contains(object key);
    IDictionaryEnumerator GetEnumerator();
    void Remove(object key);

    // свойства
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[object key] { get; set; }
    ICollection Keys { get; }           // все ключи словаря
    ICollection Values { get; }         // все значения словаря
}
```



```
public interface IDictionary<TKey, TValue> :  
    ICollection<KeyValuePair<TKey, TValue>>  
{  
    // методы  
    void Add(TKey key, TValue value);  
    bool ContainsKey(TKey key);  
    bool Remove(TKey key);  
    bool TryGetValue(TKey key, out TValue value);  
  
    // свойства  
    TValue this[TKey key] { get; set; }  
    ICollection<TKey> Keys { get; }  
    ICollection<TValue> Values { get; }  
}
```

# интерфейс ISet<T>

```
public interface ISet<T> : ICollection<T>
{
    bool Add(T item);
    void ExceptWith(IEnumerable<T> other);
    void IntersectWith(IEnumerable<T> other);
    bool IsProperSubsetOf(IEnumerable<T> other);
    bool IsProperSupersetOf(IEnumerable<T> other);
    bool IsSubsetOf(IEnumerable<T> other);
    bool IsSupersetOf(IEnumerable<T> other);
    bool Overlaps(IEnumerable<T> other);
    bool SetEquals(IEnumerable<T> other);
    void SymmetricExceptWith(IEnumerable<T> other);
    void UnionWith(IEnumerable<T> other);
}
```

- необобщенный интерфейс IEnumerator или обобщенный интерфейс IEnumerator<T> (Перечислители)

- ▶ Реализация **object Current { get; }**
- ▶ **bool MoveNext()**
- ▶ **void Reset()**
- ▶ **Доступ только для чтения**

```
List<int> arrayList = new List<int>();
    Random ran = new Random();

    for (int i = 0; i < 10; i++)
        arrayList.Add(ran.Next(1, 20));

    // Используем перечислитель
    IEnumerator<int> e = arrayList.GetEnumerator();
    e.MoveNext() ;
    Console.Write(e.Current + "\t");
```

# Перечеслители

```
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```

```
public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```

# 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.

# Именованный итератор

```
public IEnumerable  
    имя_итератора(список_параметров) {  
    // ...  
yield return obj;  
}
```

**yield break;** // *Выход из итератора*

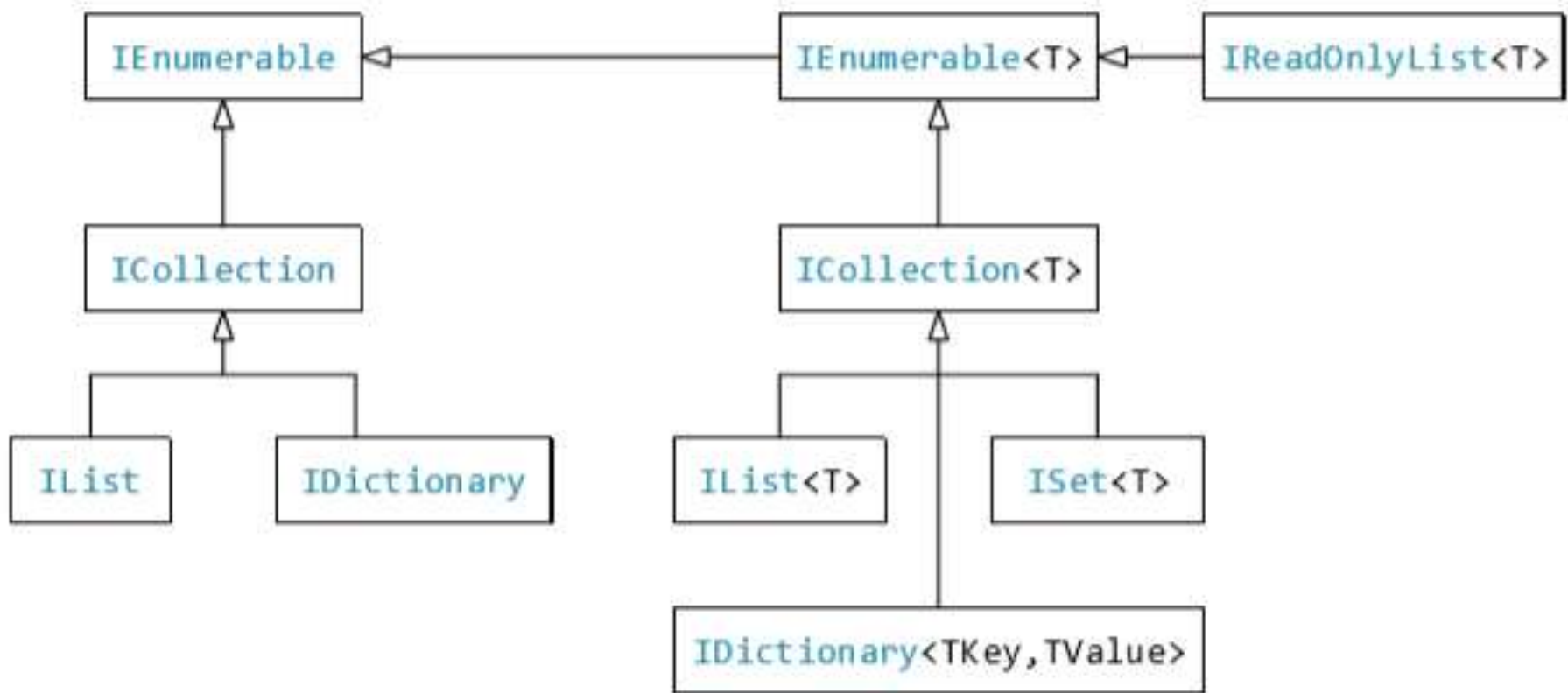
# yield

```
public class Shop : IEnumerable<string>
{
    private string[] _items = new string[0];

...

    public IEnumerator<string> GetEnumerator()
    {
        foreach (var item in _items)
        {
            yield return item;
        }
    }
}
```

# Стандартные интерфейсы коллекций





# 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 источником 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

## ► Операции запросов

- отложенные операции
- не отложенные операции

набор классов, содержащих  
типичные методы обработки  
коллекций

## Возврат

- `IEnumerable<T>` или `var`

## Код

- именованные методы
- анонимные методы
- лямбда-выражения

## Форма

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

# Операции:

**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
```

```
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) 0..\* 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

# Отложенные операции

## Операция Where

### ► Фильтрация элементов в последовательность

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

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

указывает на метод-обобщение,  
идентифицирующий  
извлекаемые поля

ссылается на тип,  
подвергшийся расширению

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

псевдонимом для строки в массиве

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

# Как написаны операции?

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

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

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

# Операция Select - проекция

- 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

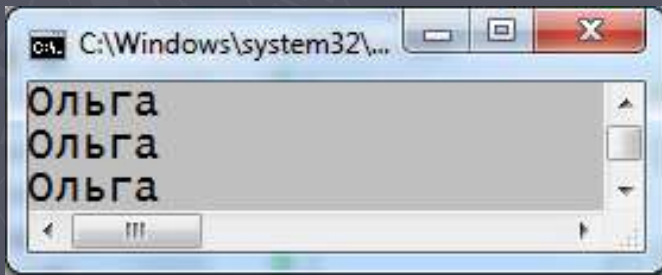
```
string[] names = { "Ольга", "Станислав", "Ольга", "Сева" , "Ольга"};
```

```
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";
```

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

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

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

# Операция 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
```

# Операция 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);
```

# Операция TakeWhile

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

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

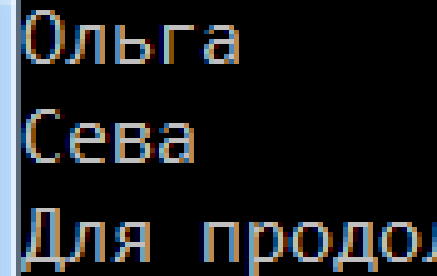
```
IEnumerable<string> shortNames =  
    names.TakeWhile(p => p.Length < 5);
```

# Операция 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
```



# Операция 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 IObservable<T> OrderBy<T, K>(
    this IEnumerable<T> source,
    Func<T, K> keySelector)
    where K : IComparable<K>;
```

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

```
IObservable<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
```

# Операция 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 .
```

# Операция GroupBy

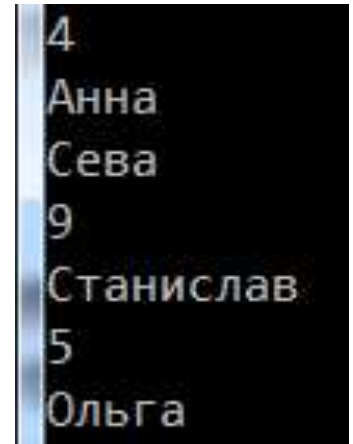
- Используется для группирования элементов входной последовательности.

```
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 );
```

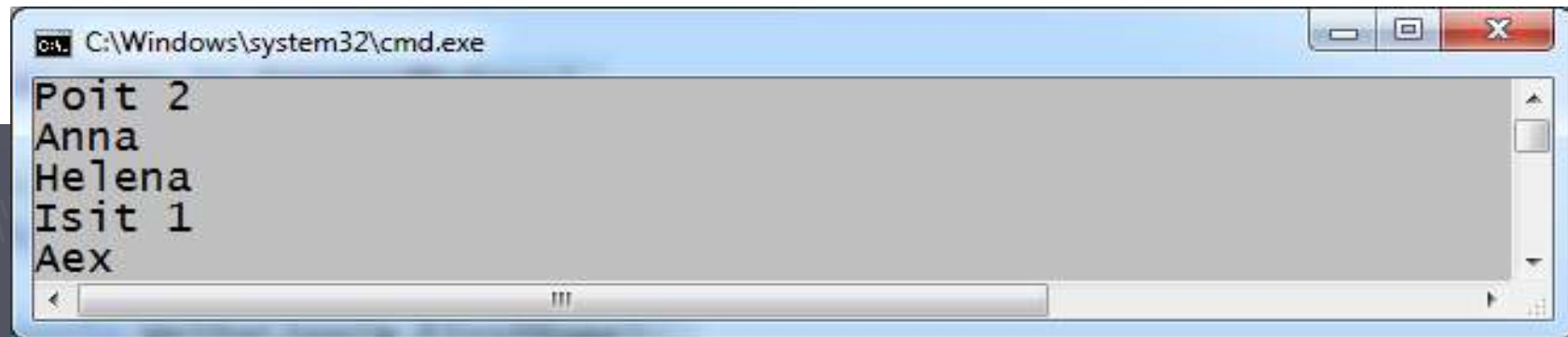
```
foreach (var item in outerSequence)
{
    Console.WriteLine(item.Key);
    foreach (var element in item)
        Console.WriteLine(element);
}
```



```
4
Анна
5
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);  
    }  
}
```



C:\Windows\system32\cmd.exe

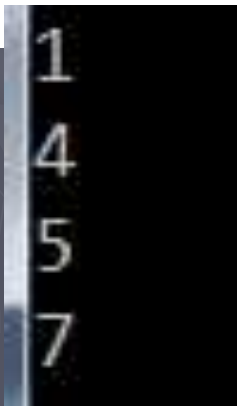
```
Poit 2  
Anna  
Helena  
Isit 1  
Aex
```



# Операция 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);
```

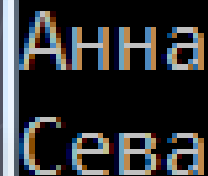


```
1  
4  
5  
7
```

# Операция 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);
```

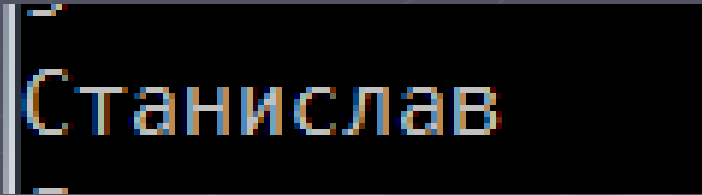


Анна  
Сева

# Операция 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

# Операция Эксерт

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

# Операция 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());
```

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

# Операция 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);
```

23

Для продолж

- ▶ Операция `DefaultIfEmpty` возвращает последовательность, содержащую элемент по умолчанию, если входная последовательность пуста.
- ▶ Операция `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 + " ");
```

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

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

- ▶ Операция Empty генерирует пустую последовательность заданного типа.



# Не отложенные операции

## Операция 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

# Операция ToList

- Создает List типа T из входной последовательности типа T.

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

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

## ► Операция ToDictionary создает Dictionary

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

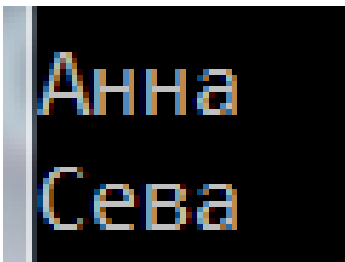
```
4  Анна9  Станислав5  Ольга
```

# Операция ToLookup

- Создает объект Lookup типа <K, T> или, возможно, <K, E> из входной последовательности типа T, где K — тип ключа, а T — тип хранимых значений.

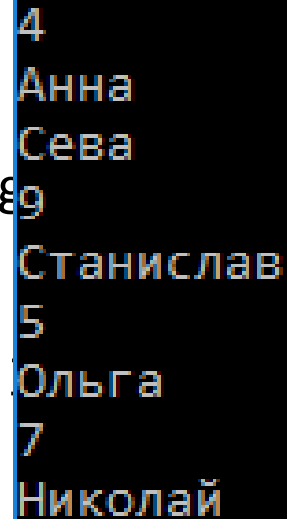
```
string[] actor = { "Анна", "Станислав", "Ольга",  
"Сева", "Николай" };
```

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



Анна  
Сева

```
IEnumerable<string> actors =  
    foreach (var u in actors)  
        Console.WriteLine( u);
```



4  
Анна  
Сева  
5  
Станислав  
Ольга  
7  
Николай

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

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

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

- ▶ Операция 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"));
```

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

- ▶ Операция `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();
```

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

- ▶ Операция `ElementAt` возвращает элемент из исходной последовательности по указанному индексу.
- ▶ Операция `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"));
```



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

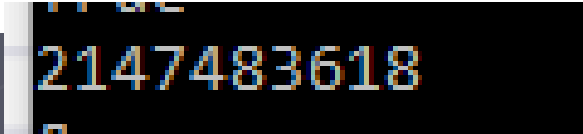
```
string[] names = {"Анна", "Станислав", "Ольга",  
"Сева"};  
rex = names.All(s => s.Length > 2);
```

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

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

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

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



2147483618

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

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

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

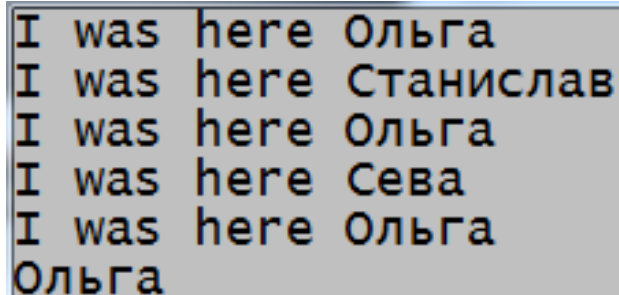
```
IEnumerable<string> aStud =  
    students.Where(s => s.Country.StartsWith("B"))  
        .Where(c => c.Spec.Equals("Poit"))  
        .Select(n => n.FirstName);  
  
string aMax = aStud.Max();
```

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

# Отложенная инициализация

создание объекта откладывается до первого использования

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



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

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

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

# yield - КОНТЕКСТНОЕ КЛЮЧЕВОЕ СЛОВО

## Именованный итератор

```
static public IEnumerable<string> FindL(this IEnumerable<string>
values, Func<string, bool> test)
{
```

I was here Ольга  
Ольга

```
    foreach (var str in values)
    {
```

```
        Console.WriteLine("I was here {0}", str);
```

```
        if (test(str))
```

```
        {
```

```
            yield return str;
```

```
        }
```

```
    }
```

```
}
```

```
}
```

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("О")).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 источникам 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
```

```
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

# AsOrdered()

- ▶ данные склеиваются в общий набор неупорядоченно

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

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

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

```
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.

```
10  
16  
12  
14  
24  
26  
28  
30  
32  
18  
36  
20  
22  
42  
40  
46
```

# Обработка ошибок в Parallel

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

исключение **AggregateException**

Прекратить операцию до ее завершения.  
**WithCancellation()**



