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 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 (Операторы запросов LINQ)
  - Стандартная точечная нотации C# с вызовом методов на объектах и классах (Методы расширений LINQ)

```
string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };

// создаем новый список для результатов
var selectedPeople = new List<string>();
// проходим по массиву
foreach (string person in people)
{
    // если строка начинается на букву T, добавляем в список
    if (person.ToUpper().StartsWith("T"))
        selectedPeople.Add(person);
}
// сортируем список
selectedPeople.Sort();

foreach (string person in selectedPeople)
    Console.WriteLine(person);
```

LINQ позволяет значительно сократить код с помощью интуитивно понятного и краткого синтаксиса.

определение запроса LINQ

from переменная in набор_объектов select переменная;

проходит по всем элементам массива

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

```
string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };

// создаем новый список для результатов
var selectedPeople = from p in people // передаем каждый элемент из people в переменную p
                     where p.ToUpper().StartsWith("T") //фильтрация по критерию
                     orderby p  // упорядочиваем по возрастанию
                     select p; // выбираем объект в создаваемую коллекцию

foreach (string person in selectedPeople)
    Console.WriteLine(person);
```

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

Методы расширения LINQ

определены для интерфейса IEnumerable

реализуют ту же функциональность, что и операторы LINQ типа where или orderby.

```
string[] people = { "Tom", "Bob", "Sam", "Tim", "Tomas", "Bill" };

var selectedPeople = people.Where(p => p.ToUpper().StartsWith("T")).OrderBy(p => p);

foreach (string person in selectedPeople)
    Console.WriteLine(person);
```

В качестве аргумента эти методы принимают делегат или лямбда-выражение.

не каждый метод расширения имеет аналог среди операторов LINQ. И в этом случае можно сочетать оба подхода

```
int number = (from p in people where p.ToUpper().StartsWith("T") select p).Count();

Console.WriteLine(number); // 3
```

Операции:

Агрегация (Count, Min, Max)

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

Конкатенация (Concat)

Элемент (Last, First, Single , ElemetAt+ 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(SequeceEqual)

Синтаксис

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


```
// Использование точечной нотации
IEnumerable<string> rezult1 = names
  .Where(n => n.Length < 6)
  .Select(n => n);

// Использование синтаксиса выражения запроса
IEnumerable<string> rezult2 = 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.

Select: определяет проекцию выбранных значений

Where: определяет фильтр выборки

OrderBy: упорядочивает элементы по возрастанию

OrderByDescending: упорядочивает элементы по убыванию

Join: соединяет две коллекции по определенному признаку

GroupBy: группирует элементы по ключу

GroupJoin: выполняет одновременно соединение коллекций и группировку элементов по ключу

Reverse: располагает элементы в обратном порядке

All: определяет, все ли элементы коллекции удовлятворяют 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 обоих коллекциях

Count: подсчитывает количество элементов коллекции, которые удовлетворяют определенному условию

Sum: подсчитывает сумму числовых значений в коллекции

Average: подсчитывает среднее значение числовых значений в коллекци

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

Last: выбирает последний элемент коллекции

LastOrDefault: выбирает последний элемент коллекции или возвращает значение по умолчанию

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.

Отложенные операции

LINQ-выражение не выполняется, пока не будет произведена итерация или перебор по выборке, например, в цикле foreach

AsEnumerable, **Cast**, **Concat**, DefaultIfEmpty, Distinct, Except, **GroupBy**, GroupJoin, Intersect, **Join**, OfType, OrderBy, OrderByDescending, Range,Repeat, Reverse, **Select**, SelectMany, Skip, SkipWhile, **Take**, TakeWhile, ThenBy, ThenByDescending, **Union, Where**

LINQ-запрос разбивается на три этапа:

1. Получение источника данных;
2. Создание запроса;
3. Выполнение запроса и получение его результатов.

Получение источника данных - определение массива teams:

```
string[] people = { "Tom", "Sam", "Bob" };
```

Создание запроса - определение переменной selectedPeople:

```
var selectedPeople = people.Where(s=>s.Length == 3).OrderBy(s=>s);
```

Выполнение запроса и получение его результатов

```
foreach (string s in selectedPeople)
    Console.WriteLine(s);
```

выполнение запроса происходит не в строке определения: var selectedPeople = people.Where..., а при переборе в цикле foreach.

- После определения запроса он может выполняться множество раз.
- До выполнения запроса источник данных может изменяться.
- **переменная запроса** сама по себе не выполняет никаких действий и не возвращает никаких данных. Она только **хранит набор команд**, которые необходимы для получения результатов. То есть выполнение запроса после его создания **откладывается**. Само получение результатов производится при переборе в цикле foreach.

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 Enumerable, находится в пространстве имен System. Linq

указывает на метод-обобщение, идентифицирующий извлекаемые поля

ссылается на тип, подвергшийся расширению

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

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

псевдонимом для строки в массиве

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);
//4,9,5,4
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 });
```

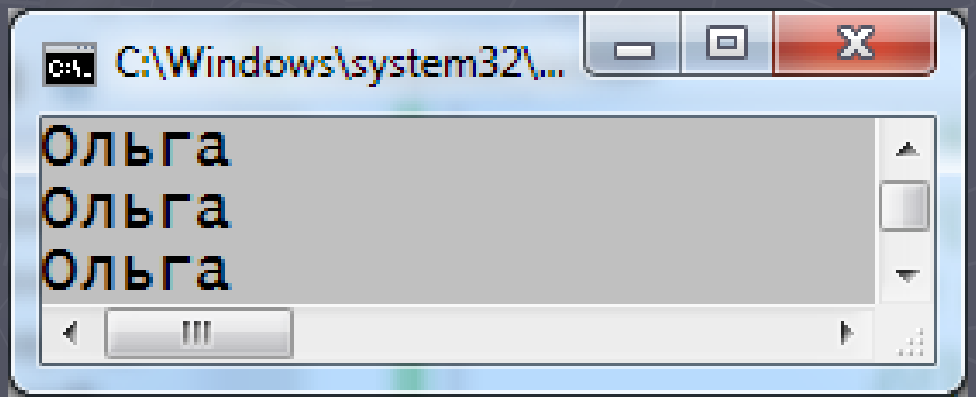
```
Анна4
Станислав9
ольга5
Сева4
```

Выборка данных

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

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

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

фильтрует данные в соответствии с указанным критерием

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



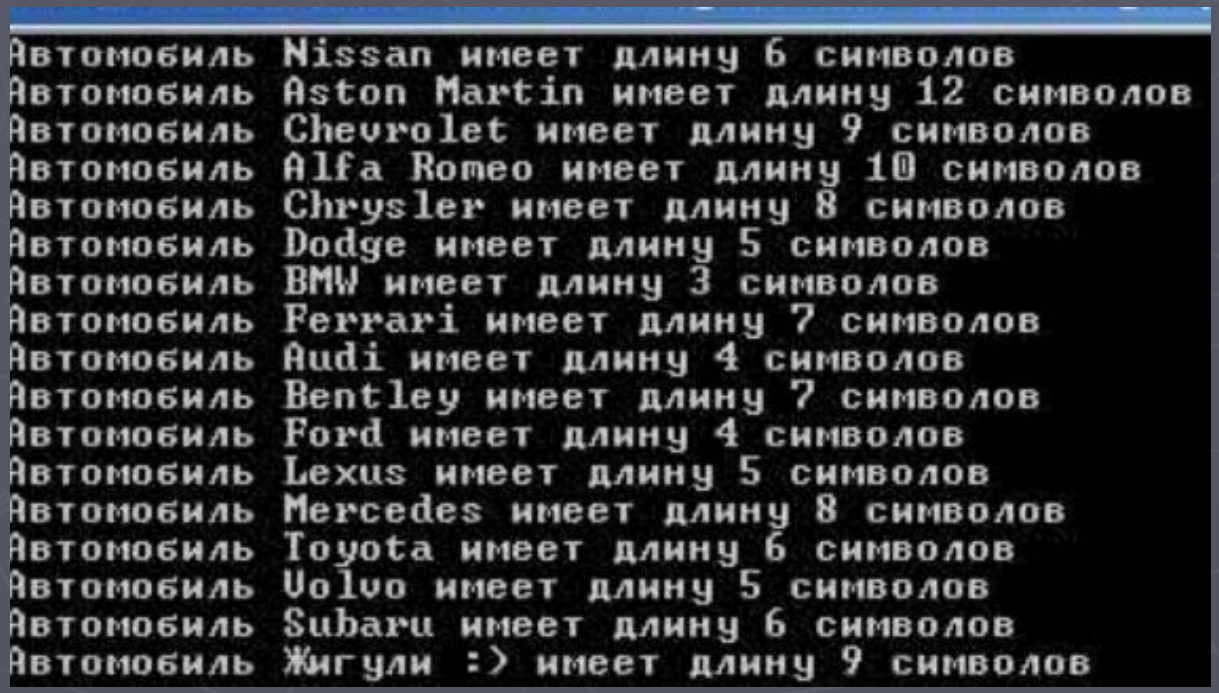
```
var students = new[] {
    new { studentID = 1, FirstName = "Anna", Country = "Belarus",
          Spec = "Poit" },
    new { studentID = 2, FirstName = "Helena", Country  = "Bulgaria",
          Spec = "Poit" },
    new { studentID = 3, FirstName = "Aex", Country  = "Germany",
          Spec = "Isit"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

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

```
Anna
Helena
```

```
string[] cars = { "Nissan", "Aston Martin", "Chevrolet", "Alfa Romeo", "Chrysler", "Dodge", "B
MW",
                          "Ferrari", "Audi", "Bentley", "Ford", "Lexus", "Mercedes", "Toyot
a", "Volvo", "Subaru", "Жигули :)"};
```

```
var carObj = cars.Select(p => new { LastName = p, Length = p.Length });

foreach (var i in carObj)
       Console.WriteLine("Автомобиль {0} имеет длину {1} символов", i.LastName, i.Length);
```



```
var carObj = cars.Select((p, i) => new { Index = i + 1, LastName = p });

foreach (var i in carObj)
      Console.WriteLine( i.Index + ". " + i.LastName);
```

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"
- при перечислении проходит по входной последовательности, получая каждый элемент

```
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());
```

```
АннаСтаниславОльгаСева
```

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);
```

Анна
Станислав

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

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

Операция TakeWhile

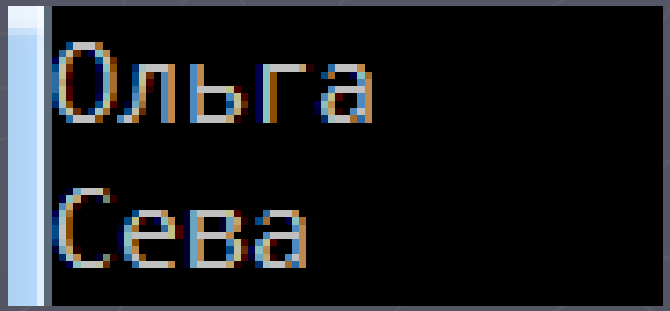
- Возвращает элементы из входной последовательности, пока истинно некоторое условие, начиная с начала последовательности

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

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

Операция Skip

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

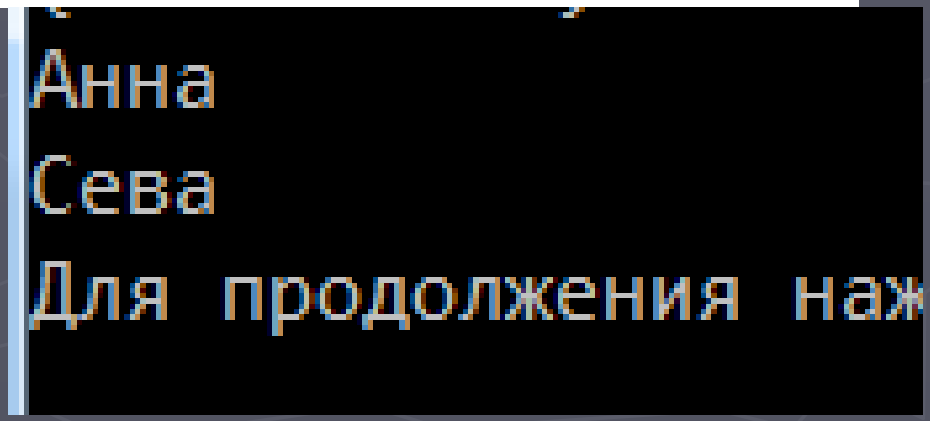
```
string[] names = {"Анна", "Станислав", "Ольга",
"Сева"};

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



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));
```



OrderBy и OrderByDescending
(упорядочивает по убыванию)

- 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);
```

```
Анна
Сева
Ольга
Станислав
```

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

Выполняет две операции сортировки, что неэффективно и некорректно

```
inputSequence.OrderBy(s => s.LastName).OrderBy(s => s.FirstName)...
```

Вместо нее должна использоваться такая цепочка

```
inputSequence.OrderBy(s => s.LastName).ThenBy(s => s.FirstName)...
```

ThenBy и 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).
                   OrderBy(s => s);
```

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

сначала упорядочивает элементы по их длине, затем упорядочивает по самому элементу

ThenBy и ThenByDescending

```
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
```

сначала упорядочивает элементы по их длине, затем упорядочивает по самому элементу

Операция Join

- выполняет внутреннее соединение по эквивалентности двух последовательностей на основе ключей

```
from объект1 in набор1
join объект2 in набор2 on объект2.свойство2
equals объект1.свойство1
```

идет выборка объектов из второй коллекции

свойство объекта из первой выборки, которому должно быть равно свойство объекта из второй выборки. Если эти свойства равны, то оба объекта попадают в финальный результат.

```
class Person(string Name, string Company);
class Company(string Title, string Language);
```

```
Person[] people =
{
    new Person("Tom", "Microsoft"), new Person("Sam", "Google"),
    new Person("Bob", "JetBrains"), new Person("Mike", "Microsoft"),
};
Company[] companies =
{
    new Company("Microsoft", "C#"),
    new Company("Google", "Go"),
    new Company("Oracle", "Java")
};
var employees = from p in people
            join c in companies on p.Company equals c.Title
            select new { Name = p.Name, Company = c.Title, Language = c.Language };

foreach (var emp in employees)
    Console.WriteLine($"{emp.Name} - {emp.Company} ({emp.Language})");

record class Person(string Name, string Company);
record class Company(string Title, string Language);
```

Tom - Microsoft (C#)
Sam - Google (Go)
Mike - Microsoft (C#)

если значение свойства p.Company совпадает со значением свойства c.Title.

Метод Join

```
Join(IEnumerable<TInner> inner,
    Func<TOuter,TKey> outerKeySelector,
    Func<TInner,TKey> innerKeySelector,
    Func<TOuter,TInner,TResult> resultSelector);
```

второй список, который соединяем с текущим

делегат, который определяет свойство объекта из текущего списка, по которому идет соединение

делегат, который определяет свойство объекта из второго списка, по которому идет соединение

делегат, который определяет новый объект в результате соединения

```
Person[] people =
{
    new Person("Tom", "Microsoft"), new Person("Sam", "Google"),
    new Person("Bob", "JetBrains"), new Person("Mike", "Microsoft"),
};
Company[] companies =
{
    new Company("Microsoft", "C#"),
    new Company("Google", "Go"),
    new Company("Oracle", "Java")
};
var employees = people.Join(companies, // второй набор
            p => p.Company, // свойство-селектор объекта из первого набора
            c => c.Title, // свойство-селектор объекта из второго набора
            (p, c) => new { Name = p.Name, Company = c.Title, Language = c.Language });

foreach (var emp in employees)
    Console.WriteLine($"{emp.Name} - {emp.Company} ({emp.Language})");

record class Person(string Name, string Company);
record class Company(string Title, string Language);
```

Операция Join

```
string[] names = {"Анна", "Станислав", "Ольга", "Сева"};
   int[] key = { 1, 4, 5, 7 };

                var sometype = names
                 .Join(
                  key,              //  второй набор
                  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  }
```

```
string[] names = {"A", "Станислав", "Ольга", "Сева"};
   int[] key = { 1, 4, 5, 7 };

                var sometype = names
                 .Join(
                  key,              //  второй набор
                  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 = A, name = 1 }
{ id = Ольга, name = 5 }
{ id = Сева, name = 4 }
```

Операция GroupBy

- Используется для группирования элементов входной последовательности.

```
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
```

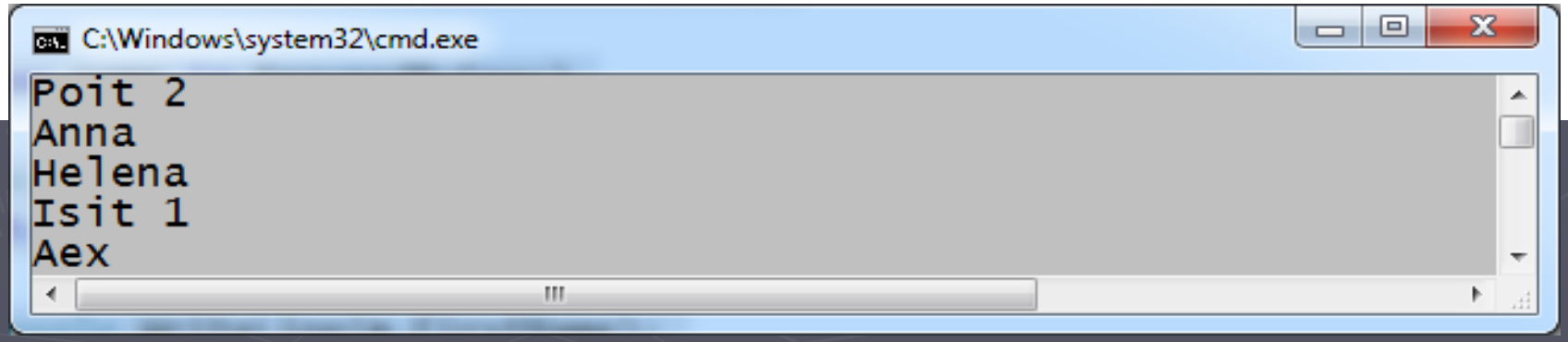
результатом работы метода GroupBy является перечисляемый набор групп, каждая из которых представляет собой перечисляемый набор строк

```
Person[] people =
{
    new Person("Tom", "Microsoft"), new Person("Sam", "Google"),
    new Person("Bob", "JetBrains"), new Person("Mike", "Microsoft"),
    new Person("Kate", "JetBrains"), new Person("Alice", "Microsoft"),
};

var companies = from person in people
                group person by person.Company;

foreach(var company in companies)
{
    Console.WriteLine(company.Key);

    foreach(var person in company)
    {
        Console.WriteLine(person.Name);
    }
    Console.WriteLine(); // для разделения между группами
}

record class Person(string Name, string Company);
```

```
Microsoft
Tom
Mike
Alice

Google
Sam

JetBrains
Bob
Kate
```

```
var companies = people.GroupBy(p => p.Company);

foreach(var company in companies)
{
    Console.WriteLine(company.Key);

    foreach(var person in company)
    {
        Console.WriteLine(person.Name);
    }
    Console.WriteLine(); // для разделения между группами
}

record class Person(string Name, string Company);
```

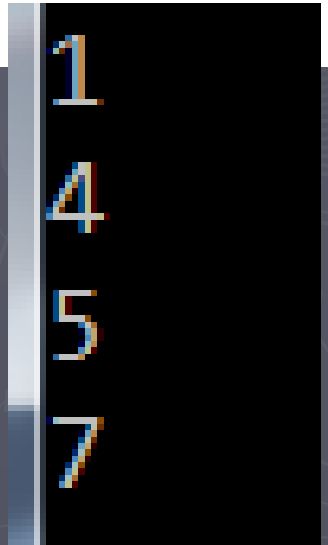
```
var students = new[] {
    new { studentID = 1, FirstName = "Anna", Country = "Belarus",
          Spec = "Poit" },
    new { studentID = 2, FirstName = "Helena", Country  = "Bulgaria",
          Spec = "Poit" },
    new { studentID = 3, FirstName = "Aex", Country  = "Germany",
          Spec = "Isit"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

      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);
```



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);
```



Операция Distinct

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

// удаление дублей
var result = soft.Distinct();

foreach (string s in result)
    Console.WriteLine(s);
```

```
Microsoft
Google
Apple
```

Операция Union

- Возвращает объединение множеств из двух исходных последовательностей. Повторяющиеся элементы добавляются в результат только один раз

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

                 IEnumerable<string> names9 = names.Take(1);
                 IEnumerable<string> names10 = names.Take(2);

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

```
Анна
Станислав
```

Union

```
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);
```

```
Microsoft
Google
Apple
IBM
Samsung
```

Concat

Если нужно простое объединение двух наборов. те элементы, которые встречаются в обоих наборах, дублируются в резутирующей последовательности.

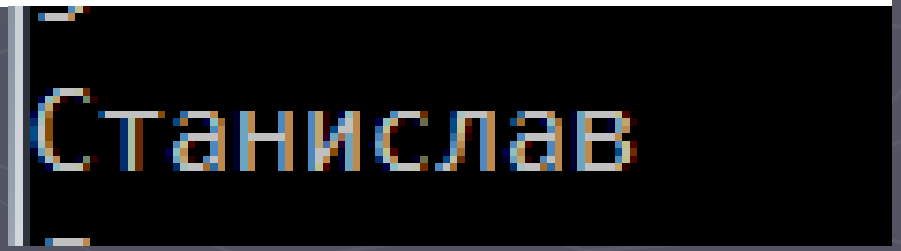
Последовательное применение методов Concat и Distinct будет подобно действию метода Union.

```
var result = soft.Concat(hard);
```

Операция Intersect

- Возвращает пересечение множеств (общих для обоих наборов элементов) 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);
```



```
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);
```

Apple

Операция Except

- разность двух последовательностей. Возвращает последовательность, содержащую все элементы первой последовательности, которых нет во второй последовательности

```
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);
```

из массива soft убираются все элементы, которые есть в массиве hard

```
Microsoft
Google
```

Операция Cast

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

Операция Cast

```
ArrayList employees = Employee.GetEmployeesArrayList();
        Console.WriteLine("Тип данных employees: " +
employees.GetType());

        var seq = employees.Cast<Employee>();
        Console.WriteLine("Тип данных seq: " +
seq.GetType());

        var emps = seq.OrderBy(e => e.firstName);
        foreach (Employee emp in emps)
            Console.WriteLine("{0} {1}", emp.firstName,
emp.lastName);
```

```
Тип данных employees: System.Collections.ArrayList
Тип данных seq: System.Linq.Enumerable+<CastIterator>d__b1`1[LINQ_to_Objects.Emp
loyee]
Anders Hejlsberg
David Lightman
Joe Rattz
Kevin Flynn
William Gates
```

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,  начиная со значения
        int count); протяженностью до
```

```
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.

Немедленное выполнение запроса

методы, которые возвращают одно атомарное значение или один элемент или данные типов Array, List и Dictionary.

Aggregate, All, Any, Average, Contains, Count, ElementAt, ElementAtOrDefault, Empty, First, irstOrDefault, Last, LastOrDefault, LongCount, Max, Min, SequenceEqual, Single, SingleOrDefault, Sum, ToArray, ToDictionary, ToList, ToLookup

Count()

возвращает число элементов последовательности:

который выполняет запрос, неявно выполняет перебор по последовательности элементов, генерируемой этим запросом, и возвращает число элементов в этой последовательности.

```
string[] people = { "Tom", "Sam", "Bob" };
// определение и выполнение LINQ-запроса
var count = people.Where(s=>s.Length == 3).OrderBy(s=>s).Count();

Console.WriteLine(count);    // 3 - до изменения коллекции

people[2] = "Mike";
Console.WriteLine(count);    // 3 - после изменения коллекции
```

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  Анна
9  Станислав
5  Ольга
```

- Операция SequenceEqual определяет, эквивалентны ли две входные последовательности.

```
// returns false
bool isEqual4 = new[] { 2, 1, 2 }.SequenceEqual(new[] { 1, 1, 2 });

// returns true
bool isEqual5 = new[] { 2, 1, 2 }.Skip(1).SequenceEqual(new[] { 1, 1, 2
}.Skip(1));
```

- Операция First возвращает первый элемент последовательности или первый элемент последовательности, соответствующий предикату

Станислав

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

```
string[] people = { "Tom", "Bob", "Tim", "Sam" };

// проверяем, есть ли строка Tom
var first = people.First();  // Tom
Console.WriteLine(first);
```

```
string[] people = { "Tom", "Bob", "Kate", "Tim", "Mike", "Sam" };

// первая строка, длина которой равна 4 символам
var firstWith4Chars = people.First(s=> s.Length == 4);  // Kate
Console.WriteLine(firstWith4Chars);
```

First

если коллекция пуста или в коллекции нет элементов, который соответствуют условию, то будет сгенерировано исключение.

```
string[] people = { "Tom", "Bob", "Kate", "Tim", "Mike", "Sam" };

// первая строка, длина которой равна 5 символам
var firstWith5Chars = people.First(s => s.Length == 5);  // ! исключение
Console.WriteLine(firstWith5Chars);

var first = new string[] {}.First();  // ! исключение
Console.WriteLine(first);
```

- Операция FirstOrDefault подобна First во всем, кроме
  если коллекция пуста или в коллекции не окажется элементов, которые соответствуют условию, то метод возвращает значение по умолчанию

```
string[] people = { "Tom", "Bob", "Kate", "Tim", "Mike", "Sam" };

// первый элемент
var first = people.FirstOrDefault();  // Tom
Console.WriteLine(first);

// первая строка, длина которой равна 4 символам
var firstWith4Chars = people.FirstOrDefault(s => s.Length == 4);  // Kate
Console.WriteLine(firstWith4Chars);

// первый элемент из пустой коллекции
var firstOrDefault = new string[] {}.FirstOrDefault();
Console.WriteLine(firstOrDefault);  // null
```

- Операция Last возвращает последний элемент последовательности или последний элемент, соответствующий предикату
- Операция LastOrDefault подобна Last .

```
string[] people = { "Tom", "Bob", "Kate", "Tim", "Mike", "Sam" };

string? last = people.LastOrDefault();
Console.WriteLine(last); // Sam

string? lastWith4Chars = people.LastOrDefault(s => s.Length == 4);
Console.WriteLine(lastWith4Chars); // Mike

string? lastWith5Chars = people.LastOrDefault(s => s.Length == 5);
Console.WriteLine(lastWith5Chars); // null

string? lastWith5CharsOrDefault = people.LastOrDefault(s => s.Length == 5, "Undefined");
Console.WriteLine(lastWith5CharsOrDefault); // Undefined

// первый элемент из пустой коллекции строк
string? lastOrDefault = new string[] {}.LastOrDefault("hello");
Console.WriteLine(lastOrDefault);  // hello
```

- Операция Single возвращает единственный элемент последовательности или единственный элемент последовательности, соответствующий предикату и ошибку, если элементов больше одного

```
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.

```
    string[] names =
  { "Hartono, Tommy", "Adams, Terry", "Andersen, Henriette Thaulow",

      "Hedlund, Magnus", "Ito, Shu" };

string name = names.ElementAt(3);

Console.WriteLine("The name chosen ... '{0}'.", name);
```

```
The name chosen ... 'Hedlund, Magnus'.
```

- 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); //True
```

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

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

- Операция Count возвращает количество элементов во входной последовательности.

```
        string[] names = {"Анна", "Станислав", "Ольга", "Сева"};
int rex = names.Count();
Console.WriteLine (rex); //4
```

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

```
string[] names = { "Анна", "Станислав", "Ольга", "Сева" };
int sq=names.Sum(x => x.Length);//22
```

- 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.

```
string[] names = { "Анна", "Станислав", "Ольга", "Сева" };
double sq=names.Average(x=>x.Length);//5.5
```

Итераторы

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

yield return: определяет возвращаемый элемент

yield break: указывает, что последовательность больше не имеет элементов

Итераторы

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

возвращает один элемент и запоминает текущую позицию

```
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 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
Ольга
Ольга
```

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

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

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
I was here Станислав
I was here Ольга
Ольга
I was here Сева
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("О")).Take(1);
```

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));
```

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

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);
```

```
10
14
12
16
86
18
96
20
106
22
108
24
26
```

```
var a=from num in source.AsParallel()
                where num % 2 == 0
                select num;
      foreach(int q in a)
      {
          Console.WriteLine(q);
      }
```

```
var source = Enumerable.Range(10, 100);
        var a=from num in source.AsParallel().AsOrdered()
                  where num % 2 == 0
                  select num;
        foreach(int q in a)
        {
            Console.WriteLine(q);
        }
```

```
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
```

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**