

LINQ



1. Что такое LINQ?
2. Цели и задачи LINQ
3. Понятие запроса
 - a. Запрос в LINQ
 - b. Синтаксис запроса
 - c. Исполнение запроса
 - d. Сортировка
 - e. Ключевые слова let и into
 - f. Группировка
 - g. Вложенные запросы
 - h. Объединения (join)

Проблемы, связанные с источниками данных (БД, XML):

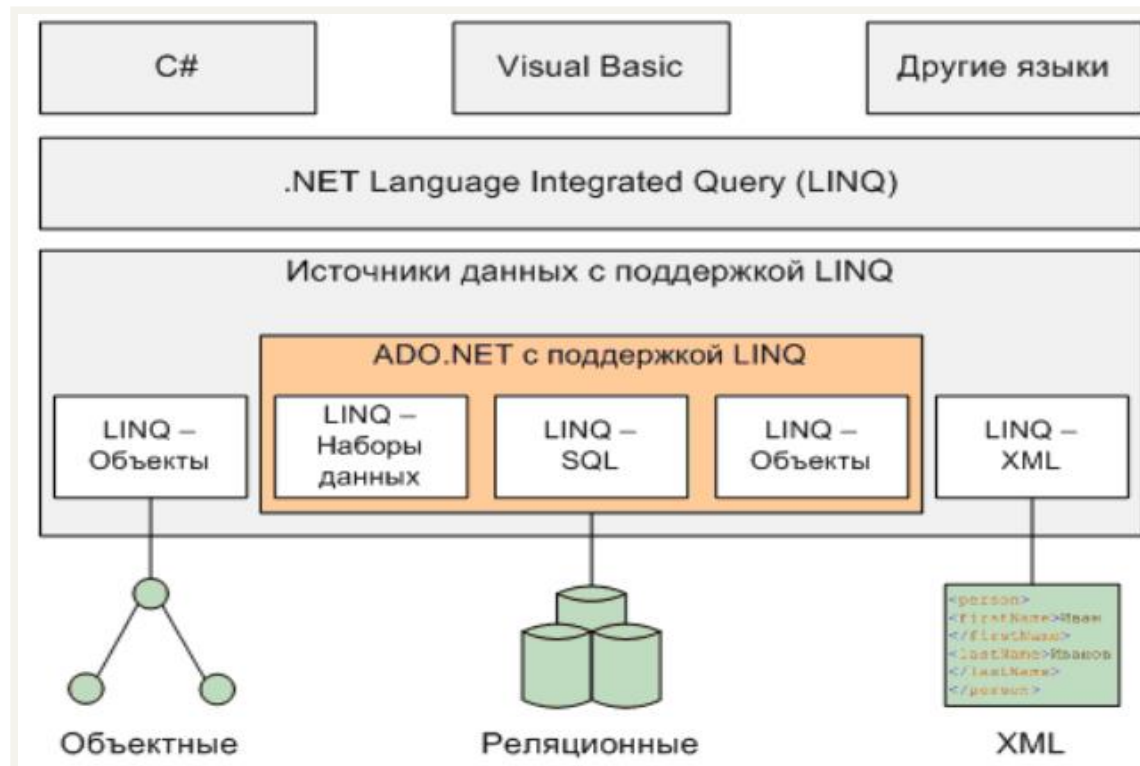
- **Первая сложность** в том, что нельзя программно взаимодействовать с базой данных на уровне естественного языка.
- **Вторая проблема** связана с неудобством, которое вызвано различными типами данных используемыми определенным доменом данных, например, разница между типами базы данных или типами XML и типами данных в языке, на котором написана программа.

Language Integrated Query (LINQ) – проект Microsoft по добавлению синтаксиса языка запросов, напоминающего SQL, в языки программирования платформы. NET Framework.

LINQ – представляет стандартные, легко изучаемые шаблоны для создания запросов и обновления данных; технология может быть расширена для поддержки потенциально любого типа хранилища данных.

LINQ – является революционной инновацией в **.NET Framework** версии **3.5**, которая является мостом между миром объектов и миром данных

Архитектура LINQ



Разновидности LINQ

- **LINQ to Objects.** Эта разновидность позволяет применять запросы LINQ к массивам и коллекциям.
- **LINQ to XML.** Эта разновидность позволяет применять LINQ для манипулирования и опроса документов XML.
- **LINQ to DataSet.** Эта разновидность позволяет применять запросы LINQ к объектам DataSet из ADO.NET.
- **LINQ to Entities.** Эта разновидность позволяет применять запросы LINQ внутри API-интерфейса ADO.NET Entity Framework (EF).
- **Parallel LINQ** (он же PLINQ). Эта разновидность позволяет выполнять параллельную обработку данных, возвращенных запросом LINQ.

Интересный факт...

LINQ — это понятие, связанное только с запросами, поскольку расшифровывается как язык интегрированных запросов (***Language Integrated Query***).

Не думайте так!!!!

Предпочтительнее воспринимать LINQ как механизм итерации данных (***data iteration engine***), но возможно в Microsoft не захотели обозначать эту технологию аббревиатурой DIE ("умереть").

C# использует следующие связанные с LINQ средства:

- неявно типизированные локальные переменные;
- синтаксис инициализации объектов и коллекций;
- лямбда-выражения;
- расширяющие методы;
- анонимные типы.

Синтаксис инициализации объектов и коллекций

```
static List<Car> CreateCars()
{
    return new List<Car>
    {
        new Car(){ID=1, Name="BMW", Price=25000, DriverID=1},
        new Car(){ID=2, Name="Ford", Price=15000, DriverID=2},
        new Car(){ID=3, Name="Opel", Price=1900, DriverID=2},
        new Car(){ID=4, Name="Audi", Price=2200, DriverID=3},
    };
}
```


Расширяющие методы (extension methods) дополняют общедоступный интерфейс типа.

- Методы расширения представляют собой особую разновидность статического метода, но вызываются так же, как методы экземпляра в расширенном типе.
- Методы расширения т.к. не являются методами экземпляра они не нарушают инкапсуляцию.

Особенности

- Методы расширения рекомендуется реализовывать в **ограниченном количестве** и только при необходимости.
- Метод расширения никогда не будет вызван, если он имеет ту же сигнатуру, что и метод, определенный в типе.
- Методы расширения вводятся в область действия на уровне **пространства имен**.

Расширяющие методы

```
class Car
{
    public int Num { get; set; }
    public string Name { get; set; }
    public double Fuel { get; set; }

    public void Go()
    {    // Метод ехать !    }

    public override string ToString()
    {
        return string.Format("{0} - {1} литров", Name, Fuel);
    }
}

//-----
static class FuelStation
{    // Расширяющий метод!
    public static void Filling(this Car car, double fuel)
    {
        car.Fuel += fuel;
    }
}
```

// класс должен быть статическим!

```
static class Program
```

```
{
```

```
    public static void Main()
```

```
    {
```

```
        Car car = new Car { Name = "BMW", Num = 1, Fuel = 50 };
```

```
        Console.WriteLine(car);
```

```
        car.Filling(100);
```

```
        Console.WriteLine(car);
```

```
        string state=car.FuelControl();
```

```
        Console.WriteLine(state);
```

```
        Console.ReadKey();
```

```
    }
```

// РАСШИРЯЮЩИЙ МЕТОД!

```
    public static string FuelControl(this Car car)
```

```
    {
```

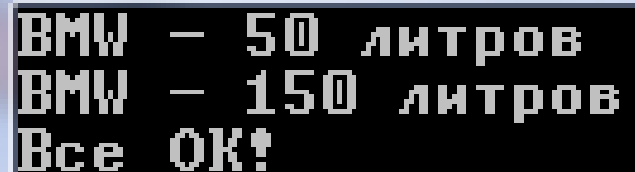
```
        if(car.Fuel<=0)
```

```
            return "Закончилось топливо";
```

```
        return "Все ОК!";
```

```
    }
```

```
}
```



```
BMW - 50 литров  
BMW - 150 литров  
Все ОК!
```

Anonymous type

Анонимные типы предлагают удобный способ инкапсуляции набора свойств в один объект без необходимости предварительного явного определения типа.

```
var instance = new { Name = "Alex", Age = 27 };
```

AnonymousType 'a

Anonymous Types:

'a is new { string Name, int Age }

Имя типа создается компилятором и не доступно на уровне исходного кода.

Анонимные типы являются ссылочными типами, которые происходят непосредственно от класса **object**.

Особенности LINQ

Все операции запроса LINQ состоят из трех различных действий:

- Получение источника данных.
- Создание запроса.
- Выполнение запроса

Любой LINQ - запрос, трансформируется в последовательность вызовов расширяющих методов.

LINQ поддерживают только два языка программирования: C# и Visual Basic.NET.

Запрос в LINQ

Базовый синтаксис выборки

```
var результат = from сопоставляемыйЭлемент in контейнер  
                 select сопоставляемыйЭлемент;
```

Получение подмножества данных

```
var результат = from элемент in контейнер  
                 where булевскоеВыражение select элемент;
```

В LINQ возвращенный набор называется **последовательностью** (*sequence*). Большинство последовательностей LINQ имеют тип [IEnumerable<T>](#)

где T — тип данных объектов, находящихся в последовательности.

Например, если есть последовательность целых чисел, они должны храниться в переменной типа `IEnumerable<int>`.

Возврат результата запроса LINQ

```
// Исходный массив
int[] array = new int[10] {5,8,7,6,2,3,4,7,8,1 };

// получение из массива array значений, которые > 5
// Возвращенная последовательность хранится
// в переменной по имени newArray,
// тип которой реализует обобщенную
// версию интерфейса IEnumerable<T>,
// где T – тип System.Int32
IEnumerable<int> newArray1 = from value in array
                             where value > 5 select value;

foreach (int i in newArray1)
    Console.Write(i+" ");
```



8 7 6 7 8

Часто используемые операции запросов LINQ

Операции запросов	Назначение
from, in	Используются для определения основы любого выражения LINQ , позволяющей извлечь подмножество данных из нужного источника
where	Используется для определения ограничений , т.е. какие элементы должны извлекаться из источника
select	Используется для выбора последовательности из источника
join, on, equals, into	Выполняют соединения на основе указанного ключа .
orderby, ascending, descending	Позволяют упорядочить результирующий набор в порядке возрастания или убывания
group, by	Выдают подмножество с данными, сгруппированными по указанному значению

Выражение запроса должно начинаться предложением **from и оканчиваться предложением **select** или **group**.**

Возврат результата запроса LINQ

```
// получение из массива array значений > 5
// использование неявно типизированной переменной
var newArray2 = from a in array where a > 5 select a;

foreach (var i in newArray2)
    Console.Write(i + " ");

Console.WriteLine(new String('-', 20));

// Исходный массив
string[] Technologies =
    { "WinForms", "ASP.NET", "ADO.NET", "WCF", "WPF" };
// получение из массива Technologies значений, начинающихся на "W"
// использование неявно типизированной переменной
var newArray3 = from a in Technologies where a.StartsWith("W")
select a;

foreach (var i in newArray3)
    Console.Write(i + " ");
```

8 7 6 7 8

WinForms WCF WPF

Language Integrated Query

ID	FirstName	LastName	Salary	StartDate
1	Ivan	Ivanov	94 000	1/4/1992
2	Petr	Petrov	123 000	12/3/1985

```
var employees = new List<Employee> {  
    new Employee  
    {  
        FirstName = "Ivan",  
        LastName = "Ivanov",  
        Salary = 94000,  
        StartDate = DateTime.Parse("1/4/1992")},  
    new Employee  
    {  
        FirstName = "Petr",  
        LastName = "Petrov",  
        Salary = 123000,  
        StartDate = DateTime.Parse("12/3/1985")}};
```

```
var empNames = new List<String>();  
  
foreach (var employee in employees)  
    empNames.Add(employee.LastName);
```

Language Integrated Query

Создание LINQ запроса:

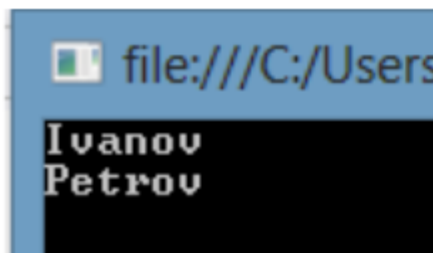
Определяем запрос

```
var empNames =  
    from employee in employees  
    select employee.LastName;
```

//Выполнение другой работы...

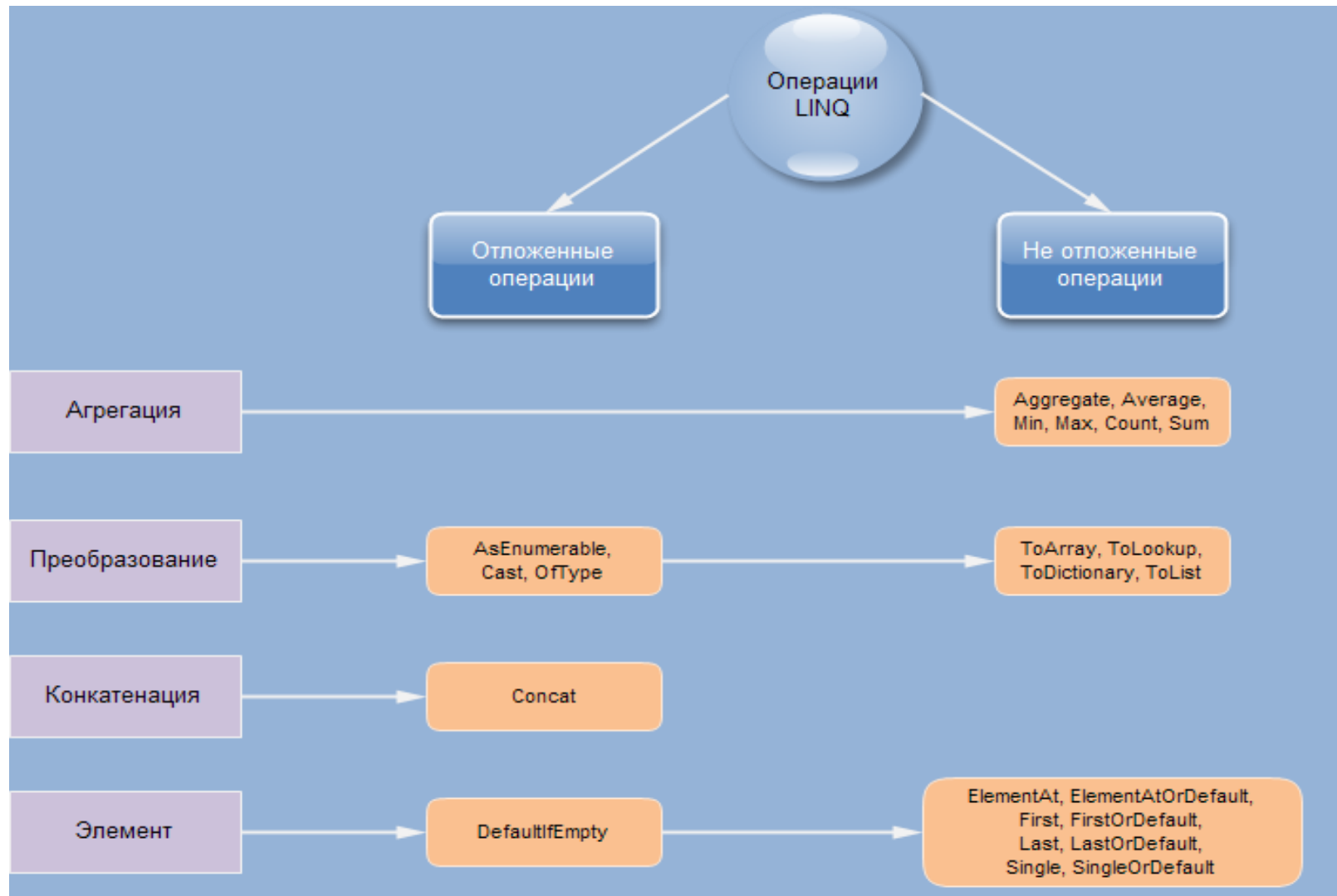
Выполняем запрос

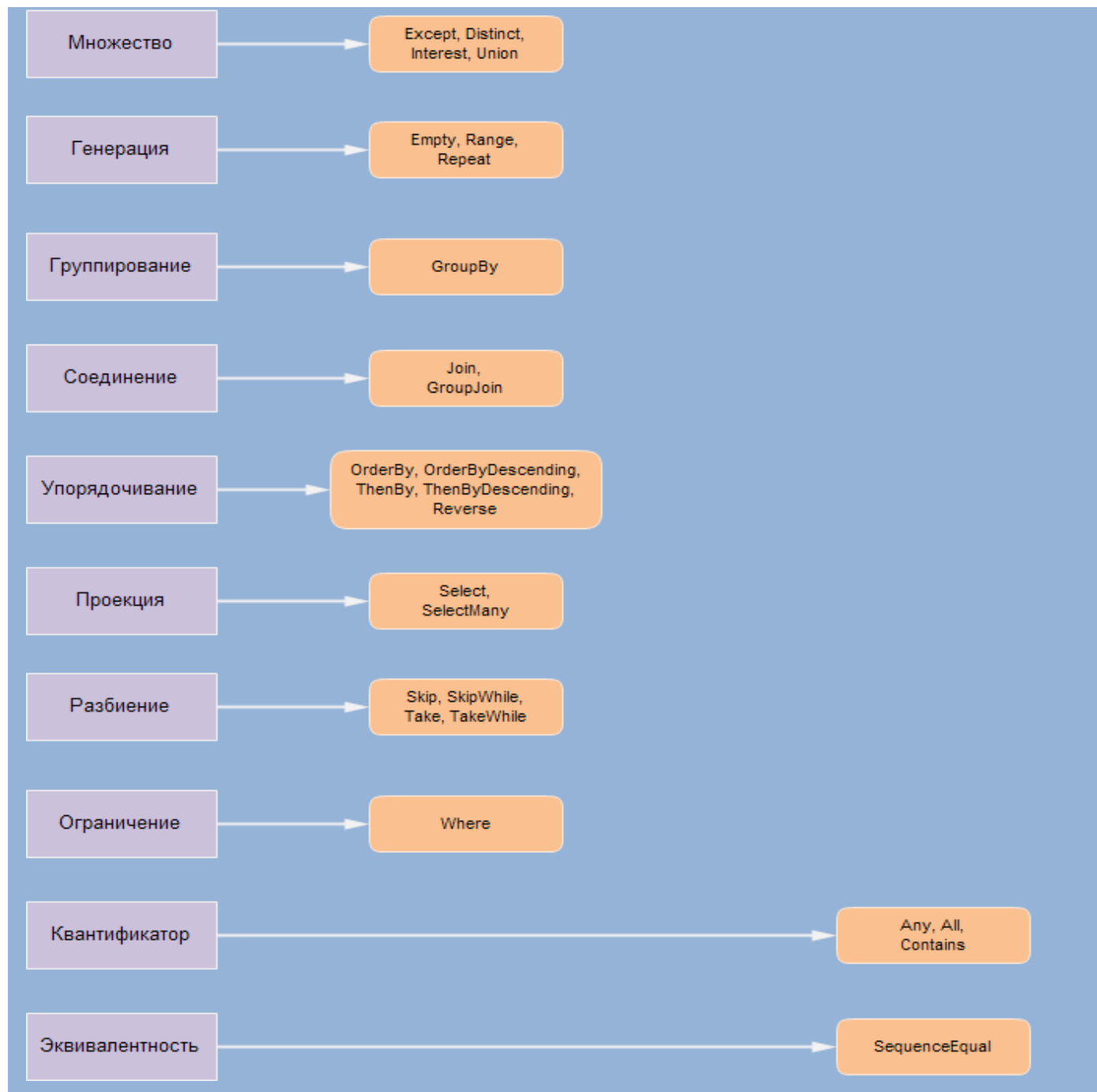
```
foreach (var empName in empNames)  
    Console.WriteLine(empName);
```



from – задает источник данных.
select – проецирует результаты запроса

LINQ включает в себя около 50 стандартных операций запросов, разделяемых на 2 большие группы - **отложенные операции** (выполняются не во время инициализации, а только при их вызове) и **не отложенные операции** (выполняются сразу).





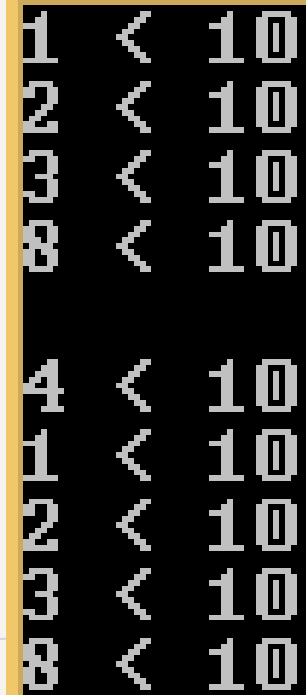
Отложенное выполнение

```
// ОТЛОЖЕННОЕ ВЫПОЛНЕНИЕ
int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

// Получить числа меньше 10.
var subset = from i in numbers where i < 10 select i;

// Оператор LINQ выполняется здесь!
foreach (var i in subset)
    Console.WriteLine("{0} < 10", i);
Console.WriteLine();

// Изменить некоторые данные в массиве.
numbers[0] = 4;
// Оператор LINQ снова выполняется!
foreach (var j in subset)
    Console.WriteLine("{0} < 10", j);
Console.WriteLine();
```




```
1 < 10
2 < 10
3 < 10
8 < 10

4 < 10
1 < 10
2 < 10
3 < 10
8 < 10
```

Немедленное выполнение

```
int[] numbers2 = { 10, 20, 30, 40, 1, 2, 3, 8 };  
// Оператор LINQ выполняется здесь!  
var subset2 = (from i in numbers2 where i < 10 select i).ToArray();  
  
foreach (var i in subset2)  
    Console.WriteLine("{0} < 10", i);  
Console.WriteLine();  
  
// Изменить некоторые данные в массиве.  
numbers2[0] = 4;  
  
foreach (var j in subset2)  
    Console.WriteLine("{0} < 10", j);  
Console.WriteLine();
```

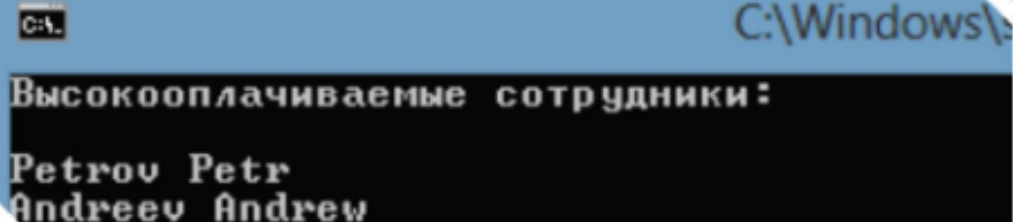


```
1 < 10  
2 < 10  
3 < 10  
8 < 10  
  
1 < 10  
2 < 10  
3 < 10  
8 < 10
```

В выражении **where** мы можем отфильтровать данные согласно нашим условиям.

```
var query = from emp in employees
             where emp.Salary > 100000
             select emp.LastName + " " + emp.FirstName;
```

```
foreach (string emp in query) Console.WriteLine(emp);
```



```
C:\>
Высокооплачиваемые сотрудники:
Petrov Petr
Andreev Andrew
```

where – фильтр данных.

Фильтрация и сортировка данных

```
Console.WriteLine(" Запрос на выборку ");  
var list1 = from employee in Empls where employee.Salary > 10000  
            select employee;
```

```
Console.WriteLine(" Запрос на выборку со сложным критерием ");  
var list2 = from p in Empls where p.Salary > 10000 &&  
            p.FirstName.StartsWith("A") select p;
```

```
Console.WriteLine(" Запрос на выборку и сортировка по возрастанию");  
var list3 = from p in Empls where p.Salary > 10000  
            orderby p.FirstName select p;
```

```
Console.WriteLine(" Запрос на выборку и сортировка по убыванию");  
var list4 = from p in Empls where p.Salary > 10000  
            orderby p.FirstName, p.Salary descending select p;
```

Сортировка по двум
значениям

Language Integrated Query

Создание анонимного типа, для удобного представления результатов выборки

ID	FirstName	LastName	Salary	StartDate
1	Ivan	Ivanov	94 000	1/4/1992
2	Petr	Petrov	123 000	12/3/1985
3	Andrew	Andreev	100 000	1/12/2005

```
var topNames =  
    from emp in employees  
    where emp.Salary > 100000  
    select  
        new {  
            FirstN = emp.FirstName,  
            LastN = emp.LastName };
```



FirstN	LastN
Petr	Petrov
Andrew	Andreev

Выполнение LINQ запроса:

```
foreach (var item in topNames)  
    Console.WriteLine("{0} {1}", item.LastN, item.FirstN);
```

Создание нового типа данных (порекция)

```
Console.WriteLine("Запрос на выборку только фамилий ");  
var list5 = from p in Empls where p.Salary > 10000  
            select p.LastName;  
  
Console.WriteLine("Запрос на выборку и проецирование нового типа");  
var list6= from p in Empls where p.Salary > 10000  
           select new { p.LastName, p.StartDate };  
foreach (var item in list6)  
    Console.WriteLine(item.LastName+": "+item.StartDate);
```

ID	FirstName	LastName	Salary	StartDate
1	Ivan	Ivanov	94 000	1/4/1992
2	Petr	Petrov	123 000	12/3/1985
3	Andrew	Andreev	100 000	1/12/2005

```
var topNames =
    from emp in employees
    where emp.Salary > 100000
    orderby emp.FirstName, emp.LastName descending
    select
        new {
            FirstN = emp.FirstName,
            LastN = emp.LastName };

```



FirstN	LastN
Andrew	Andreev
Petr	Petrov

Выполнение LINQ запроса:

```
foreach (var item in topNames)
    Console.WriteLine("{0} {1}", item.LastN, item.FirstN);

```

Агрегатные операции

```
List<Employee> Empls = CreateList();  
// СУММА  
decimal Summa = (from p in Empls select p.Salary).Sum();  
// СРЕДНЕЕ ЗНАЧЕНИЕ  
decimal Average = (from p in Empls select p.Salary).Average();  
// МАКСИМАЛЬНОЕ ЗНАЧЕНИЕ  
decimal Max = (from p in Empls select p.Salary).Max();  
// МИНИМАЛЬНОЕ ЗНАЧЕНИЕ  
decimal Min = (from p in Empls select p.Salary).Min();  
// КОЛИЧЕСТВО ЗАПИСЕЙ  
int Count = (from p in Empls where p.Salary>10000  
                                     select p.Salary).Count();  
// ВЗЯТЬ 2 ПЕРВЫХ ЭЛЕМЕНТА  
var FirstTwo = (from p in Empls select p).Take(2);  
foreach (var item in FirstTwo)  
    Console.WriteLine(item);  
  
// ПРОПУСТИТЬ 2 ПЕРВЫХ ЭЛЕМЕНТА  
var AfterTwo = (from p in Empls select p).Skip(2);  
foreach (var item in AfterTwo)  
    Console.WriteLine(item);
```

Операции объединения

```
List<string> Subjects1 = new List<String> { "C#", "WPF", "ADO.NET" };
List<string> Subjects2 = new List<String> { "C++", "C#", "WPF" };

// Метод Except возвращает разность между двумя коллекциями
var subDiff = (from c in Subjects1 select c)
               .Except(from c2 in Subjects2 select c2);
// Выводит ADO.NET (из 1-ой вычитает 2-ую)

// Метод Intersect возвращает общие элементы коллекций
var subCommon = (from c in Subjects1 select c)
                 .Intersect(from c2 in Subjects2 select c2);
// Выводит C# и WPF

// Метод Union возвращает объединение двух коллекций");
var subUnion = (from c in Subjects1 select c)
                .Union(from c2 in Subjects2 select c2);
// Выводит всё без повторов

// Метод Concat возвращает содержимое двух коллекций"
var subConcat = (from c in Subjects1 select c)
                 .Concat(from c2 in Subjects2 select c2);
// Выводит всё с повторами
```

Группировка данных

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

```
var query = from x in numbers  
            group x by x % 2;
```



Key	Values
0	{ 0, 2, 4, 6, 8 }
1	{ 1, 3, 5, 7, 9 }

Выполнение LINQ запроса:


```
foreach (var group in query)  
{  
    Console.WriteLine("mod == {0}", group.Key);  
    foreach (var number in group)  
        Console.WriteLine("{0}, ", number);  
}
```

```
public interface IGrouping<out TKey, out TElement>  
    : IEnumerable<TElement>, IEnumerable  
{  
    TKey Key { get; }  
}
```

Language Integrated Query

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

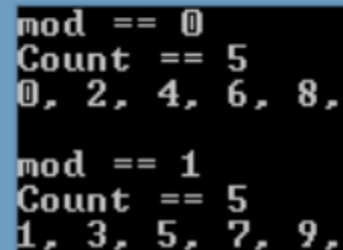
```
var query =  
    from x in numbers  
    group x by x % 2 into partition  
    select  
    new { Key = partition.Key,  
          Count = partition.Count(),  
          Group = partition };
```



Key	Count	Values
0	5	{ 0, 2, 4, 6, 8 }
1	5	{ 1, 3, 5, 7, 9 }

Выполнение LINQ запроса:

```
foreach (var item in query)  
{  
    Console.WriteLine("mod == {0}", item.Key);  
    Console.WriteLine("Count == {0}", item.Count);  
    foreach (var number in item.Group)  
        Console.Write("{0}, ", number);  
    Console.WriteLine("\n");  
}
```



```
C:\>  
mod == 0  
Count == 5  
0, 2, 4, 6, 8,  
  
mod == 1  
Count == 5  
1, 3, 5, 7, 9,
```


Группировка (исходные данные)

```
public class Employee
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public decimal Salary { get; set; }
    public string Company { get; set; }
    public DateTime StartDate { get; set; }
    public override string ToString()
    {
        return String.Format("{0} {1}: {2} {3:C} {4}",
                               FirstName, LastName, Company, Salary, StartDate);
    }
}
```

```
static List<Employee> CreateList()
{
    return new List<Employee>
    {
        new Employee
        {
            FirstName = "Ivan", Company="ItProject", LastName = "Ivanov",
            Salary = 15000, StartDate = DateTime.Parse("1/4/1992")
        },
        new Employee
        {
            FirstName = "Petr", Company="EffectSoft", LastName = "Petrov",
            Salary = 120000, StartDate = DateTime.Parse("12/3/1985")
        },
        new Employee
        {
            FirstName = "Andrew", Company="ItProject", LastName = "Andreev",
            Salary = 100000, StartDate = DateTime.Parse("1/4/1992")
        }
    };
}
```

Группировка

```
// Группировка данных по компании
var GroupCompany= from p in EmpIs
    group p by p.Company // указываем критерий (ключ) группировки
    into gr               // сохранение сгруппированных данных в gr
    select                // формирование анонимного типа
    new {
        // создаем свойство и указываем ключ группировки
        Company = gr.Key,
        // суммируем значения в группе
        Salary = gr.Sum(s => s.Salary) };

```

```
{ Company = ItProject, salary = 115000 }
{ Company = EffectSoft, salary = 120000 }
```

```
// Группировка по нескольким ключам
var GroupCompany3 = from p in EmpIs
    group p by new { p.Company, p.StartDate }
    into gr
    select
    new { Company = gr.Key.Company,
        Date = gr.Key.StartDate, Salary = gr.Sum(p => p.Salary) };

```

```
{ Company = ItProject, Date = 01.04.1992 0:00:00, salary = 115000 }
{ Company = EffectSoft, Date = 12.03.1985 0:00:00, salary = 120000 }
```

Группировка (создание строго типизированной коллекции)

```
// Группировка данных по компании с формированием коллекции
List<InfoCompany> GroupCompany2 = (from p in Empls
    group p by p.Company
    into gr select
    new InfoCompany
    { Company = gr.Key, Salary = gr.Sum(p => p.Salary)}).ToList();
```

```
public class InfoCompany
{
    public string Company { get; set; }
    public decimal Salary { get; set; }

    public override string ToString()
    {
        return String.Format("{0} {1:C} ", Company, Salary);
    }
}
```

Language Integrated Query

Любой LINQ-запрос, трансформируется в последовательность вызовов расширяющих методов

```
var query = from employee in employees  
            where employee.Salary > 100000  
            select employee;
```



```
var topEmployees = employees.Where(emp => emp.Salary > 100000);
```

Language Integrated Query

Любой LINQ-запрос, трансформируется в последовательность вызовов расширяющих методов

```
var query = from employee in employees
             where employee.Salary > 100000
             orderby employee.LastName, employee.FirstName
             select new {
                 LastName = employee.LastName,
                 FirstName = employee.FirstName};
```

```
var query = employees
             .Where(emp => emp.Salary > 100000)
             .OrderBy(emp => emp.LastName)
             .ThenBy(emp => emp.FirstName)
             .Select(emp => new
             {
                 LastName = emp.LastName,
                 FirstName = emp.FirstName
             });
```

Группировка с применением расширяющих методов

// Группировка данных по компании

```
var GroupCompany4 = EmpIs.GroupBy(p => p.Company).  
    Select(gr => new  
        { Company = gr.Key,  
          Salary   = gr.Sum(s => s.Salary)  
        });
```

```
List<InfoCompany> GroupCompany5 = (EmpIs.GroupBy(p => p.Company).  
    Select(gr => new InfoCompany  
        { Company = gr.Key,  
          Salary   = gr.Sum(s => s.Salary)  
        })).ToList();
```

```
var GroupCompany6 = EmpIs.  
    GroupBy(p => new {p.Company, p.StartDate}).  
    Select(gr => new  
        { Company = gr.Key.Company,  
          Date     = gr.Key.StartDate,  
          Salary   = gr.Sum(s => s.Salary)  
        });
```

Расширяющие методы

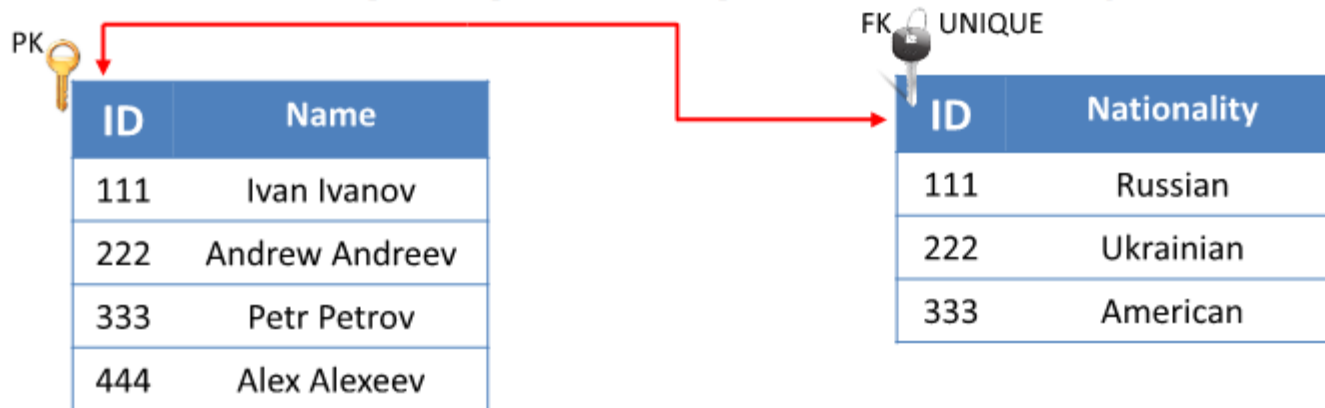
- Компилятор C# на этапе компиляции транслирует все операции C# LINQ в вызовы **методов класса Enumerable**.
- Большинство методов Enumerable прототипированы так, чтобы принимать в качестве аргументов **делегаты**, в частности, делегат по имени **Func<>**.

```
// Перегруженная версия метода Enumerable.Where<T>()  
public static IEnumerable<TSource> Where<TSource>  
    (this IEnumerable<TSource> source,  
     System.Func<TSource, bool> predicate)
```

- Делегат Func<> представляет шаблон метода с набором до 16 аргументов и возвращаемым значением.

```
// Различные форматы делегата Func<>.  
public delegate TResult Func<T1, T2, TResult>(T1 arg1, T2 arg2)  
public delegate TResult Func<T2, TResult>(T1 arg1)
```

Language Integrated Query



```
var query =  
    from emp in employees  
    join n in empNationalities  
    on emp.Id equals n.Id  
    orderby n.Nationality descending  
    select  
        new {  
            Id = emp.Id,  
            Name = emp.Name,  
            Nationality = n.Nationality};
```

ID	Name	Nationality
222	Andrew Andreev	Ukrainian
111	Ivan Ivanov	Russian
333	Petr Petrov	American

Объединение данных (исходные данные)

```
static List<Car> CreateCars()
{
    return new List<Car>
    {
        new Car(){ID=1, Name="BMW", Price=25000, DriverID=1},
        new Car(){ID=2, Name="Ford", Price=1500, DriverID=2},
        new Car(){ID=3, Name="Opel", Price=1900, DriverID=2},
        new Car(){ID=4, Name="Audi", Price=2200, DriverID=3},
    };
}

static List<Driver> CreateDrivers()
{
    return new List<Driver>
    {
        new Driver(){ID=1, Name="Alex", Phone="+555-135246"},
        new Driver(){ID=2, Name="Ivan", Phone="+555-123456"},
        new Driver(){ID=3, Name="Inna", Phone="+555-654321"},
        new Driver(){ID=4, Name="Bobr", Phone="+555-246135"},
    };
}
```

Объединение данных (операторы запросов + методы расширения)

// Получение данных из двух коллекций с использованием операторов запроса

```
var Info1 = from c in Cars           // 1-ая коллекция
            join d in Drivers       // 2-ая коллекция
            on c.DriverID equals d.ID // связь
            select new              // формирование анонимного типа
            {
                Name = d.Name,
                Price = c.Price
            };
```

// Получение данных из двух коллекций с расширяющих методов

```
var Info2 = Cars.Join(Drivers,
    c => c.DriverID,
    d => d.ID,
    (c, d) => new { car = c, driver = d }).
    Select(p => new
    {
        Name = p.driver.Name,
        Price = p.car.Price
    });
```

Работа с необобщенными коллекциями

Проблема: Ни одна из необобщенных коллекций C# из пространства имен System.Collections не реализует IEnumerable<T>

Разница: **Операция Cast** пытается привести все элементы в коллекции к указанному типу, помещая их в выходную последовательность, если в коллекции есть объект типа, который не может быть приведен к указанному генерируется исключение.

Операция OfType пытается поместить в выходную последовательность только те элементы, которые могут быть приведены к указанному типу.

```
// Унаследованная коллекция
```

```
ArrayList arr = new ArrayList();  
arr.Add("one");  
arr.Add("two");  
arr.Add("three");
```

```
// Приведем коллекцию к типу IEnumerable с помощью LINQ
```

```
IEnumerable<string> numbers1 = arr.Cast<string>()  
                                .Where(n => n.Length < 4);
```

```
// То же самое с помощью операции OfType
```

```
IEnumerable<string> numbers2 = arr.OfType<string>()  
                                .Where(n => n.Length < 4);
```

Развитие LINQ

Оператор «let» - представляет новый локальный идентификатор, на который можно ссылаться в остальной части запроса.

Представляет собой локальную переменную видимую **только внутри выражения запроса**.

```
var query = from emp in employees
             let fullName = emp.FirstName + " " + emp.LastName
             orderby fullName descending
             select fullName;

foreach (var person in query)
    Console.WriteLine(person);
```

Подводим итог:

1. Выражения запросов создаются с использованием различных **операций запросов С#**.
2. Операции запросов - это просто сокращенная нотация вызова **расширяющих методов, определенных в типе System.Linq.Enumerable**.
3. Многие методы Enumerable принимают **в качестве параметров делегаты** (в частности, **Func<>**).
4. Любой метод, ожидающий параметр-делегат, может принимать вместо него **лямбда-выражение**.