

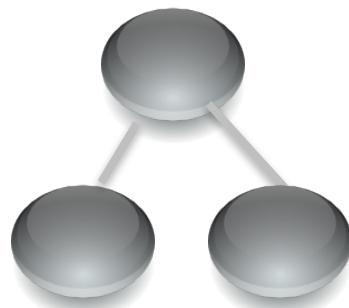


# INTRODUCTION TO LANGUAGE INTEGRATED QUERY (LINQ)

**.NET & JS LAB, RD BELARUS**

Anzhelika Kravchuk

# Что такое LINQ?



**Объекты**



**РСУБД**



**XML**

Разнообразие данных обуславливает проблемы при работе с ними – обычно используемые подходы в значительной мере определяются их типом, структурой и источником, а универсальные механизмы фактически отсутствуют

# Что такое LINQ?

C#

VB.NET

Другие языки .NET Framework

LINQ

Поставщик LINQ

Поставщики LINQ ADO.NET

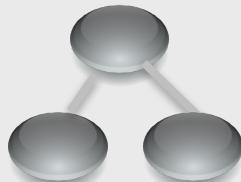
LINQ to  
Objects

LINQ to  
DataSets

LINQ to  
SQL

LINQ to  
Entities

LINQ to  
XML



Объекты



РСУБД



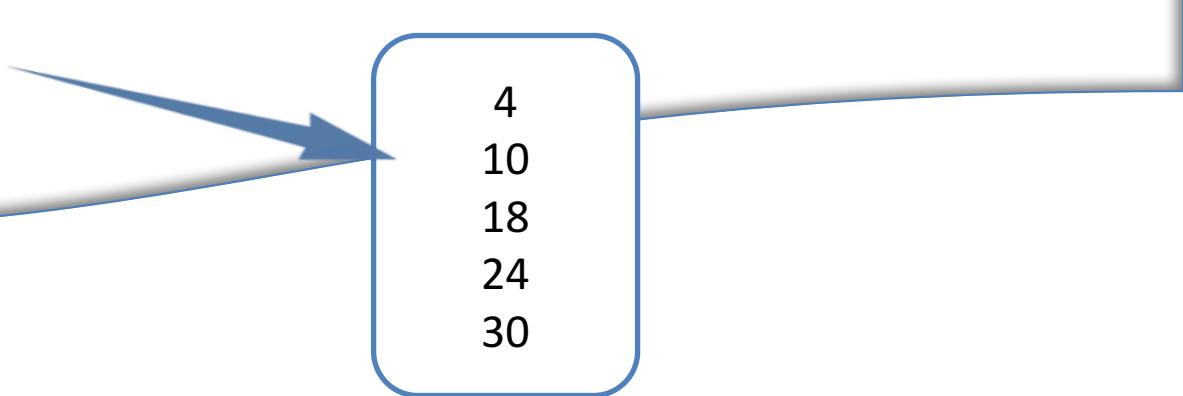
XML

Язык интегрированных запросов (LINQ) представляет собой набор языковых и платформенных средств для описания **структурированных, безопасных в отношении типов** запросов к локальным коллекциям объектов и удаленным источникам данных

## Введение в запросы LINQ

Выборка и формирование упорядоченного списка элементов массива традиционным способом

```
int[] numbers = { 10, 5, 13, 18, 4, 24, 65, 41, 30 };
List<int> evens = new List<int>();
foreach (var number in numbers)
{
    if (number % 2 == 0)
        evens.Add(number);
}
evens.Sort();
foreach (int number in evens)
{
    Console.WriteLine(number);
}
```



## Введение в запросы LINQ

Выборка и формирование упорядоченного списка элементов массива с применением LINQ

```
int[] numbers = { 10, 5, 13, 18, 4, 24, 65, 41, 30 };
var evens =
    from number in numbers
    where (number % 2) == 0
    orderby number
    select number;

foreach (int number in evens)
{
    Console.WriteLine(number);
}
```

```
from number in numbers
where (number % 2) == 0
orderby number
select number;
```

4  
10  
18  
24  
30

## Введение в запросы LINQ

Базовые единицы данных LINQ – последовательности и элементы

Последовательность – это любой объект, реализующий интерфейс `IEnumerable<T>`

Элемент – элемент внутри последовательности

```
string[ ] names = { "Tom", "Dick", "Harry" };
```

Локальная последовательность – последовательность, представляющая локальную коллекцию объектов в памяти

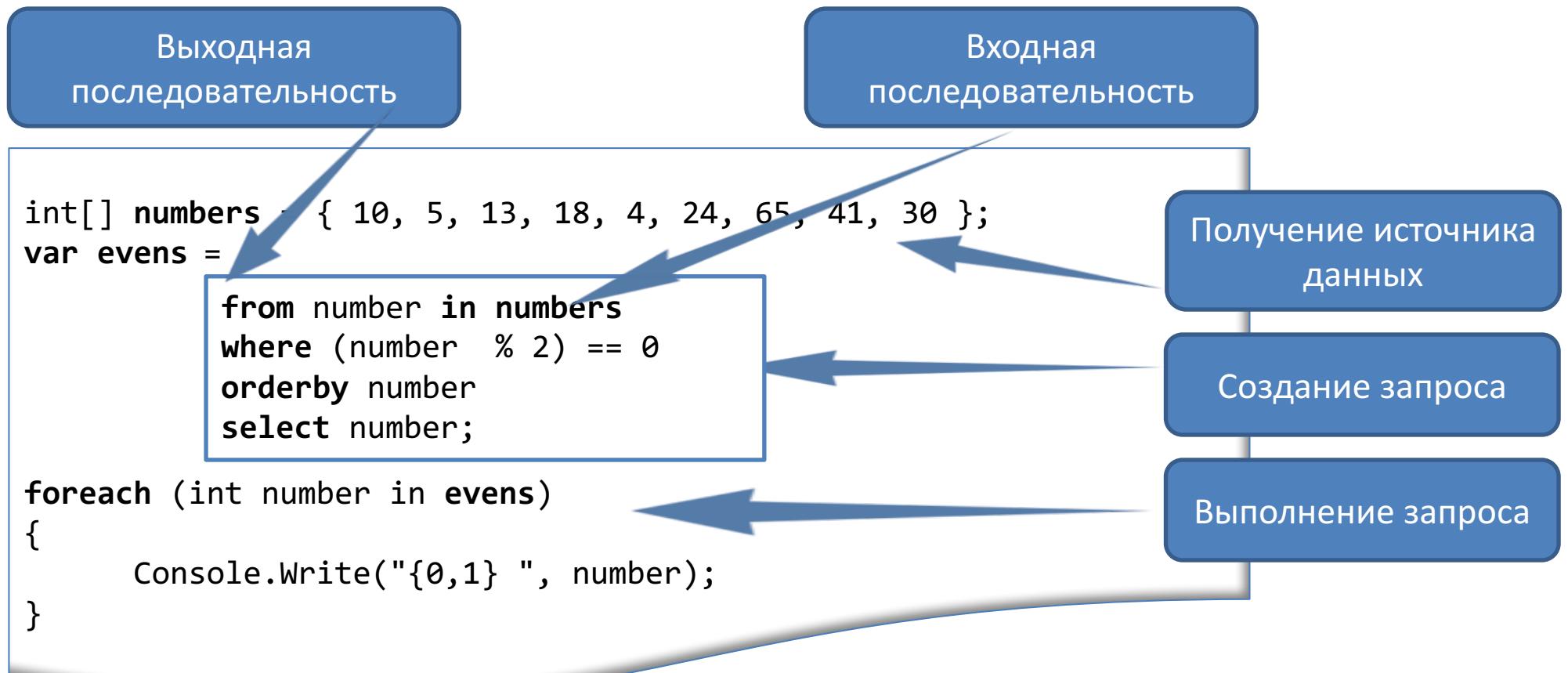
Операция запроса – это метод , трансформирующий входную последовательность

В классе `Enumerable` (`System.Linq`) имеется около 40 операций запросов (реализованы в виде методов расширения) - стандартные операции запроса

Запрос представляет собой выражение, которое при перечислении трансформирует последовательности с помощью операций запросов

Запросы, оперирующие на локальных последовательностях, называются локальными запросами или запросами `LINQ to Object` (`LINQ2Object`, `L2O`)

# Введение в запросы LINQ



Во многих случаях лежащий в основе тип не очевиден и даже напрямую недоступен в коде (или иногда генерируется во время компиляции)

Типы, которые поддерживают `IEnumerable<T>` (`IQueryable<T>`), называются запрашиваемыми типами

## Введение в запросы LINQ

LINQ также поддерживает последовательности, которые могут наполняться из удаленных источников, таких как SQL Server

```
// Northwind inherits from System.Data.Linq.DataContext.  
Northwind nw = new Northwind(@"northwind.mdf");
```

```
var companyNameQuery =  
    from cust in nw.Customers  
    where cust.City == "London"  
    select cust.CompanyName;  
  
foreach (var customer in companyNameQuery)  
{  
    Console.WriteLine(customer);  
}
```

Переменная  
запроса

Получение источника  
данных

Создание запроса

Выполнение запроса

Выражение  
запроса

Поддерживаются посредством соответствующего набора стандартных операций запросов в классе `Queryable`

## Введение в запросы LINQ

Запрос представляет собой выражение, которое при перечислении трансформирует последовательности с помощью операций запросов (метод расширения)

```
string[] names = { "Tom", "Dick", "Harry" };  
  
IEnumerable<string> filteredNames =  
    System.Linq.Enumerable.Where (names, n => n.Length >= 4);  
  
foreach (string n in filteredNames)  
    Console.Write (n + "|");
```

using System.Linq;

Dick|Harry|

```
IEnumerable<string> filteredNames = names.Where(n => n.Length >= 4);
```

Синтаксис операций  
запросов (fluent)

```
IEnumerable<string> filteredNames = from n in names  
                                         where n.Contains ("a")  
                                         select n;
```

Синтаксис выражений  
запросов

Синтаксис построения запросов, при котором используются методы расширения и лямбда-выражения называется **текущим (fluent)** синтаксисом

Текущий синтаксис позволяет формировать цепочки операций запросов

## Текущий синтаксис (Fluent Syntax)

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

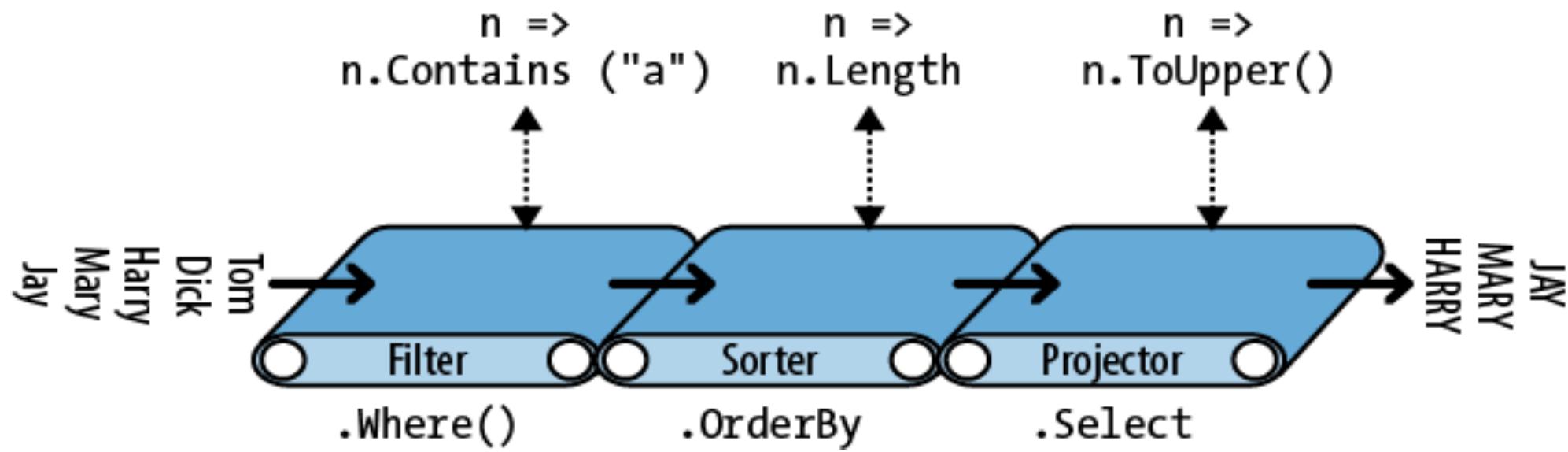
IQueryable<string> query = names
    .Where (n => n.Contains ("a"))
    .OrderBy (n => n.Length)
    .Select (n => n.ToUpper());

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

JAY

MARY

HARRY



## Текущий синтаксис (Fluent Syntax)

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

public static IEnumerable<TSource> OrderBy<TSource,TKey>
(this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)

public static IEnumerable<TResult> Select<TSource,TResult>
(this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

## Текущий синтаксис (Fluent Syntax)

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IQueryable<string> query = names
    .Where(n => n.Contains("a"))
    .OrderBy(n => n.Length)
    .Select(n => n.ToUpper());

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



JAY  
MARY  
HARRY

```
IEnumerable<string> filtered = names.Where(n => n.Contains("a"));
IQueryable<string> sorted = filtered.OrderBy(n => n.Length);
IQueryable<string> finalQuery = sorted.Select(n => n.ToUpper());
```

Harry      **Filtered**  
Mary  
Jay

Jay      **Sorted**  
Mary  
Harry

JAY      **FinalQuery**  
MARY  
HARRY

## Текущий синтаксис (Fluent Syntax)

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };  
  
IQueryable<string> query = names  
.Where (n => n.Contains ("a"))  
.OrderBy (n => n.Length)  
.Select (n => n.ToUpper());
```

Без методов расширения запрос теряет свою текущесть

```
IEnumerable<string> query =  
Enumerable.Select (  
Enumerable.OrderBy (  
Enumerable.Where (  
    names, n => n.Contains ("a"))  
, n => n.Length  
, n => n.ToUpper())  
);
```

## Составление лямбда-выражений

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

```
IEnumerable<string> query = names  
    .Where (n => n.Contains ("a"))  
    .OrderBy (n => n.Length)  
    .Select (n => n.ToUpper());
```

Лямбда-выражение, которое принимает значений и возвращает bool, называется предикатом

Лямбда-выражение в операции запроса всегда работает на индивидуальных элементах во входной последовательности, а не на последовательности в целом

Операция запроса вычисляет лямбда-выражение по запросу – обычно один раз на элемент во входной последовательности

Лямбда-выражение позволяет помещать собственную логику внутрь операций запроса

## Составление лямбда-выражений

```
public static IEnumerable<TResult> Select<TSource, TResult>
    (this IEnumerable<TSource> source, Func<TSource, TResult> selector)
```

```
public static IEnumerable<TSource> OrderBy<TSource, TKey>
    (this IEnumerable<TSource> source, Func<TSource, TKey> keySelector)
```

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

Generic type letter	Meaning
TSource	Element type for the input sequence
TResult	Element type for the output sequence—if different from TSource
TKey	Element type for the key used in sorting, grouping, or joining

## Естественный порядок

Исходный порядок элементов входной последовательности является важным в LINQ.  
Некоторые операции запросов полагаются на это поведение – Take, Skip, Reverse

```
int[] numbers = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take (3); // { 10, 9, 8 }
```

```
int[] numbers = { 10, 9, 8, 7, 6 };
IEnumerable<int> lastTwo = numbers.Skip (3); // { 7, 6 }
```

```
int[] numbers = { 10, 9, 8, 7, 6 };
IEnumerable<int> reversed = numbers.Reverse(); // { 6, 7, 8, 9, 10 }
```

## Другие операции

Не все операции возвращают последовательность

```
int[] numbers = { 10, 9, 8, 7, 6 };
int firstNumber = numbers.First(); // 10
int lastNumber = numbers.Last(); // 6
int secondNumber = numbers.ElementAt(1); // 9
int secondLowest = numbers.OrderBy(n=>n).Skip(1).First(); // 7
```

```
//операции агрегирования
int count = numbers.Count(); // 5;
int min = numbers.Min(); // 6;
```

```
//квантификаторы
bool hasTheNumberNine = numbers.Contains (9); // true
bool hasMoreThanZeroElements = numbers.Any(); // true
bool hasAnOddElement = numbers.Any (n => n % 2 != 0); // true
```

## Выражения запросов

Язык C# предоставляет синтаксическое сокращение для записи запросов LINQ, называемое выражения запросов (оказали влияние генераторы списков – list generation – из языков функционального программирования LISP, Haskell )

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IQueryable<string> query =
from n in names
where n.Contains ("a")      // Filter elements
orderby n.Length            // Sort elements
select n.ToUpper();         // Translate each element (project)

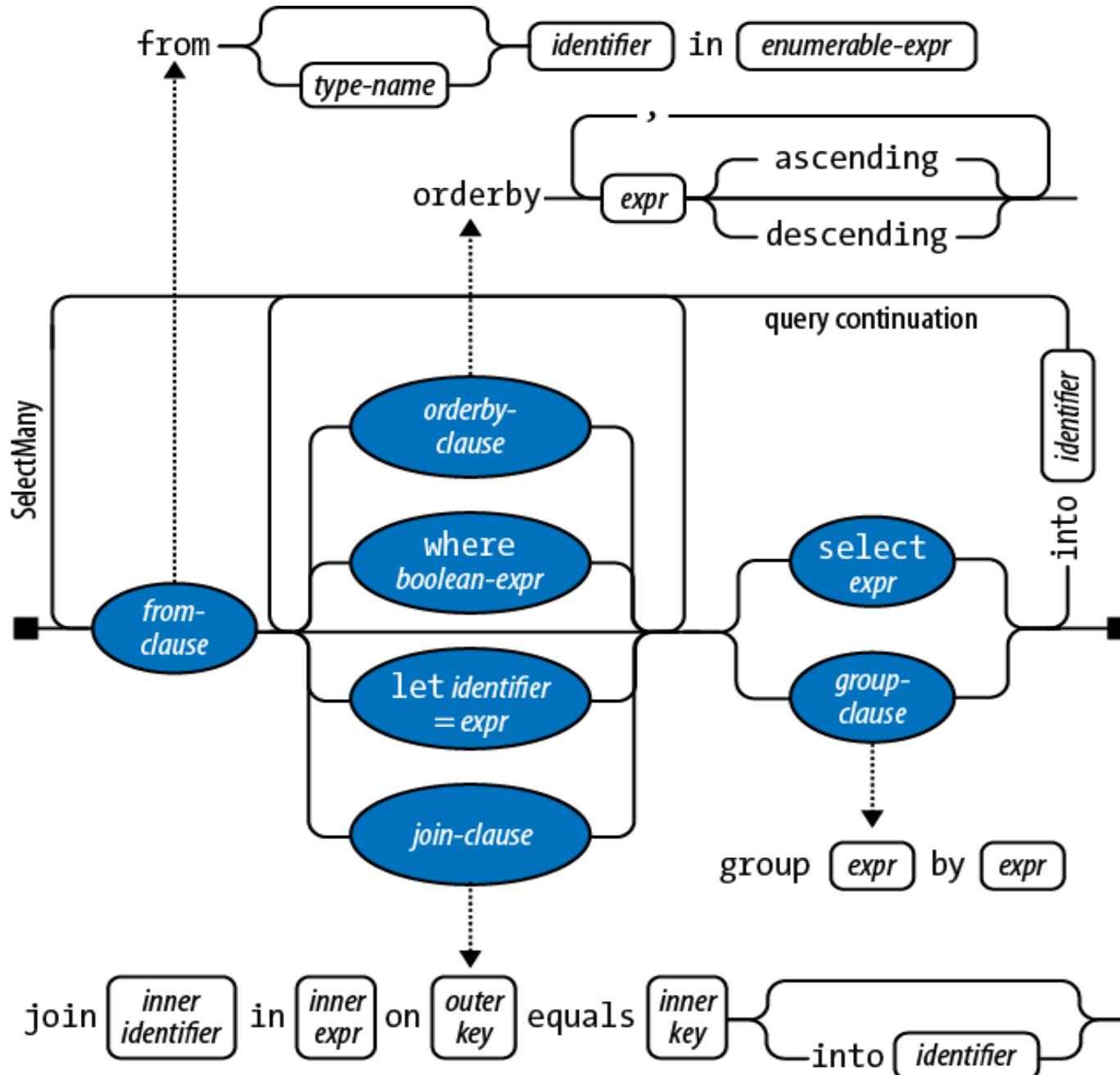
foreach (var element in query)
    Console.WriteLine(element);
```



JAY  
MARY  
HARRY

Компилятор обрабатывает выражение запроса путем его трансляции в текущий синтаксис

## Выражения запросов



## Выражения запросов. Переменные диапазона

```
from n in names           // n is our range variable
where n.Contains ("a")    // n = directly from the array
orderby n.Length          // n = subsequent to being filtered
select n.ToUpper()        // n = subsequent to being sorted
```

Переменная  
диапазона

```
names.Where (n => n.Contains ("a"))      // Locally scoped n
      .OrderBy (n => n.Length)             // Locally scoped n
      .Select (n => n.ToUpper())           // Locally scoped n
```

Выражения запросов также позволяет вводить новые переменные диапазонов с помощью следующих конструкций

- let
- into
- from (дополнительная конструкция)
- join

## Синтаксис выражений запросов vs. текущий синтаксис

Синтаксис выражений запросов проще для запросов, которые могут содержать в себе любой из следующих аспектов:

- конструкция `let` для введения новой переменной наряду с переменной диапазона
- Операция `SelectMany`, `Join`, `GroupJoin`, за которой следует ссылка на внешнюю переменную диапазона

К запросам, которые используют операции `Select`, `Where`, `OrderBy` одинаково хорошо применимы оба синтаксиса

Для запросов, состоящий из одной операции, текущий синтаксис короче и менее громоздкий

Существует множество операций, для которых не предусмотрены ключевые слова в синтаксисе запросов

Можно смешить текущий синтаксис и синтаксис выражений запросов

## Отложенное выполнение

Важная особенность большинства операций запросов – выполнение не при конструировании, а во время перечисления

```
var numbers = new List<int>();
numbers.Add (1);
IEnumerable<int> query = numbers.Select (n => n * 10);
numbers.Add (2);

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

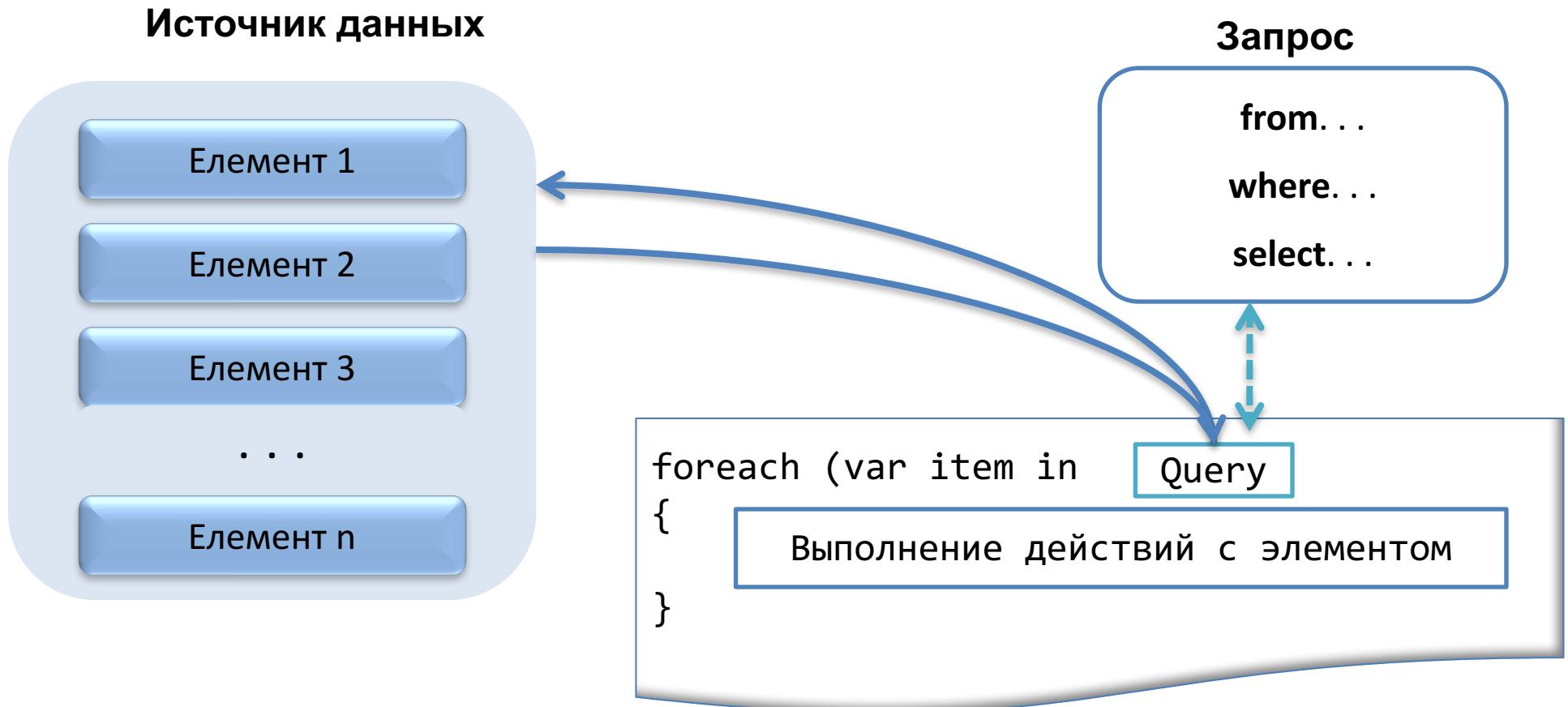
10  
20

```
Action a = () => Console.WriteLine ("Foo");
a(); // Deferred execution!
```

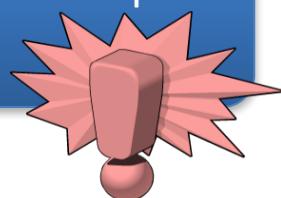
Отложенное выполнение поддерживают все стандартные операции запросов со следующими исключениями:

- операции, которые возвращают одиночный элемент или скалярное значение (First, Count)
- операции преобразования ToArray, ToList, ToDictionary, ToLookup

## Отложенное выполнение



В LINQ выполнение запроса отличается от самого запроса – создание переменной запроса само по себе не связано с получением данных



## Отложенное выполнение. Повторная оценка

Запрос с отложенным выполнением повторно оценивается при перечислении заново

```
var numbers = new List<int>() { 1, 2 };
IEnumerable<int> query = numbers.Select (n => n * 10);
foreach(var temp in query)
    Console.WriteLine(temp);

numbers.Clear();
foreach(var temp in query)
    Console.WriteLine(temp); //Ничего не выводится
```

```
var numbers = new List<int>() { 1, 2 };
IEnumerable<int> timesTen = numbers.Select (n => n * 10).ToList();
foreach(var temp in timesTen)
    Console.WriteLine(temp);

numbers.Clear();
foreach(var temp in timesTen)
    Console.WriteLine(temp);
```

## Захваченные переменные

Если лямбда-выражение запроса захватывает внешние переменные, то запрос будет принимать значения этих переменных на момент выполнения

```
IEnumerable<char> query = "Not what you might expect";
query = query.Where (c => c != 'a');
query = query.Where (c => c != 'e');
query = query.Where (c => c != 'i');
query = query.Where (c => c != 'o');
query = query.Where (c => c != 'u');
foreach(char c in query) //Nt wht y mght xpct
    Console.WriteLine(c);
```

```
IEnumerable<char> query = "Not what you might expect";
string vowels = "aeiou";
for (int i = 0; i < vowels.Length; i++)
    query = query.Where (c => c != vowels[i]);
foreach(char c in query) //IndexOutOfRangeException
    Console.WriteLine(c);
```

## Захваченные переменные

```
IEnumerable<char> query = "Not what you might expect";
string vowels = "aeiou";
for (int i = 0; i < vowels.Length; i++)
{
    char vowel = vowels[i];
    query = query.Where (c => c != vowel);
}
foreach(char c in query) //Nt wht y mght xpct
    Console.WriteLine(c);
```

```
IEnumerable<char> query = "Not what you might expect";

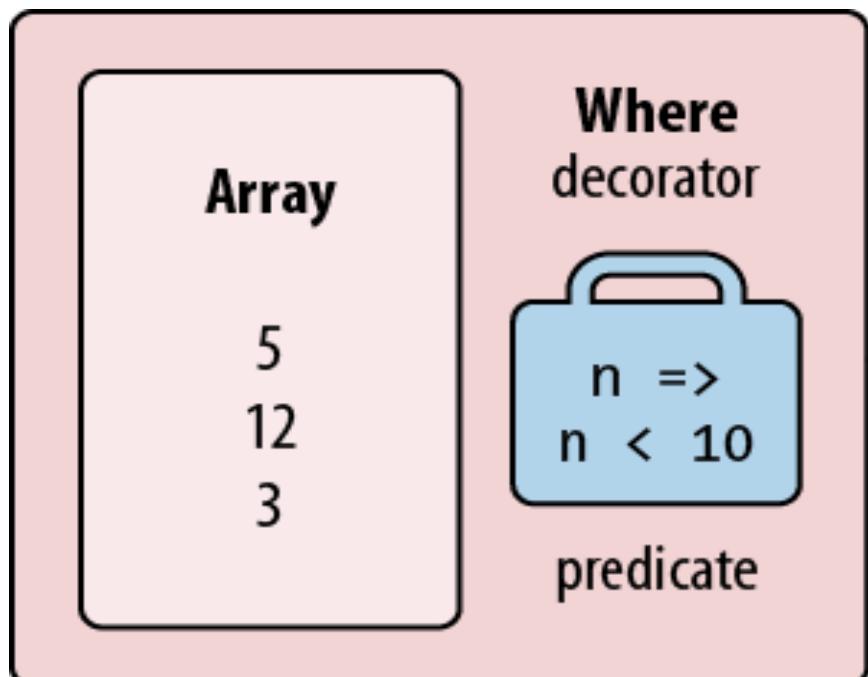
foreach (char vowel in "aeiou") //C# 5.0
    query = query.Where (c => c != vowel);

foreach(char c in query)
    Console.WriteLine(c);
```

## Как работает отложенное выполнение

Операции запросов поддерживают отложенное выполнение, возвращая декораторы последовательности (decorator sequence)

```
IEnumerable<int> lessThanTen = new int[] { 5, 12, 3 }  
.Where (n => n < 10);
```



Реализация декоратора последовательности осуществляется с помощью итератора C#

**lessThanTen**

Если выходная последовательность не подвергается трансформации, декоратор последовательность это просто прокси

## Как работает отложенное выполнение

Для создания собственной операции запроса, реализация декоратора последовательности осуществляется с помощью итератора C#

```
public static IEnumerable<TResult> Select<TSource, TResult>
    (this IEnumerable<TSource> source, Func<TSource, TResult> selector)
{
    foreach (TSource element in source)
        yield return selector (element);
}
```

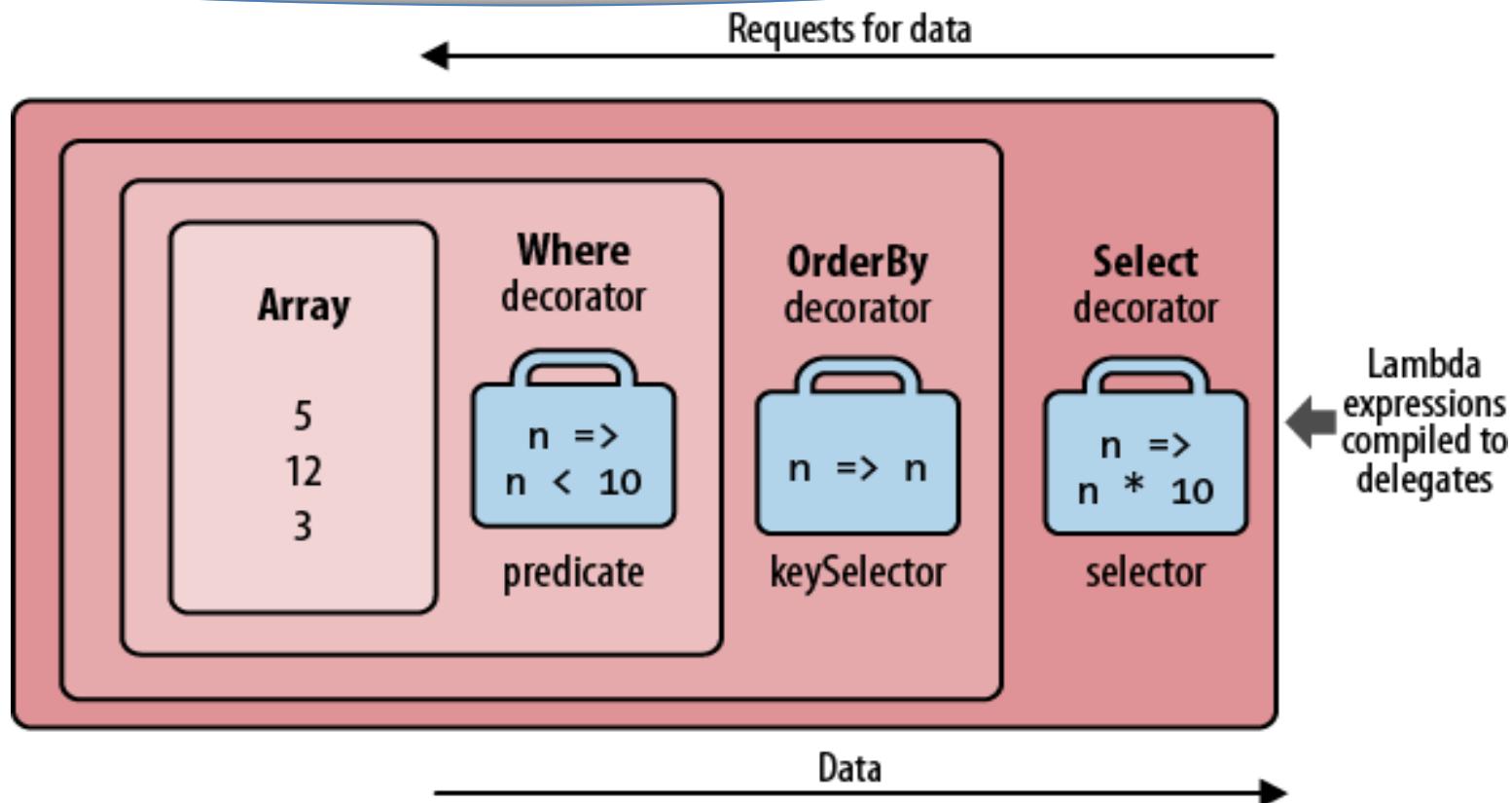
```
public static IEnumerable<TResult> Select<TSource, TResult>
    (this IEnumerable<TSource> source, Func<TSource, TResult> selector)
{
    return new SelectSequence (source, selector);
}
```

Сгенерированный компилятором класс, перечислитель которого инкапсулирует логику из метода итератора

## Как работает отложенное выполнение

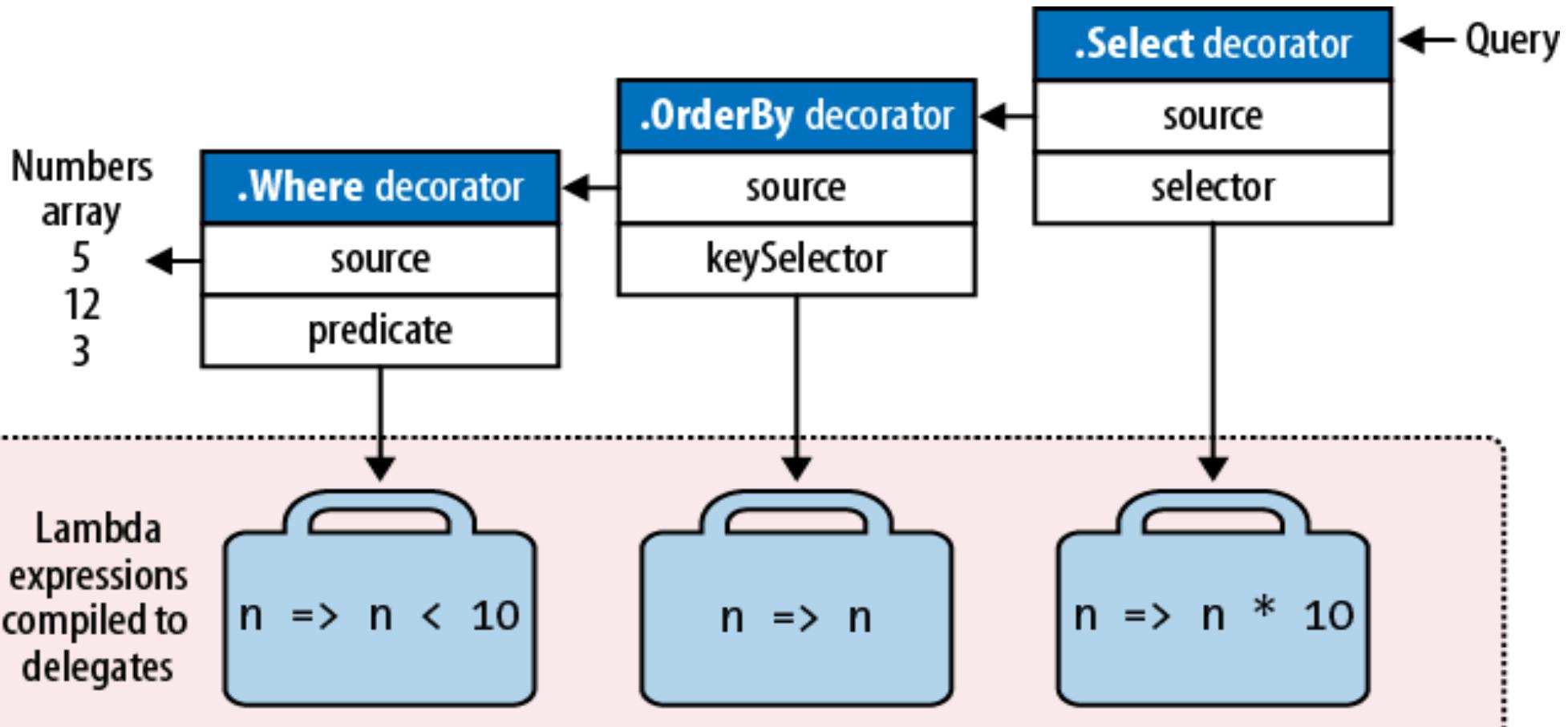
В результате построения цепочки операторов создается иерархия декораторов

```
IEnumerable<int> query = new int[] { 5, 12, 3 }  
    .Where (n => n < 10)  
    .OrderBy (n => n)  
    .Select (n => n * 10);
```



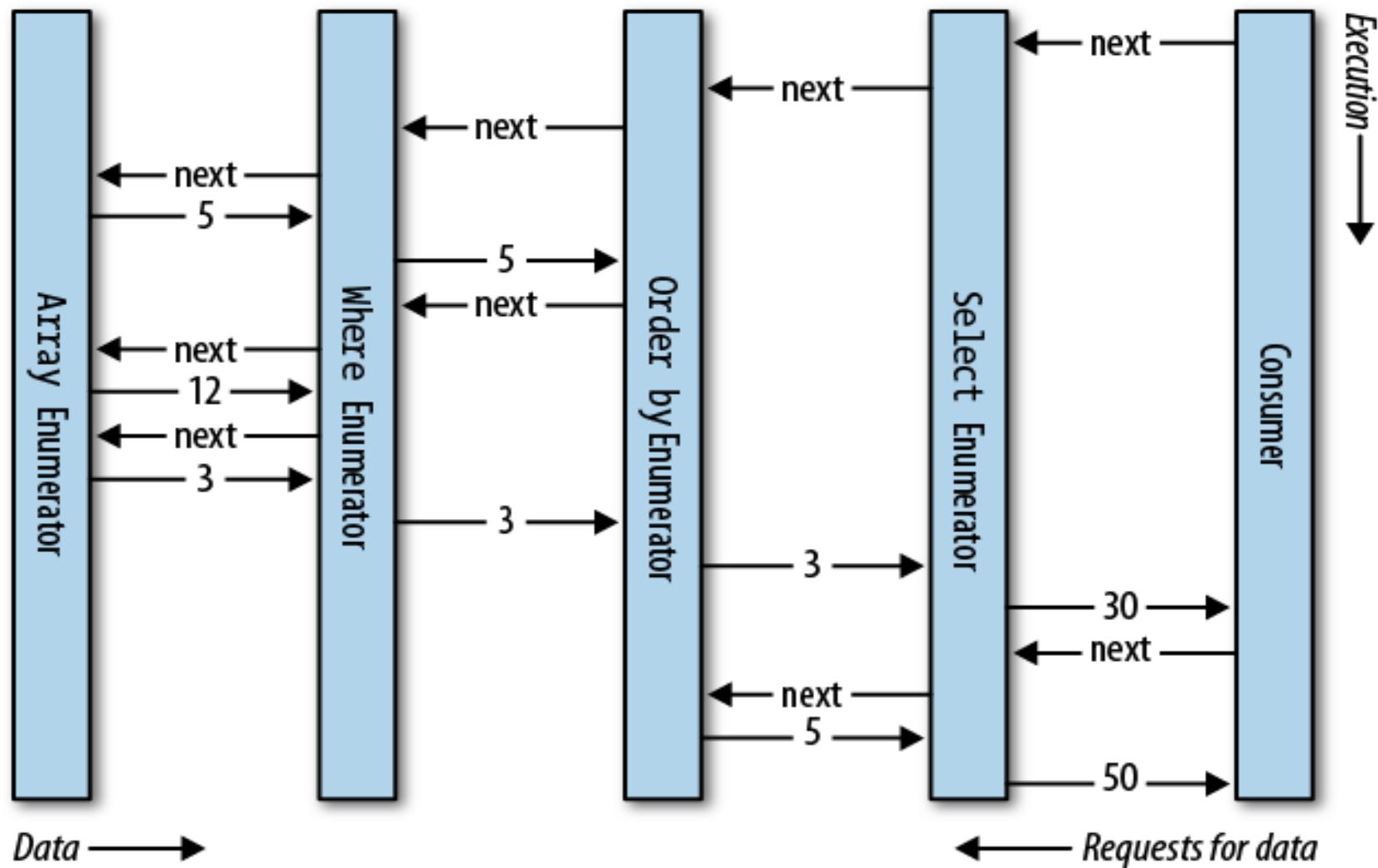
## Как работает отложенное выполнение

В результате построения цепочки операторов создается иерархия декораторов



## Каким образом выполняются запросы

```
foreach (int n in query) Console.WriteLine (n);
```



## Каким образом выполняются запросы

Запрос LINQ – это «ленивая» производственная линия, в которой все конвейерные ленты перемещают элементы только по требованию

Построение запроса конструирует производственную линию со всеми составными частями, но в остановленном состоянии

Когда потребитель запрашивает элемент (выполняет перечисление запроса), активизируется самая правая конвейерная лента, это, в свою очередь, запускает остальные конвейерные ленты – когда требуются элементы входной последовательности

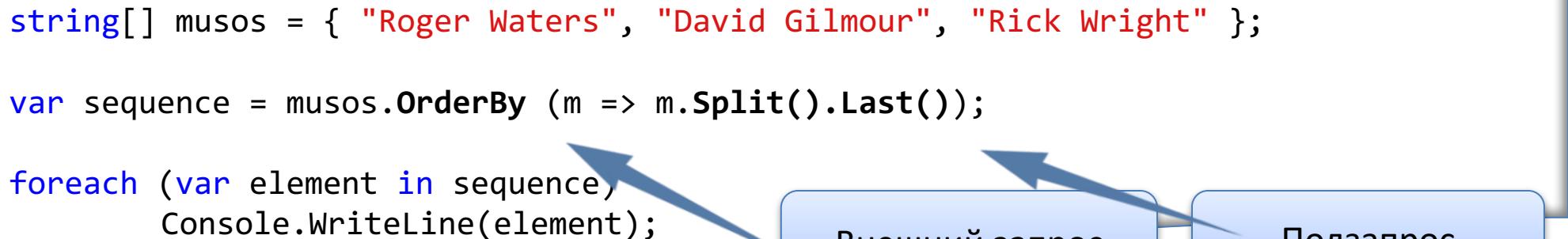
## Подзапросы

Подзапрос – это запрос, содержащий внутри лямбда-выражение другого запроса

```
string[] musos = { "Roger Waters", "David Gilmour", "Rick Wright" };

var sequence = musos.OrderBy (m => m.Split().Last());

foreach (var element in sequence)
    Console.WriteLine(element);
```



Внешний запрос

Подзапрос

В выражении запроса подзапрос означает запрос, расположенный в любой конструкции кроме from

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

var sequence = names
    .Where(n => n.Length == names
        .OrderBy (n2 => n2.Length)
        .Select (n2 => n2.Length)
        .First());

foreach (var element in sequence)
    Console.WriteLine(element);
```

Tom  
Jay

## Подзапросы

```
var query = from n in names  
            where n.Length ==  
                  (from n2 in names orderby n2.Length select n2.Length)  
                  .First()  
            select n;
```

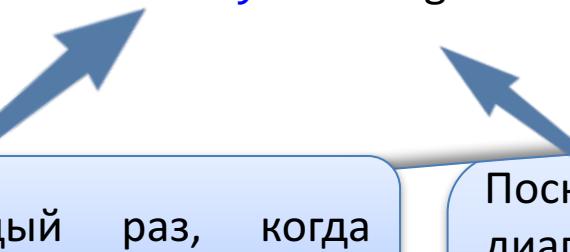
Подзапрос выполняется каждый раз, когда вычисляется включающее его лямбда-выражение

Выполнение направляется снаружи внутрь

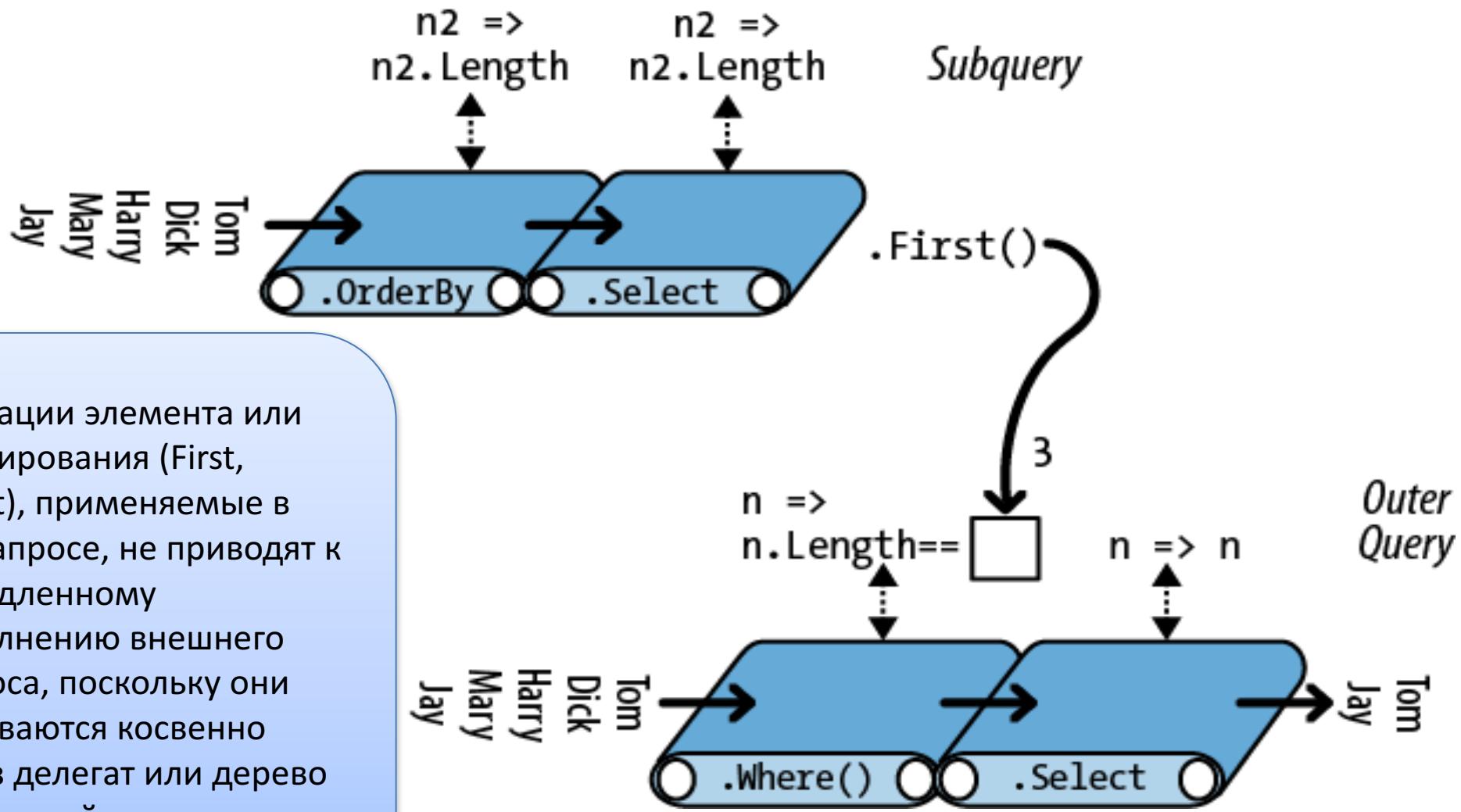
Локальные запросы следуют этой модели буквально, а интерпретируемые - концептуально

Запрос не эффективен для локальной последовательности, поскольку подзапрос вычисляется повторно на каждой итерации

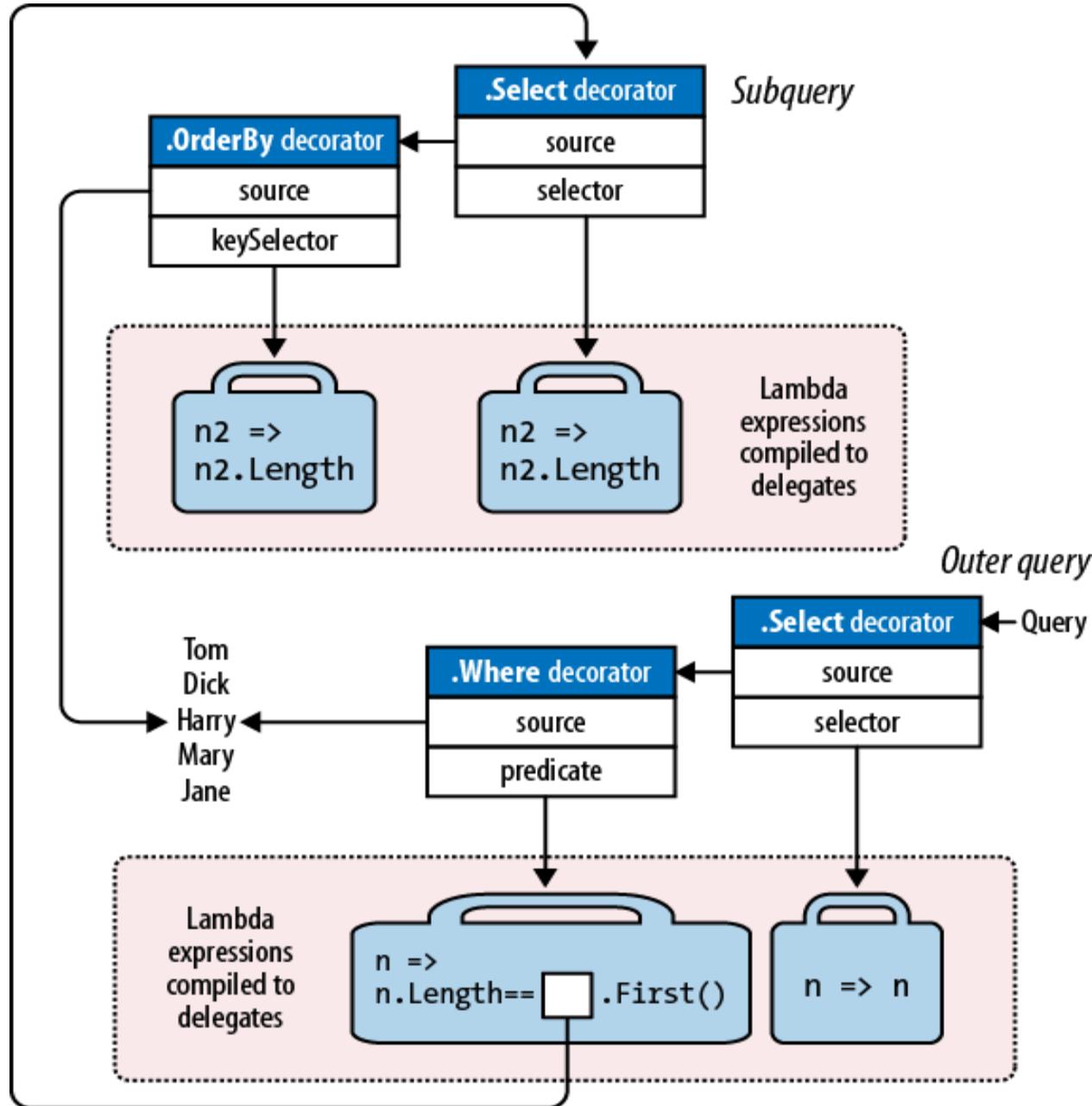
Поскольку внешняя переменная диапазона n находится в области видимости, использовать n в качестве переменной диапазона подзапроса нельзя



## Подзапросы



## Подзапросы



## Стратегии композиции

Стратегии построения более сложных запросов

- последовательное построение запросов
- использование ключевого слова `into`
- упаковка запросов

Преимущества последовательного построения запросов

- упрощение написания запросов
- операции запросов можно добавлять условно

`if (includeFilter) query = query.Where (...)`

более эффективно, чем

`query = query.Where (n => !includeFilter || <expression>)`

## Стратегии композиции. Последовательное построение запросов

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IQueryable<string> query = names
    .Select (n => n.Replace ("a", "").Replace ("e", "").Replace ("i", ""))
    .Replace ("o", "").Replace ("u", ""))
    .Where (n => n.Length > 2)
    .OrderBy (n => n);
```

RESULT: { "Dck", "Hrry", "Mry" }

```
IEnumerable<string> query = from n in names
    where n.Length > 2
    orderby n
    select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
        .Replace ("o", "").Replace ("u", "");
```

RESULT: { "Dck", "Hrry", "Jy", "Mry", "Tm" }

```
IEnumerable<string> query =
    from n in names
    select n.Replace ("a", "").Replace ("e", "").Replace ("i", "").Replace
("o", "").Replace ("u", "");
```

```
query = from n in query where n.Length > 2 orderby n select n;
```

## Стратегии композиции. Ключевое слово `into`

В зависимости от контекста **ключевое слово `into`** интерпретируется выражениями запросов

- сигнализация о продолжении запроса
- сигнализация GroupJoin

```
IEnumerable<string> query =  
    from n in names  
    select n.Replace ("a", "").Replace ("e", "").Replace ("i", "")  
        .Replace ("o", "").Replace ("u", "")  
    into noVowel  
    where noVowel.Length > 2 orderby noVowel select noVowel;
```



Позволяет продолжить запрос после выполнения проекции и является сокращением для последовательного построения запросов

Ключевое слово `into` можно использовать только после конструкции `select` или `group`

Ключевое слово `into` «перезапускает» запрос, позволяя вводить новые конструкции `where`, `orderby`, `select`

## Стратегии композиции. Упаковка запросов

Запрос, построенный последовательно, может быть сформулирован как единственный оператор **за счет упаковки одного запроса** в другой

Запрос

```
var tempQuery = tempQueryExpr
var finalQuery = from ... in tempQuery ...
```

Может быть переформулирован

```
var finalQuery = from ... in (tempQueryExpr)
```

Упаковка семантически идентична последовательному построению запросов или применению ключевого слова `into`

## Стратегии композиции. Упаковка запросов

```
var names = new[] { "Tom", "Dick", "Harry", "Mary", "Jay" };

IQueryable<string> query =
    from n in names
    select n.Replace ("a", "").Replace ("e", "").Replace ("i",
    "").Replace ("o", "").Replace ("u", "");

query = from n in query where n.Length > 2 orderby n select n;
```

```
IEnumerable<string> query =
    from n1 in
    (
        from n2 in names
        select n2.Replace ("a", "").Replace ("e", "").Replace ("i",
        "").Replace ("o", "").Replace ("u", ")
    )
    where n1.Length > 2 orderby n1 select n1;
```

## Стратегии проекции

С помощью инициализаторов объектов можно выполнять проецирование не только скалярных типов

```
class TempProjectionItem
{
    public string Original;
    public string Vowelless;
}
var names = new[] { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<TempProjectionItem> temp =
    from n in names
    select new TempProjectionItem
    {
        Original = n,
        Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i",
            "").Replace ("o", "").Replace ("u", "")
    };

```

```
IEnumerable<string> query = from item in temp
                            where item.Vowelless.Length > 2
                            select item.Original;
```

Dick  
Harry  
Mary

## Анонимные типы (Anonymous Type)

Когда неудобно декларировать тип переменной при ее объявлении?

При полноценной работе с коллекциями тип результата операции или набора операций над коллекциями может сильно отличаться от типов обрабатываемых коллекций

- Создание нового типа каждый раз, когда необходимо выполнить преобразование
- Создание одного большого типа со всеми полями, которые только могут получиться в результате операций

**Анонимные типы** – это возможность создать новый тип, декларируя его не заранее, а непосредственно при создании переменной, причем типы и имена полей выводятся компилятором автоматически из инициализации

## Стратегии проекции

Анонимные типы позволяют структурировать промежуточные результаты без написания специальных классов

```
var names = new[] { "Tom", "Dick", "Harry", "Mary", "Jay" };

var intermediate = from n in names
    select new
    {
        Original = n,
        Vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i",
            "").Replace ("o", "").Replace ("u", "")
    };
    
IQueryable<string> query =
    from item in intermediate
    where item.Vowelless.Length > 2
    select item.Original;
```

IEnumerable <random-compiler-produced-name>

## Стратегии проекции

Ключевое слово `let` вводит новую переменную вместе с переменной запроса

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

IQueryable<string> query =
    from n in names
    let vowelless = n.Replace ("a", "").Replace ("e", "").Replace ("i", "")
                  .Replace ("o", "").Replace ("u", "")
    where vowelless.Length > 2
    orderby vowelless
    select n; // Thanks to let, n is still in scope.
```

Компилятор разрешает конструкцию `let`, выполняя проецирование во временный анонимный тип, который содержит переменную диапазона и новую переменную выражения

# Обзор стандартных операций запросов

## Категории стандартных операций запросов

Последовательность на входе, последовательность на выходе  
(последовательность ->последовательность)

Последовательность на входе, одиничный элемент на или скалярное произведение на выходе

Ничего на входе, последовательность на выходе

# Обзор стандартных операций запросов

**Последовательность → последовательность**

## Фильтрация

`IEnumerable<TSource> → IEnumerable<TSource>`

Возвращают подмножество исходных элементов

`Where, Take, TakeWhile, Skip, SkipWhile, Distinct`

## Проектирование

`IEnumerable<TSource> → IEnumerable<TResult>`

Трансформируют каждый элемент с помощью лямбда-функции

`Select, SelectMany`

# Обзор стандартных операций запросов

**Последовательность → последовательность**

**Соединение**

`IEnumerable<TOuter>, IEnumerable<TInner> → IEnumerable<TResult>`

Объединяют элементы одной последовательности с другой

`Join, GroupJoin, Zip`

**Упорядочивание**

`IEnumerable<TSource> → IOrderedEnumerable<TSource>`

Возвращает упорядоченную последовательность

`OrderBy, ThenBy, Reverse`

# Обзор стандартных операций запросов

**Последовательность → последовательность**

**Группирование**

`IEnumerable<TSource> → IEnumerable<IGrouping<TSource, TElement>>`

Группирует последовательность в подпоследовательности

**GroupBy**

**Операции над множествами**

`IEnumerable<TSource>, IEnumerable<TSource> → IEnumerable<TSource>`

Принимает две последовательности одного и того же типа и возвращает их объединение, сумму, пересечение

**Concat, Union, Intersect, Except**

# Обзор стандартных операций запросов

**Последовательность → последовательность**

**Методы преобразования: импортирование**

`IEnumerable<T> → IEnumerable<TResult>`

`OfType, Cast`

**Методы преобразования: экспортование**

`IEnumerable<TSource>` → массив, список, словарь, объект `Lookup` или последовательность

`ToArray, ToList, ToDictionary, ToLookup, AsEnumerable, AsQueryable`

# Обзор стандартных операций запросов

**Последовательность → элемент или значение**

## Операции элементов

`IEnumerable<TSource> → Tsource`

Выбирает одиночный элемент из последовательности

`First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault,  
ElementAt, ElementAtOrDefault, DefaultIfEmpty`

## Методы агрегирования

`IEnumerable<TSource> → скалярное значение`

Выполняет вычисление над последовательностью, возвращая скалярное значение

`Aggregate, Average, Count, LongCount, Sum, Max, Min`

# Обзор стандартных операций запросов

**Последовательность → элемент или значение**

**Квантификаторы**

**IEnumerable<TSource> → значение bool**

Агрегация, возвращающая значение true или false

**All, Any, Contains, SequenceEqual**

# Обзор стандартных операций запросов

**Ничего → последовательность**

**Методы генерации**

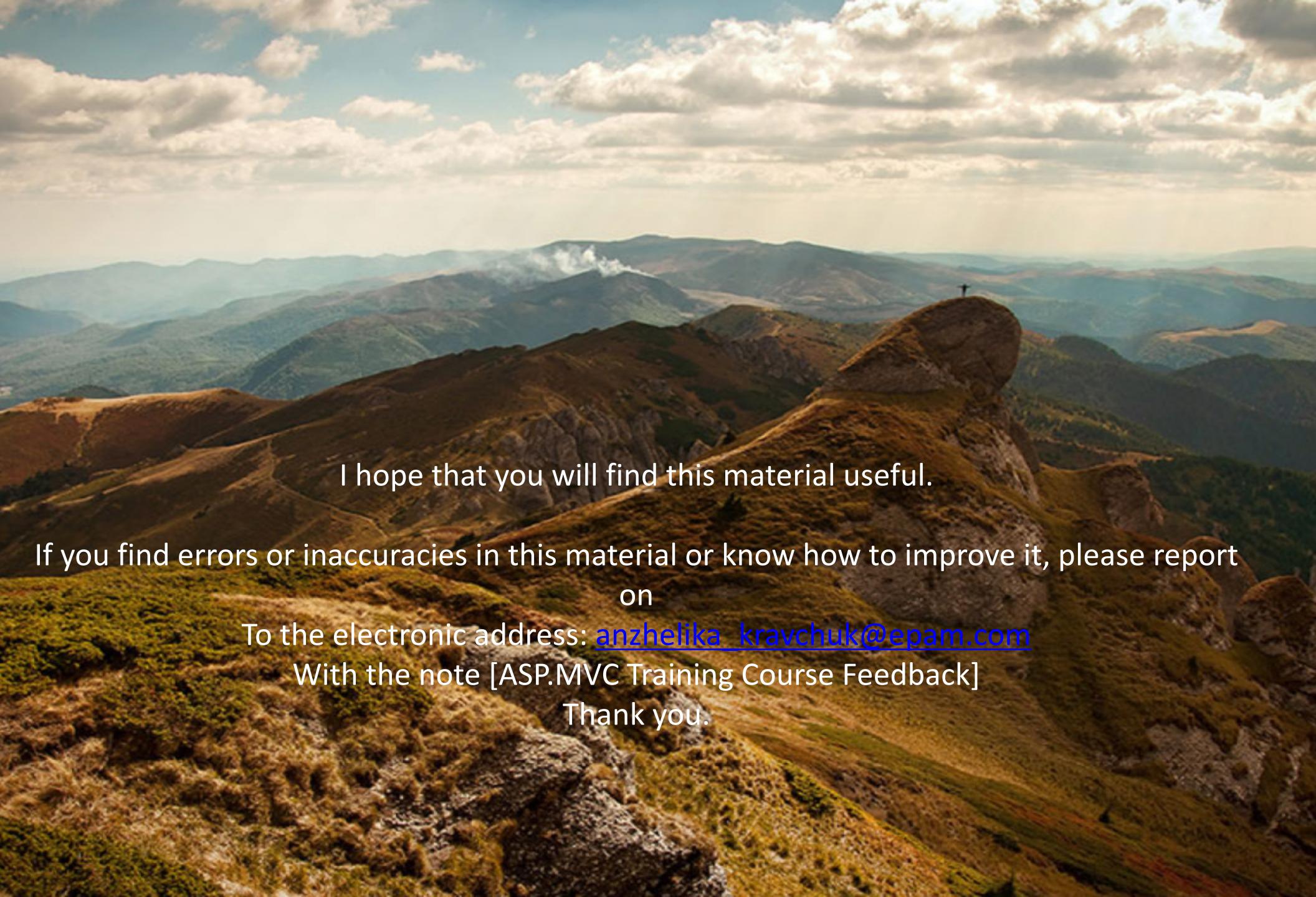
**Ничего → `IEnumerable<TResult>`**

Производит простую последовательность

`Empty`, `Range`, `Repeat`

A wide-angle photograph of a mountainous landscape under a bright sky with scattered clouds. In the foreground, rocky terrain and green slopes are visible. A prominent, rounded mountain peak rises in the center-right. On top of this peak stands a small figure of a person, appearing as a tiny dark dot against the light-colored rock. The background features a range of mountains fading into a hazy horizon.

Thank you for attention!

A wide-angle landscape photograph of a mountainous region. In the foreground, there are rocky, grassy slopes. A prominent peak on the right side has a small figure standing on its rocky ridge. The background features a range of mountains under a sky filled with scattered, bright clouds.

I hope that you will find this material useful.

If you find errors or inaccuracies in this material or know how to improve it, please report  
on

To the electronic address: [anzhelika\\_kravchuk@epam.com](mailto:anzhelika_kravchuk@epam.com)

With the note [ASP.MVC Training Course Feedback]

Thank you.