

# Визуальное программи рование

ЛЕКЦИЯ 8

# Содержание лекции

01 LINQ

02 Синтаксис LINQ

03 Подзапросы

04 Отложенное  
выполнение

# LINQ

# Понятие LINQ

**Language-Integrated Query (LINQ)** — язык интегрированных запросов, который представляет набор языковых и платформенных средств для описания структурированных и типобезопасных запросов к локальным коллекциям и удаленным источникам данных.

VB

C#

Others

.NET Language-Integrated Query(LINQ)

**LINQ Enabled ADO.NET**

LINQ  
To Objects

LINQ  
To Dataset

LINQ  
To SQL

LINQ  
To Entities

LINQ  
To XML



Objects



Relational  
Database



XML

# Понятие LINQ

Базовые единицы данных LINQ:

- **последовательность** – любой объект, реализующий интерфейс `IEnumerable<T>`
- **элемент** – элемент внутри последовательности

**Локальная последовательность** представляет локальную коллекцию объектов в памяти.

# Понятие LINQ

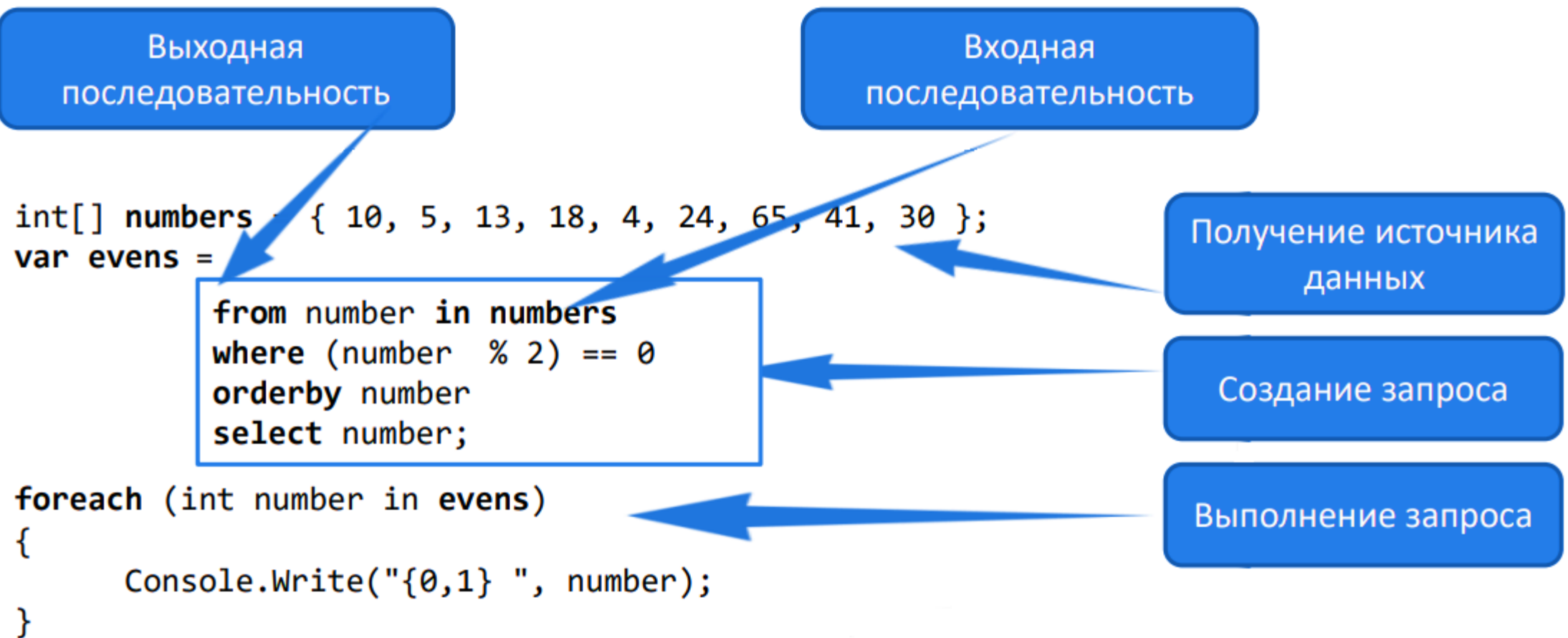
**Операция запроса** – метод, трансформирующий входную последовательность.

**Запрос** представляет собой выражение, которое при перечислении трансформирует последовательности с помощью операции запросов.

Запросы, оперирующие на локальных последовательностях, называются **локальными запросами** или запросами **LINQ to Object**.

В классе Enumerable (System.Linq) имеется около 40 операций запросов, которые реализованы в виде методов расширения.

# Понятие LINQ





# Понятие 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**.

LINQ также поддерживает последовательности, которые могут наполняться из удаленных источников, таких как SQL Server. Они поддерживаются посредством соответствующего набора стандартных операций запросов в классе `Queryable`.

# Понятие LINQ

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

Получение источника  
данных

```
var companyNameQuery =
```

```
    from cust in nw.Customers  
    where cust.City == "London"  
    select cust.CompanyName;
```

Создание запроса

```
foreach (var customer in companyNameQuery)  
{  
    Console.WriteLine(customer);  
}
```

Выполнение запроса

Переменная  
запроса

Выражение  
запроса

# Синтаксис LINQ

# Синтаксис LINQ

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

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

Текущий синтаксис позволяет формировать **цепочки операций запросов**.

# Синтаксис LINQ

```
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|

Синтаксис операций  
запросов (fluent)

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

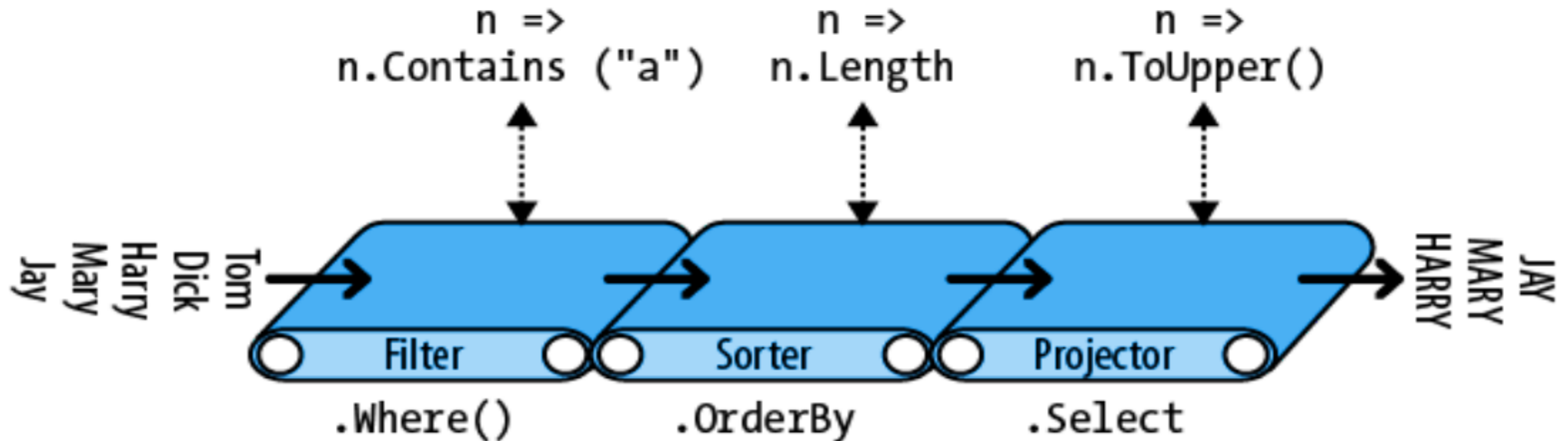
```
IEnumerable<string> filteredNames = from n in names  
                                   where n.Contains ("a")  
                                   select n;
```

Синтаксис выражений  
запросов

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

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

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



# 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)
```

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

```
IEnumerable<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"));  
IEnumerable<string> sorted      = filtered.OrderBy (n => n.Length);  
IEnumerable<string> finalQuery = sorted.Select      (n => n.ToUpper());
```

Harry      **Filtered**

Mary

Jay

Jay      **Sorted**

Mary

Harry

JAY      **FinalQuery**

MARY

HARRY

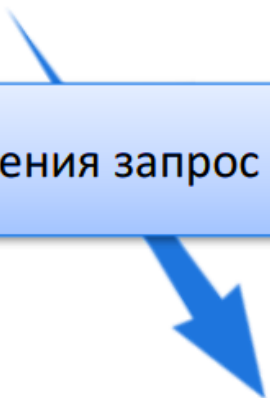


# Fluent syntax

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

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

Без методов расширения запрос теряет свою текучесть



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

# Лямбда-выражения

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

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

Лямбда-выражение позволяет помещать собственную логику внутрь операций запроса.

Лямбда-выражение, которое принимает значение и возвращает bool, называется **предикатом**.

# Лямбда-выражения

```
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)
```

# 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

```
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

Компилятор обрабатывает выражение запроса путем его трансляции в текущий синтаксис.

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

```
IEnumerable<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

# Выражения запросов

```
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
```



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

Переменная  
диапазона

# Выражения запросов

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

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



# Анонимные типы

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.

Анонимные типы позволяют структурировать промежуточные результаты без написания специальных классов.

# АНОНИМНЫЕ ТИПЫ

```
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", "")
                    };
```

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

# ПОДЗАПРОСЫ

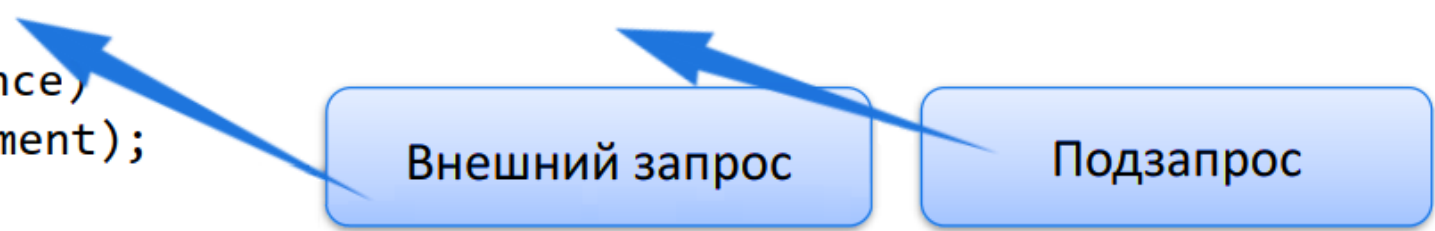
# Подзапросы

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

# Подзапросы

```
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

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

Выполнение направляется снаружи внутрь. **Локальные запросы** следуют этой модели **буквально**, а **интерпретируемые** – **концептуально**.

Операции элемента или агрегирования (First, Count), применяемые в подзапросе, не приводят к немедленному выполнению внешнего запроса, поскольку они вызываются косвенно через делегат или дерево.

# Подзапросы

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

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

# ОТЛОЖЕННОЕ ВЫПОЛНЕНИЕ



# Отложенное выполнение

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

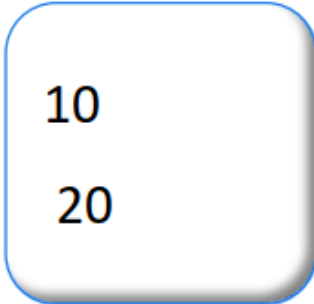
Важная особенность большинства операций запросов – выполнение не при конструировании, а **во время перечисления**.

Отложенное выполнение поддерживают все стандартные операции запросов кроме:

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

# Отложенное выполнение

```
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

```
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);
```

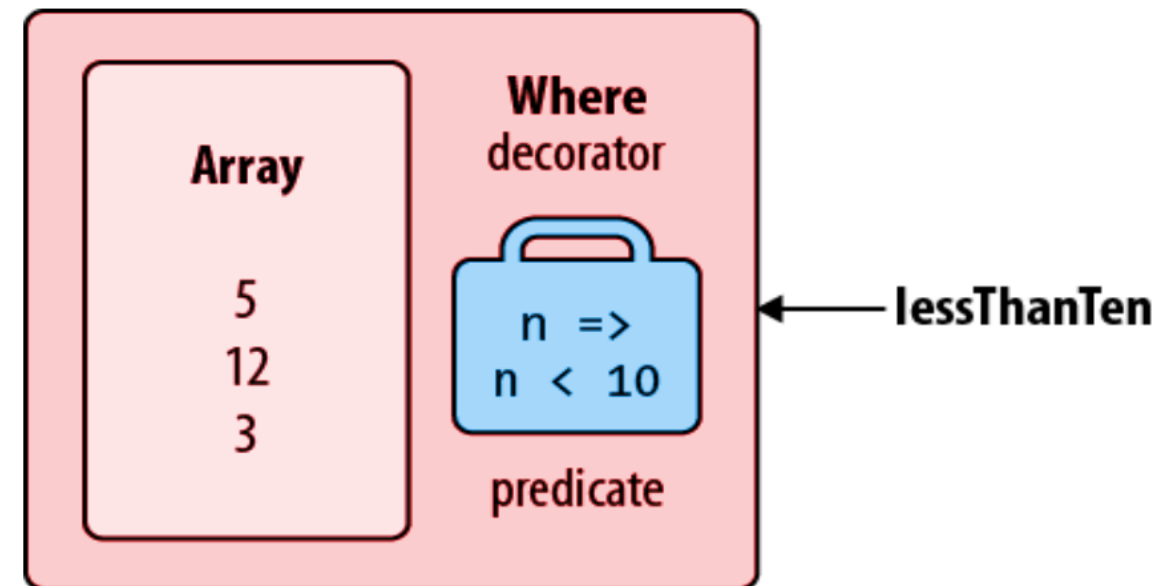
```
numbers.Clear();  
foreach(var temp in timesTen)  
    Console.WriteLine(temp);
```

# Отложенное выполнение

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

Реализация декоратора осуществляется с помощью **итератора**.

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




# Отложенное выполнение

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

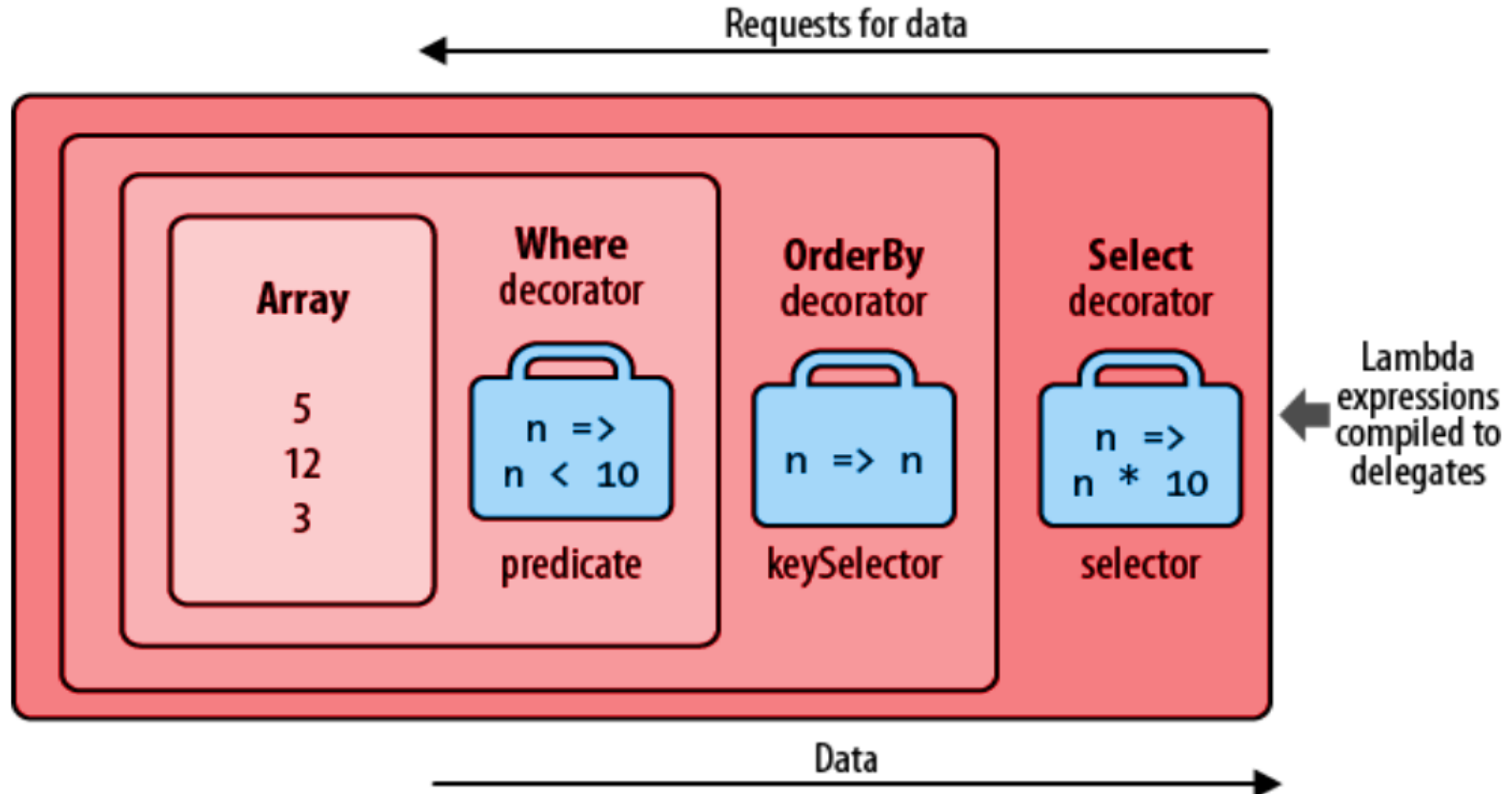
```
public static IEnumerable<TResult> Select<TSource,TResult>
    (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
    foreach (TSource element in source)
        yield return selector (element);
}

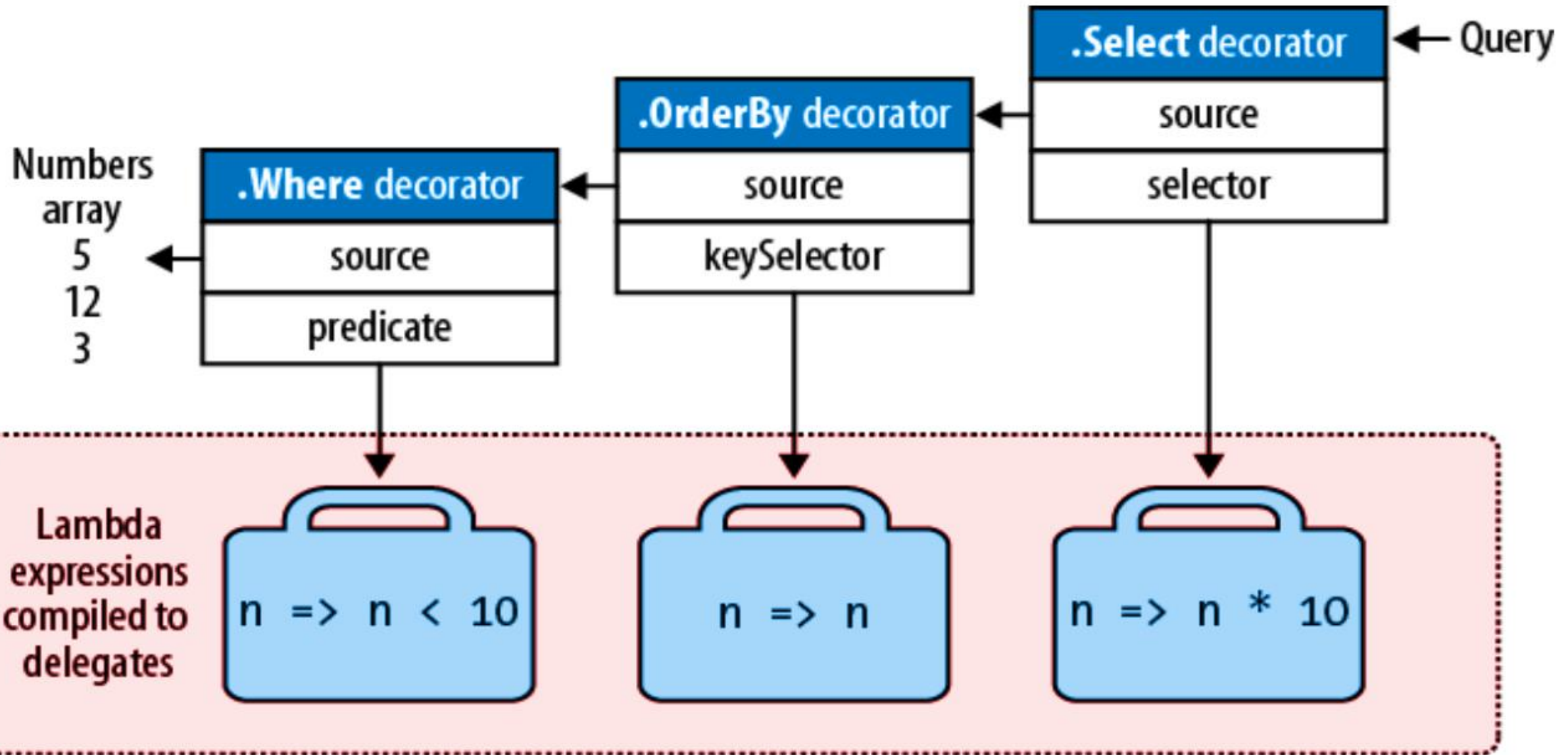
public static IEnumerable<TResult> Select<TSource,TResult>
    (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
{
    return new SelectSequence (source, selector);
}
```



Сгенерированный компилятором класс, перечислитель которого инкапсулирует логику из метода итератора

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





# ВОПРОСЫ ПО ЛЕКЦИИ