



Тема 12.

# LINQ и всё-всё-всё...

© 2014, 2016, Serge Kashkevich

# Что это такое?

Лямбда-выражения и язык LINQ (Language-INtegrated Query) представляют собой конструкции, синтаксис которых отличается от базового синтаксиса C#. Использование этих конструкций позволяет просто описать нужные нам действия, не затрагивая базовый синтаксис C#.

# Лямбда-выражения

Лямбда-выражения представляют собой альтернативу анонимным методам C#, но обладают интуитивно более понятным синтаксисом.

Основное применение лямбда-выражений – язык LINQ. Однако они могут использоваться в качестве анонимных делегатов. А поскольку лямбда-выражение считается более эффективным, чем эквивалентный ему анонимный метод, то в большинстве случаев рекомендуется отдавать предпочтение именно ему.

# Синтаксис лямбда-выражений

- *(параметры) => выражение*
- *параметр => выражение*
- *(параметры) => { тело функции }*

Под «параметром» понимается конструкция

*[тип] имя*

Таким образом, когда требуется указать два параметра или более (или для единственного, когда требуется его тип), их следует заключить в скобки. Если же выражение не требует параметров, то следует использовать пустые скобки.

Если тип параметра не указан, может быть применён любой тип, который является допустимым для выражения.

# Примеры лямбда-выражений

- `count => count + 2`

Параметром может быть любой тип, для которого определена операция сложения с целым числом. Р

- `(x, y) => x > y`

Параметрами могут быть любые типы, для которых определена операция «больше». Результат выполнения – `true` или `false`.

- `(int count) => count / 3`

`count` теперь явно объявлен как параметр типа `int`. Обратите также внимание на использование скобок. Теперь они необходимы. (Скобки могут быть опущены только в том случае, если задается лишь один параметр, а его тип явно не указывается). Результат – целочисленное деление параметра на 3.

# Использование лямбда-выражений при работе с делегатами

- Описываем делегат, подходящий по типу для лямбда-выражения
- Создаём экземпляр делегата, инициализируя его лямбда-выражением
- Используем созданный делегат при вычислениях

```
class Program
{
    delegate bool D(int a, int b);
    static D d;

    static void Main(string[] args)
    {
        d = (x, y) => x > y;
        Console.WriteLine(d(5, 6));
        Console.WriteLine(d(15, 6));
    }
}
```

## Пояснение примера из темы 2

*Задача:* С консоли вводится строка, содержащая несколько целых чисел, разделённых одним или несколькими пробелами. Сформировать на основании этой строки массив целых чисел

```
string [] ArS = (Console.ReadLine()).Split  
                (new char[]{' '});  
int[] ArI = new int[ArS.Count(s => s.Length > 0)];
```

Один из перегруженных методов **Count** класса **Array** имеет формальный параметр – делегат унарного предиката. В качестве фактического параметра может быть задан экземпляр делегата, в том числе и лямбда-выражение.

# Захват переменных

В лямбда-выражениях могут использоваться не только параметры, но и ранее описанные переменные (*захваченные* переменные). Более того, при вычислении лямбда-выражений значения захваченных переменных могут изменяться:

```
class Program {  
    delegate int D();  
    static D d;  
  
    static void Main(string[] args) {  
        int p = 0;  
        d = () => ++p;  
        for (int i = 0; i<10; i++)  
            Console.WriteLine("{0} {1} {2}", i, p, d());  
    }  
}
```



# Захват переменных (продолжение)

Срок жизни захваченных переменных продлевается до окончания срока жизни захватившего её делегата:

```
class Program {  
    delegate int D();  
    static D d;  
  
    static void Init() {  
        int p = 0;  
        d = () => ++p;  
    }  
  
    static void Main(string[] args) {  
        Init();  
        for (int i = 0; i < 10; i++)  
            Console.WriteLine("{0} {1}", i, d());  
    }  
}
```

# Блочные лямбда-выражения

Вместо отдельного выражения в правой части может быть записан блок операторов C#.

В приведенном ниже примере в теле блочного лямбда-выражения объявляется переменная `rezt`, организуется цикл `for` и используется оператор `return`. Когда в блочном лямбда-выражении встречается оператор `return`, он просто обуславливает возврат из лямбда-выражения, но не возврат из охватывающего метода

```
class Program {
    delegate int D1(int x);
    static D1 fact;

    static void Main(string[] args) {
        fact = n => {
            int rezt = 1;
            for (int i = 1; i <= n; i++)
                rezt *= i;
            return rezt;
        };

        for (int i = 0; i <= 10; i++)
            Console.WriteLine("{0} {1}", i, fact(i));
    }
}
```

# Неявно типизированные переменные

Начиная с версии C# 3.0, компилятору предоставляется возможность самому определить тип локальной переменной, исходя из значения, которым она инициализируется. Такая переменная называется неявно типизированной.

Неявно типизированная переменная объявляется с помощью ключевого слова `var` и должна быть непременно инициализирована. Для определения типа этой переменной компилятору служит тип ее инициализатора, т.е. значения, которым она инициализируется.

```
var e = 2.7183;
```

Переменная `e` получает тип `double`

```
var e = 2.7183F;
```

Переменная `e` получает тип `float`

# LINQ

Аббревиатура LINQ означает Language-Integrated Query, т.е. язык интегрированных запросов. Это понятие охватывает ряд средств, позволяющих извлекать информацию из источника данных. Извлечение данных составляет важную часть многих программ.

LINQ – это набор средств, позволяющих формировать запросы для любого LINQ-совместимого источника. При этом синтаксис, используемый для формирования запросов, остается неизменным, независимо от типа источника данных. Это, в частности, означает, что синтаксис, требующийся для формирования запроса к реляционной базе данных, практически ничем не отличается от синтаксиса запроса данных, хранящихся в массиве.

# Запросы

В основу LINQ положено понятие запроса, в котором определяется информация, получаемая из источника данных. Различают две фазы работы с запросом:

- формирование;
- выполнение.

При формировании запроса определяется, что именно следует извлечь из источника данных. А при выполнении запроса выводятся конкретные результаты.

# Ограничения на источник данных

Для обращения к источнику данных по запросу, сформированному средствами LINQ, в этом источнике должен быть реализован интерфейс `IEnumerable` (в обобщённой или необобщённой форме).

В дальнейшем будет рассматриваться только обобщённая форма реализации этого интерфейса:  
`IEnumerable <T>`

# Пример простейшего запроса

```
class Program {  
    static void Main(string[] args) {  
        int[] nums = { 1, -2, 3, 0, -4, 5 };  
        // Сформировать запрос на получение только  
        // положительных значений  
        var posNums = from n in nums  
                       where n > 0  
                       select n;  
        Console.WriteLine("Положительные значения массива:  
");  
        // Выполнить запрос и отобразить его результаты.  
        foreach (int i in posNums)  
            Console.Write(i + " ");  
        Console.WriteLine();  
    }  
}
```

Формирование  
запроса

Выполнение  
запроса

**Результат: 1 3 5**

# Пример простейшего запроса (продолжение)

Однажды сформулированный запрос может быть выполнен несколько раз, и изменения в источнике данных учитываются:

```
class Program {  
    static void Main(string[] args) {  
        int[] nums = { 1, -2, 3, 0, -4, 5 };  
        // Сформировать запрос на получение только  
        // положительных значений  
        var posNums = from n in nums  
                       where n > 0  
                       select n;  
        Console.Write("Положительные значения массива: ");  
        foreach (int i in posNums)  
            Console.Write(i + " ");  
        Console.WriteLine();  
        nums[0] = -10;  
        nums[4] = 111;  
        foreach (int i in posNums)  
            Console.Write(i + " ");  
        Console.WriteLine();  
    }  
}
```

Первое  
выполнение  
запроса

Второе  
выполнение  
запроса

**Результат:**

1 3 5  
3 111 5



# Структура простейшего запроса LINQ

```
from n in nums // указываем источник данных
where n > 0     // задаём условие отбора
select n       // указываем результат запроса
```

Аналог SQL-запроса (если nums – таблица реляционной базы данных, а n – её столбец):

```
select nums.n
from nums
where nums.n > 0;
```

# Структура запроса LINQ – описание

Запрос LINQ состоит из нескольких фраз:

Синтаксис фразы	Пояснения
from <b>переменная</b> in <b>источник</b>	указывает источник данных и объявляет <i>переменную запроса</i>
group <b>переменная</b> by <b>выражение</b>	задаёт возможность группировки переменной запроса в соответствии со значениями заданного выражения
into <b>имя</b> <b>продолжение_запроса</b>	формирует временный результат, который будет использоваться в продолжении запроса
join <b>переменная</b>   in <b>источник</b>   on <b>выражение</b>	соединяет несколько источников данных
let <b>имя</b> = <b>выражение</b>	создаёт временную переменную запроса
orderby <b>поле</b> <b>порядок</b>	упорядочивает результаты выполнения запроса
select <b>выражение</b>	формирует выражение, которое будет результатом запроса
where <b>условие</b>	формирует фильтр для запроса

# Структура запроса LINQ – окончание

Запрос LINQ должен начинаться фразой `from` и заканчиваться фразами `select` или `group`.

*Дальнейшее изложение будет базироваться на примерах!*

# Формирование выражения для результата запроса

**Задача:** Для положительных элементов массива вывести их квадратные корни

```
int[] nums = { 1, -2, 3, 0, -4, 5 };  
var posNums = from n in nums  
               where n > 0  
               select Math.Sqrt(n);  
Console.WriteLine(@"Квадратные корни для  
положительных значений из массива nums: ");  
foreach (double i in posNums)  
    Console.Write("{0:0.##} ", i);  
Console.WriteLine();
```

**Результат: | 1,73 2,24**

## Вложенные фразы from

Запрос может состоять из нескольких операторов from, которые оказываются в этом случае вложенными. Такие операторы from находят применение в тех случаях, когда по запросу требуется получить данные из двух разных источников. Эта операция в теории баз данных называется декартовым произведением.

- **Задача:** Использовать два вложенных оператора from для составления списка всех возможных сочетаний букв А, В и С с буквами X, Y и Z.

```
class ChrPair {  
    public char First;  
    public char Second;  
    public ChrPair(char c1, char c2) {  
        First = c1; Second = c2;  
    }  
}
```

# Вложенные фразы from (продолжение)

```
class Program {
```

```
// В первом операторе from организуется циклическое обращение  
// к массиву символов chrs1, а во втором операторе from –  
// циклическое обращение к массиву символов chrs2.
```

```
static void Main(string[] args) {  
    char[] chrs1 = { 'A', 'B', 'C' };  
    char[] chrs2 = { 'X', 'Y', 'Z' };  
    var pairs = from ch1 in chrs1  
                from ch2 in chrs2  
                select new ChrPair(ch1, ch2);  
    foreach (var p in pairs)  
        Console.WriteLine("{0} {1}",  
                            p.First, p.Second);  
}
```

**Результат запишите самостоятельно!**

# Группировка

Одним из самых эффективных средств формирования запроса является фраза `group`, поскольку она позволяет группировать полученные результаты по ключам. Используя последовательность сгруппированных результатов, можно без особого труда получить доступ ко всем данным, связанным с ключом. Благодаря этому свойству фразы `group` доступ к данным, организованным в последовательности связанных элементов, осуществляется просто и эффективно.

Фраза `group` является одной из двух фраз, которыми может оканчиваться запрос.

Общая форма фразы `group`:

`group` переменная\_диапазона `by` ключ

# Группировка (продолжение)

**Задача:** Для футбольных команд заданы их название и город. Требуется сгруппировать команды по городам.

```
class Team {
    public string Name;
    public string City;
    public Team(string s1, string s2) {
        Name = s1; City = s2;
    }
}

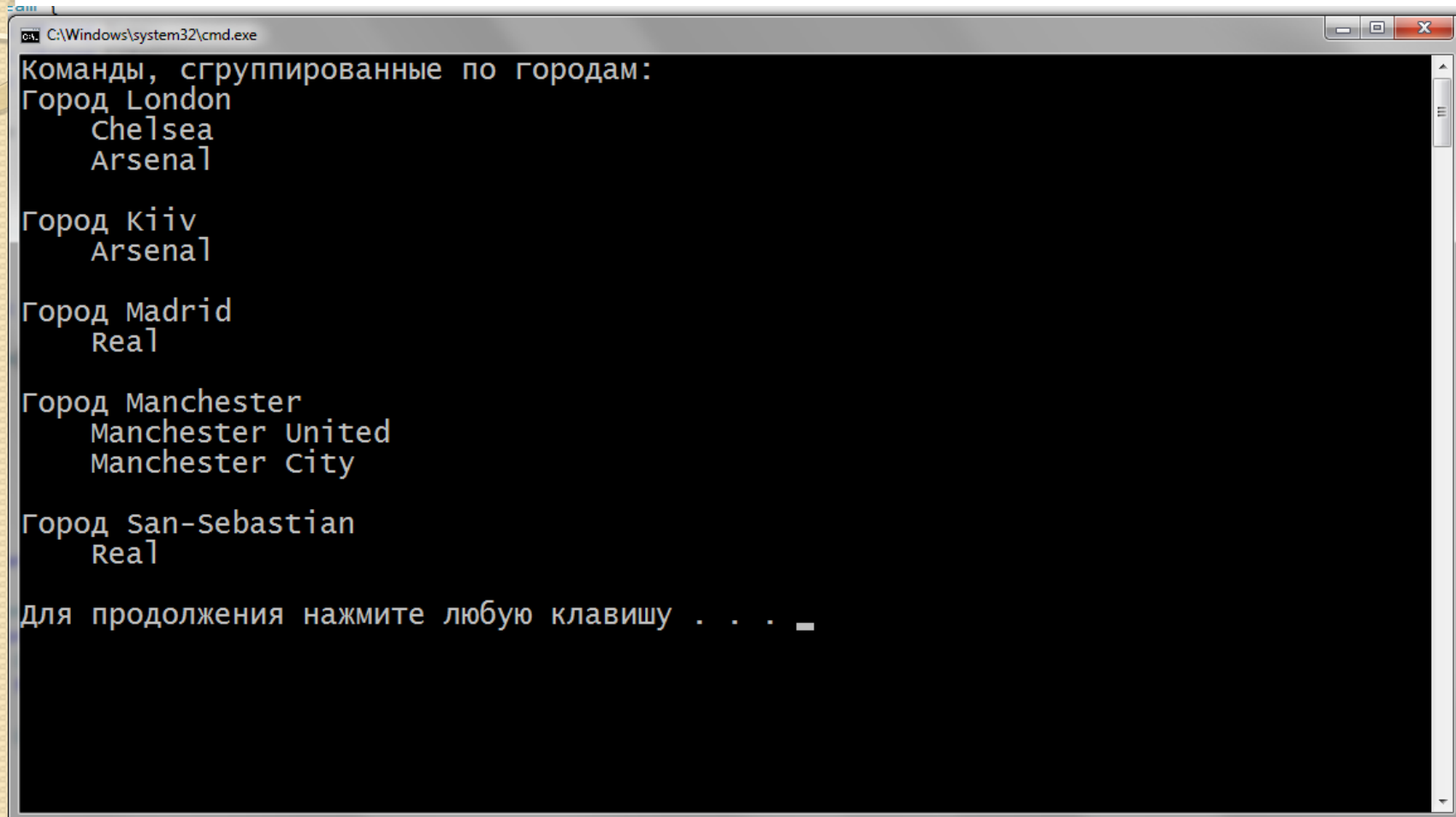
class Program
{
    static void Main(string[] args)
    {
        Team[] teams = { new Team("Chelsea", "London"),
                          new Team("Arsenal", "London"),
                          new Team("Arsenal", "Kiev"),
                          new Team("Real", "Madrid"),
                          new Team("Manchester United", "Manchester"),
                          new Team("Manchester City", "Manchester"),
                          new Team("Real", "San-Sebastian")
        };
    }
}
```



## Группировка (продолжение)

```
var rez = from t in teams
           group t by t.City;
Console.WriteLine(
    "Команды, сгруппированные по городам: ");
foreach (var city in rez) {
    Console.WriteLine("Город {0}", city.Key);
    foreach (var team in city)
        Console.WriteLine("    " + team.Name);
    Console.WriteLine();
}
}
```

# Группировка (результаты)



A screenshot of a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window has a black background with white text. The text is organized into groups by city, with each city name on a new line followed by its associated team names on subsequent lines. The cities listed are London, Kiiv, Madrid, Manchester, and San-Sebastian. At the bottom, there is a prompt for the user to press any key to continue.

```
Команды, сгруппированные по городам:  
Город London  
    Chelsea  
    Arsenal  
  
Город Kiiv  
    Arsenal  
  
Город Madrid  
    Real  
  
Город Manchester  
    Manchester United  
    Manchester City  
  
Город San-Sebastian  
    Real  
  
для продолжения нажмите любую клавишу . . .
```

# Фраза into

При использовании в запросе фразы select или group иногда требуется сформировать временный результат, который будет служить продолжением запроса для получения окончательного результата. Такое продолжение осуществляется с помощью фразы into в комбинации с фразами select или group.

Общая форма запроса с фразой into:

```
запрос_1  
into имя  
запрос_2
```

Когда оператор into используется вместе с оператором select или group, то его называют продолжением запроса, поскольку он продолжает запрос. По существу, продолжение запроса воплощает в себе принцип построения нового запроса по результатам предыдущего.

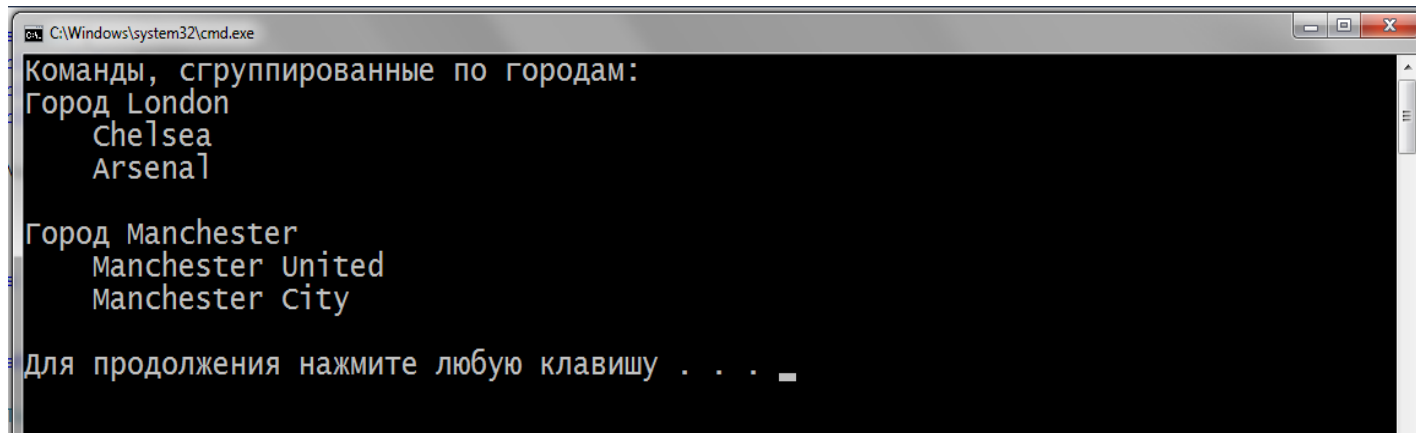
# Использование фразы into для наложения фильтров на группу

*Задача:* Изменить условие предыдущей задачи: выдавать информацию только по тем городам, в которых размещается не менее двух команд.

*Изменения:*

```
var rez = from t in teams
           group t by t.City
           into w
           where w.Count() > 1
           select w;
```

*Результат:*



```
C:\Windows\system32\cmd.exe
Команды, сгруппированные по городам:
Город London
    Chelsea
    Arsenal

Город Manchester
    Manchester United
    Manchester City

Для продолжения нажмите любую клавишу . . .
```

# Фраза join. Соединение ИСТОЧНИКОВ

Современные реляционные базы данных состоят из нескольких взаимосвязанных таблиц. Связь осуществляется за счёт того, что ключевые поля из одной таблицы мигрируют в другие таблицы в качестве внешних ключей. В процессе обработки данных нам требуется соединять данные из разных таблиц воедино. В LINQ для соединения данных используется фраза join.

Формат этой фразы:

```
from переменная_A in источник_данных_A  
join переменная_B in источник_данных_B  
on условие_соединения
```

Условие соединения записывается как

```
переменная_A.свойство equals переменная_B.свойство
```

# Соединение I:I

Рассмотрим вначале ситуацию, когда значения ключей не повторяются в обоих источниках данных (отношение «один к одному»). В результате соединения мы получаем данные по тем строкам различных источников, в которых ключи совпадают:

```
class Team {
    public uint ID;
    public string Name;
    public Team(uint aID, string s2) {
        ID = aID; Name = s2;
    }
}

class TeamCity {
    public uint ID;
    public string City;
    public TeamCity(uint aID, string s2) {
        ID = aID; City = s2;
    }
}
```

# Соединение I:I(продолжение)

```
class Result {  
    public string City;  
    public string Name;  
    public Result(string s1, string s2) {  
        City = s1; Name = s2;  
    }  
}
```

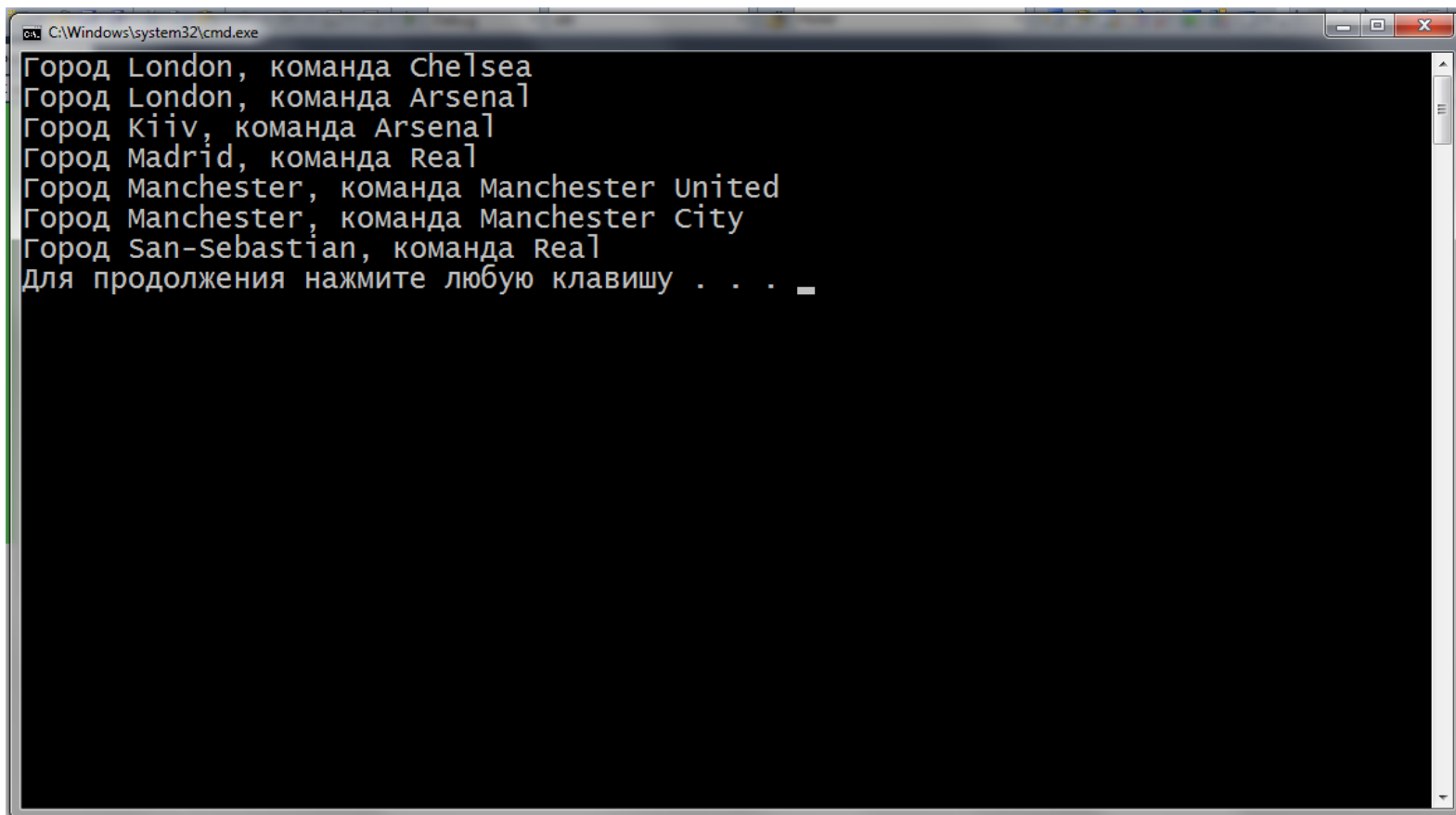
```
class Program {  
    static void Main(string[] args) {  
        Team[] tArr = { new Team(1, "Chelsea"),  
                        new Team(2, "Arsenal"),  
                        new Team(3, "Arsenal"),  
                        new Team(4, "Real"),  
                        new Team(5, "Manchester United"),  
                        new Team(6, "Manchester City"),  
                        new Team(7, "Real"),  
                        new Team(8, "BATE")  
        };  
    }  
}
```

# Соединение I:I (продолжение)

```
TeamCity[] tcArr = { new TeamCity(1, "London"),  
    new TeamCity(2, "London"),  
    new TeamCity(3, "Kiiv"),  
    new TeamCity(4, "Madrid"),  
    new TeamCity(5, "Manchester"),  
    new TeamCity(6, "Manchester"),  
    new TeamCity(7, "San-Sebastian")  
};  
  
var rez = from t in tcArr  
    join tc in tcArr  
    on t.ID equals tc.ID  
    select new Result(tc.City, t.Name);  
  
foreach (var i in rez)  
    Console.WriteLine("Город {0}, команда {1}",  
        i.City, i.Name);  
  
} // Main  
} // Program
```



# Соединение I:I (результат)



```
C:\Windows\system32\cmd.exe
Город London, команда Chelsea
Город London, команда Arsenal
Город Kiev, команда Arsenal
Город Madrid, команда Real
Город Manchester, команда Manchester United
Город Manchester, команда Manchester City
Город San-Sebastian, команда Real
для продолжения нажмите любую клавишу . . . _
```

# Анонимные типы

В C# предоставляется средство, называемое анонимным типом и связанное непосредственно с LINQ. Как подразумевает само название, анонимный тип представляет собой класс, не имеющий имени. Его основное назначение состоит в создании объекта, возвращаемого оператором `select`.

Тип возвращаемого объекта зачастую требуется только в самом запросе и не используется в остальной части программы. Благодаря анонимному типу в подобных случаях отпадает необходимость объявлять класс, который предназначается только для хранения результата запроса.

# Анонимные типы (продолжение)

Анонимный тип объявляется с помощью следующей общей формы:

```
new { имя_A = значение_A, имя_B = значение_B, ... }
```

где имена обозначают идентификаторы, которые преобразуются в свойства, доступные только для чтения и инициализируемые значениями.

В предыдущем примере мы могли бы записать

```
select new { City = tc.City, Name = t.Name };
```

и обойтись без задания класса **Result**. Более того, если не возникает двусмысленности, можно не задавать имена свойств:

```
select new { tc.City, t.Name };
```

# Соединение 1:M

Гораздо более частой является ситуация, когда источники данных связаны отношением «один ко многим»: ключ одного (главного) источника повторяется в другом (подчинённом) несколько раз. В этом случае выполняется миграция ключа из главного в подчиненный источник в качестве внешнего ключа.

```
class Team {
    public uint ID; // код команды
    public uint cityID; // код города (внешний ключ)
    public string Name;
    public Team(uint aID, uint acID, string s2){
        ID = aID; cityID = acID; Name = s2;
    }
}

class City {
    public uint ID; // код города
    public string City;
    public City(uint aID, string s2) {
        ID = aID; City = s2;
    }
}
```

# Соединение 1:М (заполнение ИСТОЧНИКОВ данных)

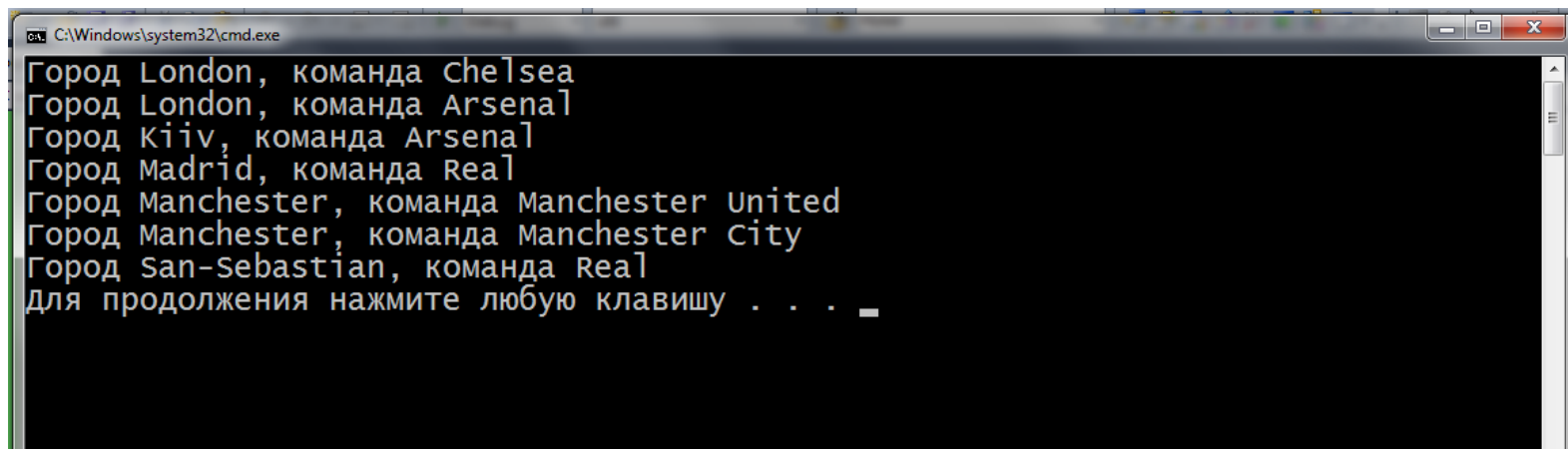
```
Team[] tArr = { new Team(1, 1, "Chelsea"),  
                new Team(2, 1, "Arsenal"),  
                new Team(3, 2, "Arsenal"),  
                new Team(4, 3, "Real"),  
                new Team(5, 4, "Manchester United"),  
                new Team(6, 4, "Manchester City"),  
                new Team(7, 5, "Real"),  
                new Team(8, 0, "BATE")  
            };  
City[] cArr = { new City(1, "London"),  
                new City(2, "Kiiv"),  
                new City(3, "Madrid"),  
                new City(4, "Manchester"),  
                new City(5, "San-Sebastian")  
            };
```

# Соединение I:M (реализация)

Для реализации соединения I:M используется комбинация фраз `join` и `into`:

```
var rezt = from c in cArr
            join t in tArr
            on c.ID equals t.cityID
            into tmp
            select new { c.Name, tmp };
foreach (var i in rezt)
    foreach (var j in i.tmp)
        Console.WriteLine("Город {0}, команда {1}",
                            i.Name, j.Name);
```

# Соединение I:M (результат)



```
C:\Windows\system32\cmd.exe
Город London, команда Chelsea
Город London, команда Arsenal
Город Kiyv, команда Arsenal
Город Madrid, команда Real
Город Manchester, команда Manchester United
Город Manchester, команда Manchester City
Город San-Sebastian, команда Real
для продолжения нажмите любую клавишу . . . _
```

Результат сгруппирован по городам (в предыдущем примере так получилось случайно)

# Nullable types (обнуляемые типы)

*Обнуляемый тип* — это особый вариант значимого типа. Помимо значений, определяемых базовым типом, обнуляемый тип позволяет хранить пустые значения (**null**). Следовательно, обнуляемый тип имеет такой же диапазон представления чисел и характеристики, как и его базовый тип. Он предоставляет дополнительную возможность обозначить значение, указывающее на то, что переменная данного типа не инициализирована.

Для обозначения обнуляемого типа необходимо записать знак вопроса после имени базового типа:

```
int? a;
```



# Nullable types (продолжение)

Обнуляемый тип совместим по операциям с базовым, но результат также будет обнуляемым типом:

```
int? a, c;  
int b;  
a = 10; b = 15;  
c = a + b; // результат - 25
```

```
int? a, c;  
int b;  
a = null; b = 15;  
c = a + b; // результат - null
```

```
int? a;  
int b, c;  
a = null; b = 15;  
c = a + b; // ошибка компиляции - необходим downcast
```

# Nullable types (проверка на пустоту)

Первый способ:

```
int? a;  
...  
if (a != null) { ... }
```

Второй способ:

```
int? a;  
...  
if (a.HasValue()) { ... }
```

Третий способ (задание значения по умолчанию):

```
int? a;  
int b;  
...  
b = (int) a ?? 0;
```

# Пример использования Nullable types

```
class Team {  
    public uint ID; // код команды  
    public uint? cityID; // код города (внешний ключ)  
    public string Name;  
    public Team(uint aID, uint? acID, string s2) {  
        ID = aID; cityID = acID; Name = s2;  
    }  
}  
  
...  
Team[] tArr = { new Team(1, 1, "Chelsea"),  
    new Team(2, 1, "Arsenal"),  
    new Team(3, 2, "Arsenal"),  
    new Team(4, 3, "Real"),  
    new Team(5, 4, "Manchester United"),  
    new Team(6, 4, "Manchester City"),  
    new Team(7, 5, "Real"),  
    new Team(8, null, "BATE")  
};
```