

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

Технология доступа к базам данных ADO.NET

Урок №5

Введение в LINQ

Содержание

LINQ	3
Общие сведения	3
LINQ to Objects в синтаксисе запросов	4
LINQ to Objects в синтаксисе методов	12
Краткий обзор LINQ To SQL.....	27
Краткий обзор LINQ To XML	33
Итоги	40

Общие сведения

Вы уже практически освоили технологию ADO.NET и понимаете ее назначение. Эта технология предназначена для доступа к данным, хранящимся в разных реляционных базах данных. Для общения с реляционными БД применяется язык SQL. Это очень эффективный язык программирования, отличающийся высокой скоростью обработки данных и универсальностью. С другой стороны, в тех приложениях, которые мы писали, использовался язык C#. В C# мы создавали объекты, позволяющие подключаться к поставщикам данных, передавать в БД запросы, получать результаты выполнения запросов и обрабатывать полученные результаты.

В какой-то момент у разработчиков C# возникли вопросы:

- можно ли сделать так, чтобы в самом C# была реализована возможность доступа к наборам данных и обработка этих данных?
- можно ли сделать так, чтобы доступ к наборам данных был однотипен, независимо от того, хранятся ли эти данные в какой-либо БД, или в XML-файле или в какой-либо коллекции в оперативной памяти?
- можно ли сделать так, чтобы работа с данными была максимально эффективна?

Ответом на все эти вопросы стало расширение языка C#, появившееся в платформе .NET Framework 3.5

и получившее название LINQ (Language Integrated Query). С этим расширением мы и познакомимся в нашем уроке. LINQ в значительной мере создавался под воздействием языка запросов SQL, отлично зарекомендовавшего себя при работе с данными в реляционных БД.

LINQ предназначен для работы с очень большими коллекциями объектов, предлагая для этого механизм создания запросов на языке C#. Для этих целей LINQ вводит в C# целый ряд методов расширения, которые позволяют не только выбирать из коллекции требуемые данные, но также выполнять с этими данными такие операции, как сортировка, группировка и агрегационные действия.

На сегодняшний день в C# существует несколько разновидностей LINQ. Наибольшее распространение получили такие:

- LINQ to Objects — предназначенный для работы с коллекциями, хранящимися в оперативной памяти;
- LINQ to SQL — предназначенный для работы с данными в реляционных БД;
- LINQ to XML — предназначенный для работы с данными в XML-файлах;

Сейчас мы рассмотрим основные возможности LINQ на примере LINQ to Objects, а затем рассмотрим остальные варианты LINQ.

LINQ to Objects в синтаксисе запросов

Создадим консольное приложение, в котором рассмотрим использование LINQ to Objects. В качестве источника данных можно использовать любую коллекцию,

например массив или список. В реальных условиях LINQ проявляет себя с наилучшей стороны при работе с очень большими коллекциями. Чем больше коллекция, тем более предпочтительным является использование LINQ. Для примера, мы будем работать с коллекцией скромного размера. Сначала это будет строковый массив. Затем мы рассмотрим работу с коллекцией объектов.

Приведите код созданного приложения к такому виду:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LinqTest1
{
    class Program
    {
        static void Main(string[] args)
        {
            LinqMethod1();
        }

        private static void LinqMethod1()
        {
            //массив - источник данных
            string[] countries = {"Albania", "UK",
                                "Lithuania", "Andorra", "Austria",
                                "Latvia", "Liechtenstein", "Switzerland",
                                "Ireland", "Sweden", "Italy", "France",
                                "Liechtenstein", "Spain", "Turkey", "Germany",
                                "Switzerland", "Monaco", "Montenegro",
                                "Norway", "Finland", "Turkey", "UK", "Poland",
                                "Portugal", "Lithuania", "Luxembourg"};
        }
    }
}
```

```

//LINQ запрос
var result = from c in countries
              where c.StartsWith("L")
              select c;

Console.WriteLine("Countries beginning with L:");
//продолжение работы с запросом
foreach (var item in result)
{
    Console.WriteLine(item);
}
Console.Write("Press Enter to complete");
Console.ReadLine();
}
}
}

```

Обратите внимание, что в нашем массиве с названиями стран некоторые страны повторяются. Скоро мы используем этот момент. Также отметьте тот факт, что подключение пространства имен `System.Linq` присутствует в проекте изначально. Это говорит о важности LINQ, которую ему уделяют разработчики. В этом пространстве имен располагаются классы и методы расширения, из которых состоит LINQ. Давайте поговорим о самой необычной строке в коде метода `LinqMethod1()`:

```

var result = from c in countries where c.StartsWith("L")
              select c;

```

Это и есть LINQ запрос, написанный на языке C#. В глаза сразу бросается структура этой строки «from ... where ... select», напоминающая SQL-запросы, но это

все-таки C#. Результат этого запроса заносится в переменную `result`, тип которой описан спецификатором `var`. `Var` — это новый спецификатор, обозначающий обобщенный тип данных, т.е. тип данных, который определяется динамически во время присваивания результата. Это очень удобный способ описывать типы, которые определяются динамически. В нашем случае, единственное, что можно сказать о типе переменной `result`, это то, что он будет производным от интерфейса `IEnumerable<>`. Дальнейшая детализация типа зависит от типа источника данных и от самого запроса.

Давайте сделаем небольшое отступление и поговорим об императивном и декларативном программировании. Начнем с примера. У вас есть таблица `MyTable` с каким-то количеством строк под управлением MS SQL Server. Кроме таблицы у вас есть в программе на C# коллекция объектов какого-либо класса. Вам надо вывести на экран все строки из таблицы и все объекты из коллекции. Вы, не задумываясь, напишете для таблицы запрос `select * from MyTable`. А для коллекции вы напишете цикл `foreach`. Или `while`? Или `for`? А давайте здесь немного задумаемся. Чем отличается запрос `select` от какого-либо цикла? Запрос содержит в себе описание того, что мы хотим получить. Но, при этом, запрос никак не уточняет, каким образом вы хотите получить результат. Мы все привыкли, что детали выполнения SQL запросов зависят от того, как эти запросы реализованы разработчиками СУБД. Мы подозреваем, что, возможно, запрос `select`

на MS SQL Server выполняется не так, как на Oracle. Результат в обоих случаях одинаков. А как обстоит дело с циклами? Здесь мы все должны сделать самостоятельно — выбрать цикл, решить, как отображать объекты нашего класса и перегрузить в этом классе ToString(), определить, по какому условию остановить выполнение цикла и т.п. Другими словами, мы пишем пошаговый набор инструкций на языке C# для вывода элементов нашей коллекции. В случае с использованием цикла, мы сами определяем, как получить, требуемый нам результат. В случае с таблицей мы использовали декларативное программирование, указав, что нам надо, но не объясняя СУБД, как она должна получить требуемый результат. В случае с коллекцией мы использовали императивное программирование, описывая данные, которые хотели получить. Такие языки разметки, как HTML и XML, язык XAML и конечно же SQL — являются типичными представителями декларативного программирования. Традиционные языки программирования C++, C#, Java — представители императивного программирования. Но, это деление условно. Т.к. внутри того же C# есть много элементов декларативного вида. Например, строка `Array.Sort(coll)`, выполняющая сортировку коллекции `coll` обладает всеми чертами декларативности. Мы не указываем, каким алгоритмом выполнять сортировку, в каком направлении, а полагаемся на настройки в классе `Array`. А вот если мы сами напишем функцию для сортировки такой коллекции, то это уже будет императивный подход.

Вам надо привыкать к тому, что язык LINQ — декларативный. Еще раз повторим, что при декларативном программировании мы указываем, «что» мы хотим делать, а вот «как» должно выполняться указанное действие — система выбирает сама. В пространстве имен `System.Linq` есть много специализированных классов для хранения результатов запросов и LINQ сам определит оптимальным образом тип для переменной `result` в нашем примере. Поэтому, если кому-то из вас хочется все-таки указать тип этой переменной явно, смиритесь с тем, что LINQ сделает это лучше вас.

Для любознательных, по секрету надо сообщить, тип этой переменной всегда будет производным от типа `IEnumerable<>`. В случае выборки из БД этот тип будет производным от типа `IQueryable<>`. Дальнейшая детализация типа зависит от типа выбираемых элементов, от типа коллекции, от используемых в запросе инструкций и даже от вида LINQ. Например, если вы будете сортировать свою выборку, то тип будет производным от `IOrderedQueryable<>`. А еще этот тип может изменяться компилятором при оптимизации. В любом случае, вы никогда не должны полагаться на точный тип этой переменной! Если вам хочется посмотреть на этот тип, можете использовать `GetType()`.

Анализируем запрос дальше. В строке «`from c in countries`» после спецификатора «`in`» указывается источник данных. В нашем случае — это строковый массив `countries`. Какой объект может быть источником данных

для LINQ to Objects? Единственное требование к этому объекту, чтобы он был производным от интерфейса `IEnumerable<>`. Массивы и любые другие коллекции отвечают этому требованию. Поскольку источник данных это коллекция, элементы которой перебираются в ходе выполнения запроса, то переменная «с» предназначена для хранения текущего элемента коллекции на каждой итерации. Понятно, что имя этой переменной может быть любым.

Затем, после спецификатора «where» мы указываем условие, по которому отбираем элементы просматриваемой коллекции. В нашем примере мы отбираем из массива названия стран, начинающиеся с буквы «L». Обратите внимание, что для формирования условия мы используем стандартный метод `StartsWith()`, определенный в C# для типа `string`.

И, наконец, спецификатор «select» указывает, что мы отбираем текущий элемент, находящийся в переменной «с» в результат запроса, если этот элемент отвечает заданному условию отбора.

А теперь, несколько слов о цикле `foreach`, в котором мы выводим результат выполнения запроса. Дело в том, что этот цикл является частью запроса. Он реализует, так называемое «отложенное выполнение». Обратите внимание, что именно этот цикл фактически и выполняет сам запрос! До тех пор, пока мы не обратимся к результатам LINQ запроса, данные запроса не будут извлекаться из источника данных. А обращаемся мы к результатам запроса именно в цикле `foreach`.

Запустите приложение и посмотрите на результат. У меня он выглядит так:

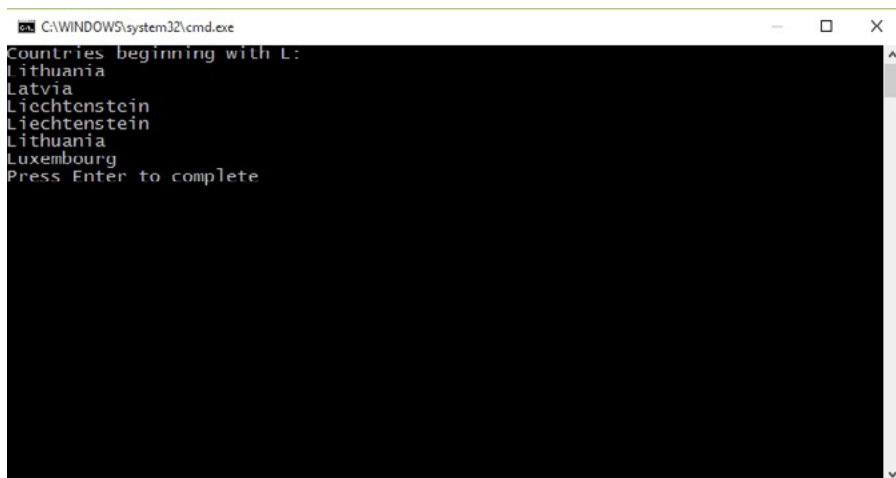


Рис. 3. Результат выполнения LINQ запроса

Поменяйте условие в запросе на такое:

```
var result = from c in countries where c.Length >
              10 select c
```

и выполните приложение снова. В этом случае вы увидите в результирующем наборе только страны, названия которых длиннее десяти символов.

Мы отметили в начале этой главы тот момент, что в массиве стран есть повторяющиеся названия. Вы видите в полученной коллекции эти повторы. Перепишите запрос таким образом:

```
var result = (from c in countries where c.StartsWith("L")
              orderby c.Length descending select c).Distinct();
```

Как и следовало ожидать, повторы исчезли.

LINQ to Objects в синтаксисе методов

Сейчас вам надо запомнить, что запрос, который мы рассмотрели выше, написан с использованием синтаксиса запросов LINQ. Синтаксис запросов — не единственный способ писать запросы LINQ. Есть еще другой способ — синтаксис методов. Давайте создадим в нашем приложении еще один метод, в котором напишем такой же запрос к нашему массиву, но в синтаксисе методов. Затем будем сравнивать эти два способа написания запросов, изменяя и дополняя наши методы.

LINQ реализован, как набор расширяющих методов для объектов, производных от интерфейса `IEnumerable` (массивы, коллекции и др.). Если вы в Visual Studio наберете имя нашего массива `countries`, а затем введете точку и посмотрите на список IntelliSense, то увидите ряд generic методов, таких как `Where<>`, `Union<>`, `Takes<>` и другие. Это все — методы LINQ. Когда вы пишете LINQ запрос, указывая спецификаторы вроде «`from ... where ... select`», компилятор подставляет вместо них соответствующие методы расширения.

Давайте перепишем наш запрос в синтаксисе методов. Вообще говоря, синтаксис запросов является более предпочтительным при написании LINQ запросов. Он выглядит более читабельно. Однако, полезно иметь представление и о синтаксисе методов, тем более, что изредка бывают ситуации, когда синтаксис методов может сделать то, что нельзя сделать в синтаксисе запросов.

Например, вы хотите получить элемент коллекции с максимальным значением свойства `Price`. Или же вы

хотите получить количество элементов с ценами в диапазоне от Price1 до Price2. Вы уже догадались, о чем идет речь? Об агрегационных функциях. Если вы хотите выполнить запрос с агрегационным результатом, надо использовать синтаксис методов. Давайте напишем такие запросы:

```
//есть класс, описывающий товар
class Product
{
    public string Name { get; set; }
    public int Price { get; set; }
}
//создаем коллекцию товаров с помощью какого-то
//метода GetProducts()
List<Product> products = GetProducts();
```

Решит ли стоящую перед нами задачу такой запрос?

```
var maxPrice = products.Select(p => p.Price).Max();
```

Конечно же нет. Этот запрос вернет максимальную цену из нашей коллекции товаров, но не объект с максимальной ценой. Сам объект можно получить так:

```
int maxPrice = products.Max(p => p.Price);
var item = products.First(x => x.Price == maxPrice);
```

Результат получен, но ценой двух запросов. Давайте рассмотрим еще одно решение. Вы можете подключить пакет расширения для LINQ с именем `morelinq`. Для установки этого пакета надо в консоли диспетчера пакетов ввести команду `Install-Package morelinq`. В этом расширении есть полезный метод `MaxBy()`:

```
var maxItem = products.MaxBy(p => p.Price);
```

Чтобы определить количество товаров с ценами в диапазоне от 100 до 1000 можно выполнить такой запрос:

```
var prodCount = products.Count(p => p.Price > 100 &&  
    p.Price <= 1000);
```

С синтаксисом методов связана одна деталь, которая часто отпугивает от применения этого вида написания запросов. Дело в том, что большинство методов LINQ используют в качестве параметров методы или функции. Для передачи таких параметров надо использовать делегаты, но LINQ предпочитает в этом случае использовать лямбда-выражения.

Лямбда-выражения — это краткий способ записи анонимных методов или функций. Характерным символом лямбда-выражения является `=>`, который разделяет параметры и условие отбора. Общий вид лямбда-выражения выглядит так:

(параметры) `=>` выражение

Например:

- `x => x > 100`, описывает функцию, которая возвращает `true`, если параметр больше 100, и `false` — в противном случае;
- `(x,y) => x+y`, описывает функцию, которая возвращает сумму своих параметров;
- `x => x.StartsWith("L")`, узнаете? Эта функция возвращает `true`, если параметр начинается со строки "L", и `false` — в противном случае;

Параметров у лямбда-выражения может быть произвольное количество и, как правило, их надо заключать

в скобки. Скобки можно опускать только в случае одного параметра. Но, лучше использовать их и в этом случае. Как правило, компилятор всегда сам старается определить тип параметров лямбда-выражения, но, если для него это может оказаться сложным, можно указывать тип параметров явно:

- `(int x, string s) => s.Length > x`, эта функция отбирает строки длиной больше `x`.

Вы можете использовать лямбда-выражения и в обычном коде для инициализации делегатов, не только в LINQ:

```
delegate int del(int i);
static void TestLambda()
{
    del myDelegate = (x) => x * x;
    int sq = myDelegate(5); //sq = 25
}
```

Но мы сконцентрируемся на применении лямбда-выражений в LINQ запросах.

Добавьте в наш проект такой метод и вызовите его в `Main()` вместо предыдущего метода:

```
private static void LinqMethod2 ()
{
    //массив - источник данных
    string[] countries = { "Albania", "UK",
        "Lithuania", "Andorra", "Austria", "Latvia",
        "Liechtenstein", "Switzerland", "Ireland",
        "Sweden", "Italy", "France", "Liechtenstein",
        "Spain", "Turkey", "Germany", "Switzerland",
        "Monaco", "Montenegro", "Norway", "Finland",
        "Turkey", "UK", "Poland", "Portugal",
        "Lithuania", "Luxembourg"
    };
}
```

```
//LINQ запрос
var result = countries.Where(c => c.StartsWith("L"));

Console.WriteLine("Countries beginning with L:");
//продолжение работы с запросом
foreach (var item in result)
{
    Console.WriteLine(item);
}
Console.Write("Press Enter to complete");
Console.ReadLine();
}
```

Результат выполнения этого метода будет таким же, как и предыдущего. Так и должно быть, поскольку мы просто записали тот же запрос в другом виде. Ви видите в выделенной строке вызов метода `Where()` от имени источника данных (нашего массива). В качестве параметра методу `Where()` передается лямбда-выражение, содержащее анонимную функцию предикат для отбора значений. Это лямбда-выражение преобразовывается компилятором в анонимный метод, который выполняется методом `Where()` на каждом элементе коллекции. Если этот анонимный метод возвращает `true`, элемент включается в результирующий набор, если `false` — элемент в набор не включается.

Приведите код запроса в первом методе к такому виду и запустите приложение:

```
var result = from c in countries where c.StartsWith("L")
              orderby c.Length select c;
```

Мы добавили требование сортировки результирующего набора данных по длине названия страны, и как вы

должны увидеть, результирующий набор отсортирован. Хотите отсортировать по убыванию, добавьте спецификатор `descending`:

```
var result = from c in countries where c.StartsWith("L")
              orderby c.Length descending select c;
```

В синтаксисе методов такие сортировки будут выглядеть так:

```
var result = countries.OrderBy(c => c.Length)
                       .Where(c => c.StartsWith("L"));
```

или, для сортировки по убыванию:

```
var result = countries.OrderBy(c => c.Length)
                       .Where(c => c.StartsWith("L"));
```

Существует ли `Distinct()` в синтаксисе методов? Пожалуйста:

```
var result = countries.OrderBy(c => c.Length)
                       .Where(c => c.StartsWith("L")).Distinct();
```

LINQ поддерживает выполнение стандартных агрегационных операций. Для этого соответствующие методы надо вызывать от имени результирующего набора. Например:

```
Console.WriteLine(result.Max());
Console.WriteLine(result.Min());
Console.WriteLine(result.Count());
Console.WriteLine(result.Average());
Console.WriteLine(result.Sum());
```

Правда, `Average()` и `Sum()` имеют смысл только для числовых коллекций. Мы рассмотрели базовые возможности

LINQ to Objects для простого набора данных. Давайте теперь используем в качестве источника данных коллекцию каких-либо объектов.

Добавьте в состав проекта такой класс:

```
class Product
{
    public string Name { set; get; }
    public int Price { set; get; }
    public string Manufacturer { set; get; }
    public int Count { set; get; }

    public override string ToString()
    {
        return String.Format("{0} {1} {2} {3}", this.Name,
            this.Price, this.Manufacturer, this.Count);
    }
}
```

После этого добавьте в главный класс приложения еще один метод:

```
private static void LinqMethod3()
{
    //снова используем этот массив, но уже не как источник данных
    string[] countries = { "Albania", "UK", "Lithuania",
        "Andorra", "Austria", "Latvia",
        "Liechtenstein", "Switzerland", "Ireland",
        "Sweden", "Italy", "France", "Liechtenstein",
        "Spain", "Turkey", "Germany", "Switzerland",
        "Monaco", "Montenegro", "Norway", "Finland",
        "Turkey", "UK", "Poland", "Portugal",
        "Lithuania", "Luxembourg"
    };
    //теперь источником данных будет список объектов
    List<Product> products = new List<Product>();
}
```

```

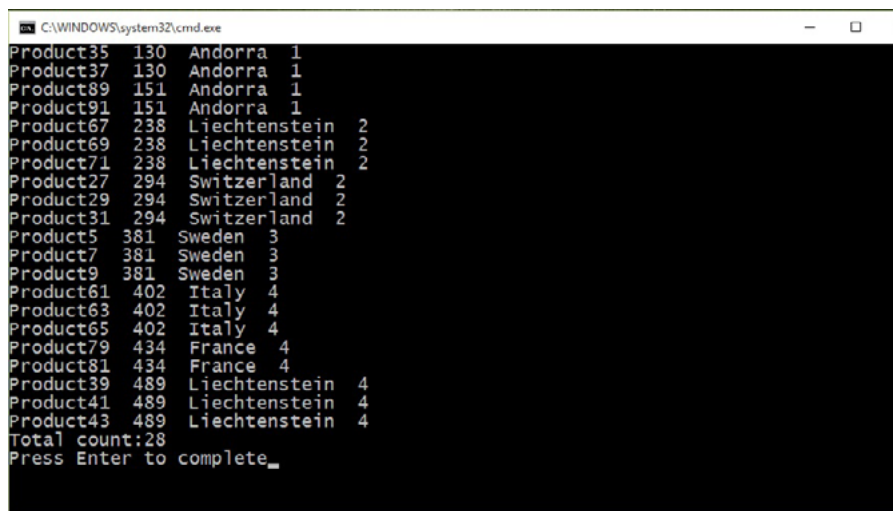
//генерируем список из 100 случайных объектов
for (int i= 0; i<100; i++)
{
    //Sleep() нужен, чтобы компилятор на каждой
    //итерации создавал новый объект Random,
    //в противном случае, компилятор использует
    //оптимизацию и не меняет объект Random для
    //всех итераций
    Thread.Sleep(5);
    products.Add(new Product { Name = "Product" + (++i),
        Price = (new Random()).Next(0,1000),
        Manufacturer =
            countries[(new Random()).
                Next(0, countries.Length - 1)],
        Count = (new Random()).Next(0, 10) });
}
//если хотите просмотреть созданный список
Console.WriteLine("All products:");
foreach (var item in products)
{
    Console.WriteLine(item);
}

//LINQ запрос к списку объектов
var result = from c in products
    where c.Price<500
    orderby c.Price
    select c;

Console.WriteLine("\nProducts with price less than 500:");
//продолжение работы с запросом
foreach (var item in result)
{
    Console.WriteLine(item);
}
Console.WriteLine("Total count:"+result.Count());
Console.Write("Press Enter to complete");
Console.ReadLine();
}

```

Запустите приложение, вызвав в Main() новый метод. У меня получился такой результат:



```

C:\WINDOWS\system32\cmd.exe
Product35 130 Andorra 1
Product37 130 Andorra 1
Product89 151 Andorra 1
Product91 151 Andorra 1
Product67 238 Liechtenstein 2
Product69 238 Liechtenstein 2
Product71 238 Liechtenstein 2
Product27 294 Switzerland 2
Product29 294 Switzerland 2
Product31 294 Switzerland 2
Product5 381 Sweden 3
Product7 381 Sweden 3
Product9 381 Sweden 3
Product61 402 Italy 4
Product63 402 Italy 4
Product65 402 Italy 4
Product79 434 France 4
Product81 434 France 4
Product39 489 Liechtenstein 4
Product41 489 Liechtenstein 4
Product43 489 Liechtenstein 4
Total count:28
Press Enter to complete_

```

Рис. 4. Результат выполнения LINQ запроса к списку объектов

Поиграйтесь с разными условиями отбора. Попробуйте выполнить запрос к нашему списку, написанный в синтаксисе методов. У вас все должно получиться. Сейчас мы можем сделать вывод, что LINQ запросы, похоже, замечательно работают с коллекциями пользовательских объектов. Но именно в этом случае, LINQ позволяет делать еще больше. Продолжим рассмотрение нашего примера и уделим более пристальное внимание спецификатору `select`. Измените запрос таким образом:

```

var result = from c in products where c.Price<500
              orderby c.Price
              select c.Name;

```

Теперь в итоговом цикле вы увидите только названия товаров. К результатам запроса можно применять еще и приемлемые трансформации:

```
var result = from c in products where c.Price<500
              orderby c.Price
              select c.Name.ToUpper();
```

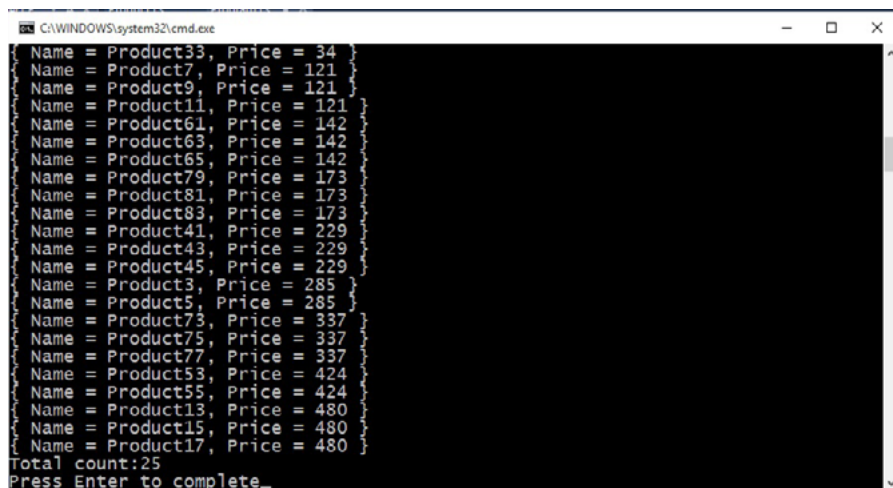
В этом случае названия товаров будут выведены в верхнем регистре. Давайте попробуем такой запрос, в котором мы хотим видеть названия и цены отобранных товаров:

```
var result = from c in products where c.Price<500
              orderby c.Price
              select c.Name, c.Price;
```

Упс. Этот запрос вызвал ошибку компиляции. Дело в том, что после спецификатора `select` можно указывать только один элемент. Но если вы перепишите запрос таким образом:

```
var result = from c in products where c.Price<500
              orderby c.Price
              select new { c.Name, c.Price };
```

то, в `select` будет один объект. Попробуйте запустить приложение с этим запросом. У меня получился такой результат:



```

C:\WINDOWS\system32\cmd.exe
Name = Product33, Price = 34
Name = Product7, Price = 121
Name = Product9, Price = 121
Name = Product11, Price = 121
Name = Product61, Price = 142
Name = Product63, Price = 142
Name = Product65, Price = 142
Name = Product79, Price = 173
Name = Product81, Price = 173
Name = Product83, Price = 173
Name = Product41, Price = 229
Name = Product43, Price = 229
Name = Product45, Price = 229
Name = Product3, Price = 285
Name = Product5, Price = 285
Name = Product73, Price = 337
Name = Product75, Price = 337
Name = Product77, Price = 337
Name = Product53, Price = 424
Name = Product55, Price = 424
Name = Product13, Price = 480
Name = Product15, Price = 480
Name = Product17, Price = 480
Total count:25
Press Enter to complete.

```

Рис. 5. Выполнение LINQ запроса с созданием новых объектов

В этом случае LINQ запрос вернул коллекцию новых объектов, созданных по заданному в select образцу. Вы можете догадаться об этом, даже по внешнему виду выведенных объектов. Попробуйте написать такой же запрос, но в синтаксисе методов:

```
var result = products.Where(p => p.Price < 500).
    Select(p => new { p.Name, p.Price });
```

Запустите приложение и убедитесь в том, что этот запрос работает, как и предыдущий LINQ запрос в синтаксисе запросов.

В синтаксисе SQL некоторые БД поддерживают такое выражение, как TOP N, позволяющее отобрать из результата запроса select первые N строк. Например, у вас есть таблица Books с большим количеством строк, а вас интересуют только первые 10 строк из результата

выполнения запроса `select * from Books`. Тогда вы можете написать такой запрос:

```
select TOP 10 * from Books
```

и вы получите только 10 строк из таблицы `Books`. Это особенно удобно, если вы, например, хотите получить 10 самых дешевых книг. Для этого напишите такой запрос:

```
select TOP 10 * from Books order by Price
```

В этом случае вы получите первые 10 строк из отсортированного по ценам списка книг. Правда учтите, что это может и не быть 10 книг с 10 разными ценами. Это будет ровно 10 строк и все эти 10 строк могут быть с одинаковой минимальной ценой, могут быть и с разными ценами.

В LINQ есть методы, выполняющие аналогичную работу. Это методы `Take()` и `Skip()`. Эти методы относятся к разбивающим операциям (*partitioning operations*) и позволяют делать выборки из результирующего набора, противоположными способами. Оба эти метода применяются к результирующему набору. Метод `Take()` позволяет выбрать указанное количество элементов, а метод `Skip()` — пропустить указанное количество, и выбрать остальные элементы. Посмотрим, как это работает. Измените последний метод в нашем приложении к такому виду:

```
private static void LinqMethod3()  
{  
    //массив - источник данных
```

```

string[] countries = { "Albania", "UK", "Lithuania",
    "Andorra", "Austria", "Latvia",
    "Liechtenstein", "Switzerland", "Ireland",
    "Sweden", "Italy", "France", "Liechtenstein",
    "Spain", "Turkey", "Germany", "Switzerland",
    "Monaco", "Montenegro", "Norway", "Finland",
    "Turkey", "UK", "Poland", "Portugal",
    "Lithuania", "Luxembourg"
};

List<Product> products = new List<Product>();
for (int i= 0; i<100; i++)
{
    Thread.Sleep(5);
    products.Add(new Product { Name = "Product" + (++i),
        Price = (new Random()).Next(0,1000),
        Manufacturer = countries[(new Random()).
            Next(0, countries.Length - 1)],
        Count = (new Random()).Next(0, 10) });
}

var result = products.Where(p => p.Price > 800)
    Select(p => new { p.Name, p.Price });

Console.WriteLine("\nProducts with price greater
    than 800:");
//продолжение работы с запросом
foreach (var item in result)
{
    Console.WriteLine(item);
}

Console.WriteLine("Total count:"+result.Count());
Console.WriteLine("\nTop 5:\n");

foreach (var item in result.Take(5))
{
    Console.WriteLine(item);
}
Console.WriteLine("\nAfter top 5:\n");

```



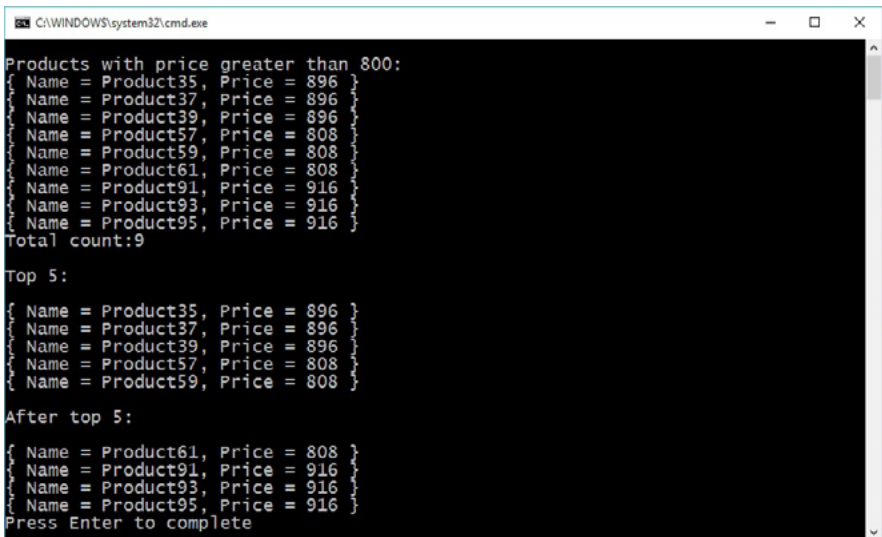
```

foreach (var item in result.Skip(5))
{
    Console.WriteLine(item);
}

Console.Write("Press Enter to complete");
Console.ReadLine();
}

```

Как вы видите, изменения коснулись в основном вывода результатов. Теперь мы выводим результаты в трех циклах. В первом — весь результирующий набор, во втором — первые пять строк, в третьем — оставшиеся, после первых пяти строки. Именно во втором и третьем циклах мы и используем методы `Take()` и `Skip()`. Еще мы немного изменили условие отбора, чтобы уменьшить количество объектов в выходной коллекции, чтобы все результаты поместились в одном окне. У меня получился такой результат:



```

C:\WINDOWS\system32\cmd.exe
Products with price greater than 800:
{ Name = Product35, Price = 896 }
{ Name = Product37, Price = 896 }
{ Name = Product39, Price = 896 }
{ Name = Product57, Price = 808 }
{ Name = Product59, Price = 808 }
{ Name = Product61, Price = 808 }
{ Name = Product91, Price = 916 }
{ Name = Product93, Price = 916 }
{ Name = Product95, Price = 916 }
Total count:9

Top 5:
{ Name = Product35, Price = 896 }
{ Name = Product37, Price = 896 }
{ Name = Product39, Price = 896 }
{ Name = Product57, Price = 808 }
{ Name = Product59, Price = 808 }

After top 5:
{ Name = Product61, Price = 808 }
{ Name = Product91, Price = 916 }
{ Name = Product93, Price = 916 }
{ Name = Product95, Price = 916 }
Press Enter to complete

```

Рис. 6. Применение `Take()` и `Skip()`

Есть еще пара полезных методов, позволяющих выбрать первый встреченный в наборе элемент, отвечающий указанному условию. Это методы `First()` и `FirstOrDefault()`. Добавьте в последний метод такие строки:

```
var pr = products.First(p => p.Price > 500 &&  
    p.Manufacturer.StartsWith("L"));  
Console.WriteLine("\nFirst:" + pr);
```

При выполнении этого кода, вы можете увидеть на экране описание найденного элемента, но, если ни один элемент в `products` не будет отвечать указанным условиям, вы получите исключительную ситуацию.

А если вы добавите такие строки:

```
var pr = products.FirstOrDefault(p => p.Price > 500 &&  
    p.Manufacturer.StartsWith("L"));  
Console.WriteLine("\nFirst:" + pr);
```

то исключительной ситуации вы не получите. Вы увидите либо описание найденного элемента, либо `null`, если такого элемента в `products` не будет.

LINQ To Objects содержит еще много полезных и интересных методов и знакомство с ними потребует много времени. Будет полезно, если вы продолжите знакомство с этим инструментом, но уже самостоятельно. Цель этого урока, познакомить вас с LINQ и дать вам основные сведения об этом расширении языка C#. А нам предстоит еще кратко рассмотреть LINQ To SQL и LINQ To XML.

Краткий обзор LINQ To SQL

LINQ To SQL автоматически транслирует запросы LINQ в запросы SQL и позволяет работать с такими БД, как MS SQL Server и Oracle. Для этого решаются задачи ORM (объектно — реляционного отображения), когда в приложении создаются классы, соответствующие таблицам в БД. Все эти вопросы мы с вами подробно рассмотрим в наших дальнейших уроках, при изучении Entity Framework. Entity Framework на сегодняшний день отодвинул технологию LINQ To SQL на второй план. Хотя, правильнее будет сказать, что он интегрировал эту технологию в себя и развил ее еще больше.

Давайте рассмотрим пример использования LINQ To SQL для нашей БД. Создайте консольное приложение с именем LinqToSqlTest. Активируйте в этом проекте в меню

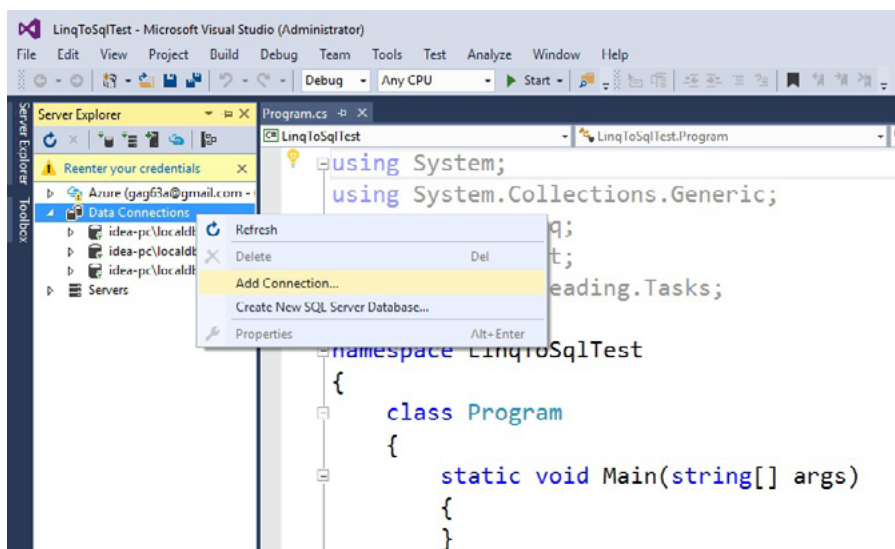


Рис. 7. Подключение к серверу

View опцию Server Explorer. В открывшемся окне Server Explorer выделите опцию Data Connections и выберите из контекстного меню команду Add Connection...

Затем выполните действие Add New Connection. В появившемся окне укажите адрес сервера, как (localdb)\v11.0 и выберите нашу БД. Все. Связь приложения с БД установлена.

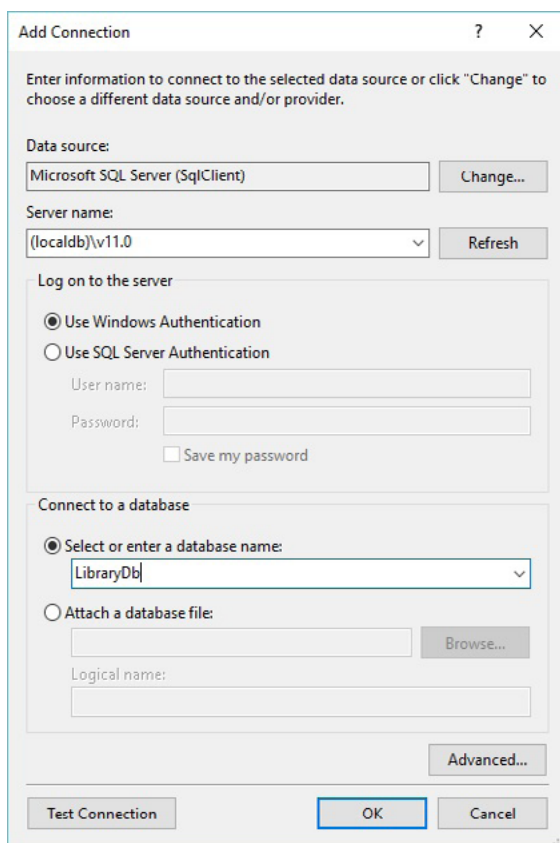


Рис. 8. Подключение к БД

Теперь выберите наше приложение в окне Solution Explorer и нажмите правую кнопку мыши. Выберите

действия Add — New Item — LINQ to SQL Classes. Это будет объект с расширением dbml, который поможет нам создать в приложении классы, соответствующие таблицам в БД.

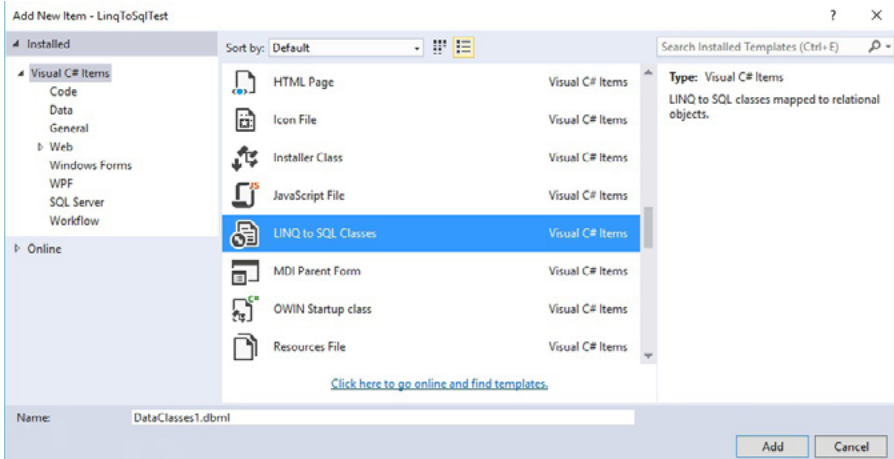


Рис. 9. Добавление LINQ to SQL Classes

После создания этого объекта ничего не изменяйте в окнах Visual Studio и вернитесь в окно Server Explorer. Раскройте узел созданного подключения, затем раскройте узел нашей БД, затем раскройте узел Tables, чтобы были видны наши таблицы. Перетащите из нашей БД и бросьте на добавленный элемент DataClasses1.dbml (который сейчас открыт в центральном окне) таблицы Books и Authors. После выполнения этих действий вы увидите диаграмму БД. На диаграмме будут отображены поля всех таблиц и связи между таблицами. Теперь элемент DataClasses1.dbml можно закрыть, выполнив при этом его сохранение.

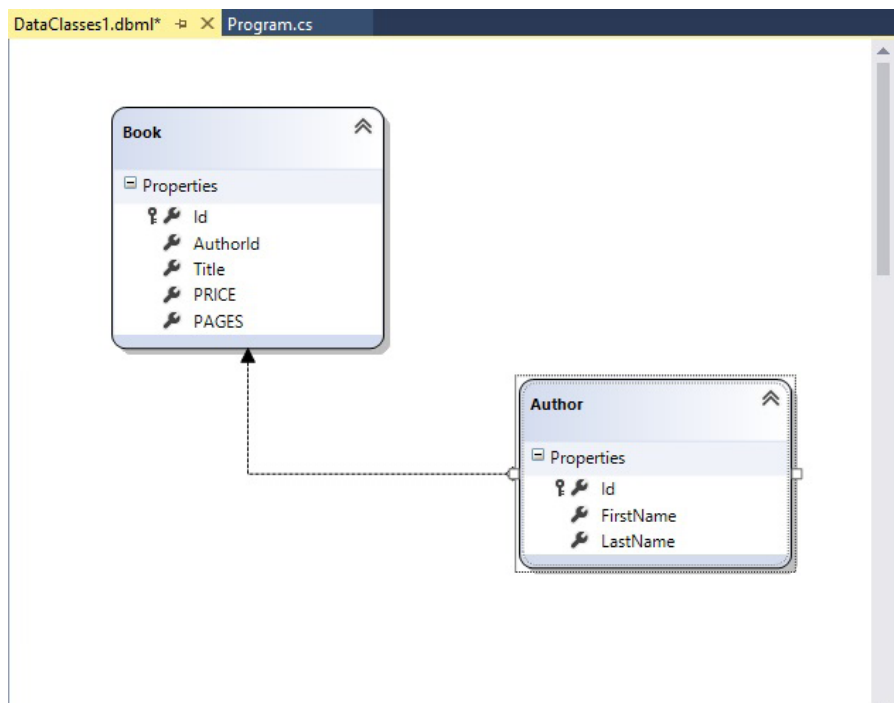


Рис. 10. Диаграмма БД

Теперь перейдите в окно обозревателя решений, раскройте в составе проекта узел `DataClasses1.dbml` и выделите в нем элемент `DataClasses1.designer.cs`. В этом элементе содержатся классы, которые студия автоматически создала в нашем приложении, когда мы перетаскивали таблицы из БД. Особо обратите внимание на такие классы:

`DataClasses1DataContext` производный от `System.Data.Linq.DataContext` — этот класс обычно называется контекстом БД и является средством доступа к БД, инструментом для передачи в БД запросов и получения из БД результатов выполнения этих запросов;

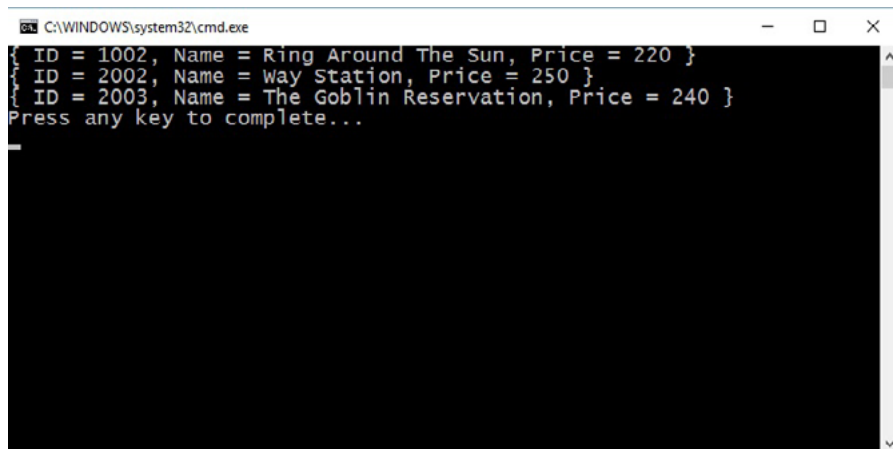
Book производный от интерфейса **INotifyPropertyChanged** — этот класс является в приложении образом таблицы Books из БД, свойства этого класса соответствуют полям таблицы. Мы в приложении будем работать с объектами этого класса, а LINQ To SQL с помощью контекста БД будет транслировать наши действия в БД;

Аналогично классу для таблицы Books в нашем приложении создан класс **Author**, соответствующий таблице Author.

Теперь наше приложение готово к работе с БД. Занесите в метод Main() такой код:

```
static void Main(string[] args)
{
    DataClasses1DataContext db =
        new DataClasses1DataContext();
    var queryResults = from c in db.Books
                        where c.PRICE > 200
                        select new
                        {
                            ID = c.Id,
                            Name = c.Title,
                            Price = c.PRICE
                        };
    foreach (var item in queryResults)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine("Press any key to complete...");
    Console.ReadLine();
}
```

Теперь выполните созданное приложение. У меня результаты выглядят таким образом:



```
C:\WINDOWS\system32\cmd.exe
{ ID = 1002, Name = Ring Around The Sun, Price = 220 }
{ ID = 2002, Name = Way Station, Price = 250 }
{ ID = 2003, Name = The Goblin Reservation, Price = 240 }
Press any key to complete...
```

Рис. 11. Результаты выполнения программы

Обсудим полученные результаты. Когда мы добавили в состав проекта объект LINQ to SQL Class с расширением dbml, мы получили возможность выполнить с помощью этого объекта целый ряд важных действий. Сначала в приложении создается класс, производный от System.Data.Linq.DataContext, который будет являться для приложения средством доступа к БД. При перетаскивании таблиц из БД на объект dbml в составе нашего приложения были созданы классы Book и Author, соответствующие таблицам. Свойства созданных классов соответствуют полям таблицы.

Код, размещенный в методе Main() является примером использования LINQ To SQL в формате запроса. Прежде всего мы создаем объект контекста БД. Через этот объект мы обращаемся к таблице Books, выбираем из нее книги с ценой, больше 200, и формируем для вывода новые объекты, которые заносим в переменную queryResults. Затем в цикле выводим полученные объекты.

Таким образом, LINQ To SQL позволяет нам работать с данными в таблицах реляционной БД в стиле ООП. Мы общаемся с классами, соответствующими таблицам в БД. Все действия, примененные к объектам этих классов, приводят к изменению состояния данных в БД. И все это — в стиле ООП с возможностью писать LINQ запросы на C#.

Краткий обзор LINQ To XML

Для чего применяется LINQ To XML? Это расширение не предназначено для решения задач, стоящих перед такими API, как XML DOM, XPath, XQuery, XSLT и другие. LINQ To XML просто предлагает дополнительные возможности для создания и опроса данных в формате XML. Да, вы все правильно поняли. LINQ To XML позволяет создавать XML документы и затем работать с ними. Ну, и, конечно же, можно загружать существующие XML файлы.

Создайте новый консольный C# проект, в котором мы рассмотрим основные возможности LINQ To XML. Сначала мы добавим в состав проекта метод, в котором создадим «с чистого листа» новый XML документ. Для создания таких документов, надо добавить в проект ссылку на пространство имен System.Xml.Linq. В этом пространстве имен находятся, так называемые функциональные конструкторы для создания XML объектов. Особенностью этих конструкторов является наглядность их использования. Вызовы этих конструкторов вкладываются один в другой, повторяя структуру создаваемого XML объекта. Таким образом, создание объекта позволяет

сразу увидеть его структуру. Добавьте в состав созданного проекта такой метод:

```
static void CreateXmlDocument()
{
    XmlDocument xmldoc = new XmlDocument(
        new XElement("computers",
            new XAttribute("Price", "800"),
            new XAttribute("Warranty", "2 years"),
            new XElement("CPU",
                new XAttribute("Name", "Intel Core i7-6700K"),
                new XAttribute("GHz", 2.5)
            ),
            new XElement("HDD",
                new XAttribute("Name", "Samsung 850 PRO"),
                new XAttribute("Size", 1.0)
            ),
            new XElement("computer",
                new XAttribute("Price", "900"),
                new XAttribute("Warranty", "2 years"),
                new XElement("CPU",
                    new XAttribute("Name", "AMD A10-5800K"),
                    new XAttribute("GHz", 2.5)
                ),
                new XElement("HDD",
                    new XAttribute("Name", "Transcend
ESD400"),
                    new XAttribute("Size", 1.0)
                )
            )
        );
    Console.WriteLine(xmldoc);
}
```

Вызовите этот метод в Main() и выполните созданный проект. Вы увидите на экране структуру созданного

XML объекта. Вызов конструктора `XDocument()` создает сам объект. Поскольку по законам XML, объект должен содержать корневой элемент, то дальше следует вызов конструктора `XElement()`. Конструктор `XElement()` создает как корневой элемент, так и элементы, вложенные в корневой элемент. В качестве параметра этот конструктор может принимать еще один конструктор — `XAttribute()`, предназначенный для создания атрибутов. Этот последний конструктор принимает в качестве параметра имя атрибута и его значение. Затем содержимое созданного объекта выводится на экран. Приведенный код говорит сам за себя и здесь нечего объяснять дополнительно. Единственное о чем надо сказать, это тот момент, что если в каком-либо элементе надо создать текстовый узел, то содержимое этого узла надо передать конструктору `XElement()` в качестве параметра. Например:

```
new XElement("computer",
    "This is not expensive and reliable computer",
    new XAttribute("Price", "800"),
    new XAttribute("Warranty", "2 years"),
    new XElement("CPU",
        new XAttribute("Name", "Intel Core i7-6700K"),
        new XAttribute("GHz", 2.5)
    )
)
```

Давайте посмотрим, каким образом можно созданный объект записать в файл. Для этого добавьте в созданный метод, после вывода XML объекта на экран такие строки:

```
xmlFilePath = @"example.xml";
xmlDoc.Save(xmlFilePath);
```

Выполните проект и перейдите в корневую папку приложения. Там вы увидите созданный XML-файл. Чтобы убедиться в том, что этот файл well formed, откройте его в Internet Explorer. Я увидел такую картину:

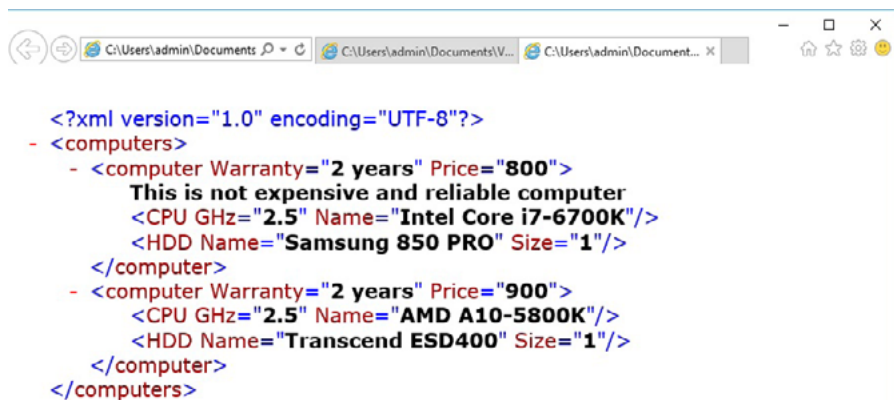


Рис. 12. Созданный XML файл

Для чтения существующего XML-файла надо выполнить такие действия:

```
XDocument xmldoc2 = XDocument.Load(xmlFilePath);
```

Т.е., надо создать XML объект типа XDocument, вызвать статический метод Load() от имени класса XDocument, указав ему в качестве параметра путь к существующему XML-файлу и присвоить результат выполнения этого метода созданному объекту. Существует еще возможность создать XML объект из строки, содержащей XML данные. Для этого используется метод Parse():

```
XDocument xmldoc3 = XDocument.Parse(xmlString);
```

В этом примере `xmlString` — это строка с XML данными.

Как вы понимаете `XDocument` представляет собой XML объект, который мы хотим использовать, как источник данных. У класса `XDocument` есть несколько очень важных свойств, которые помогут нам в доступе к содержимому. Это такие свойства, как `DescendantNodes` и `Descendants`. Первое свойство — это коллекция всех вложенных узлов (включая элементы, текстовые строки, комментарии), а второе — это коллекция только вложенных элементов. Именно эти две коллекции чаще всего и выступают, как источники данных при работе с XML объектами. Давайте посмотрим на пример использования этих свойств. Добавьте в наш проект такой метод:

```
static void ReadXmlDocument ()
{
    string xmlFilePath = @"example.xml";
    XDocument xmldoc2 = XDocument.Load(xmlFilePath);

    var result = from c in xmldoc2.Descendants(XName.
        Get("computer"))
        where Convert.ToInt32(c.Attribute(XName.
            Get("Price")).Value) < 850
        select c;

    foreach (var item in result)
    {
        Console.WriteLine(item);
    }
}
```

Вставьте вызов этого метода в `Main()` и выполните наше приложение. В моем случае результат был таким:

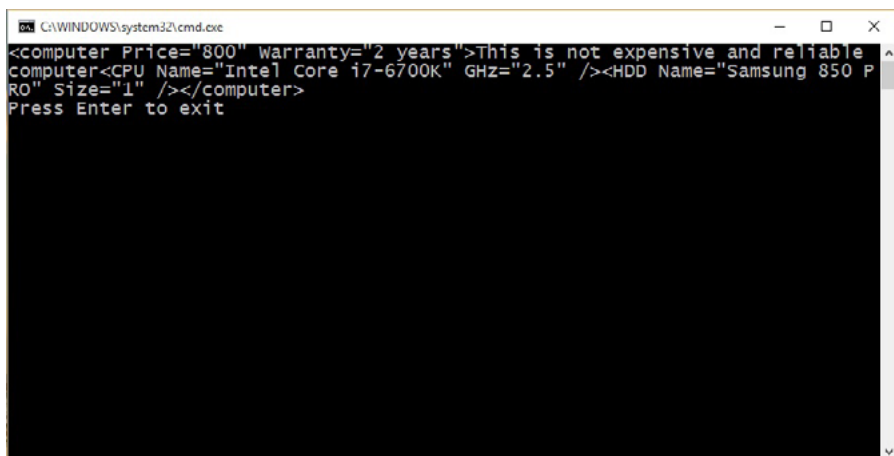


Рис. 13. Выборка из XML файла

Красоты в этом мало, но смысл понятен. Мы получили перечень элементов из нашего XML файла, отвечающих заданному условию — цена меньше 850. Вы должны помнить, что LINQ особенно хорош для работы с очень большими коллекциями, и использовать его для нашего маленького файла стоит только лишь в качестве демонстрации.

Вы видите в этом методе уже привычный вам LINQ запрос в синтаксисе запросов, который отбирает из всех узлов указанного файла, элементы `computer`, со значением атрибута `Price`, меньшим, чем 850. По пути, мы преобразовываем значение атрибута из строки в целое число.

Понятно, что этот же запрос можно переписать и в синтаксисе методов. Он может выглядеть так:

```
var result = xmldoc2.Descendants(XName.Get("computer")).
    Where(c => Convert.ToInt32(c.Attribute(XName.
        Get("Price")).Value) < 850);
```

Обратите внимание на лямбда-выражение в методе `Where()`. Результат выполнения этого запроса будет точно таким же, как и результат выполнения предыдущего запроса. Будет очень полезным упражнением найти какой-либо достаточно большой XML файл, содержащий как можно больше вложенных элементов, и сделать несколько разных выборок из такого файла, используя LINQ To XML.

Теперь вы рассмотрели все современные средства доступа к источникам данных, существующие в .NET для языка C#. Одной из задач, которые вам придется решать в будущем при работе с поставщиками данных, теперь будет выбор той технологии, которая наилучшим образом будет подходить для решения конкретной задачи. А впереди вас ждет знакомство с одним очень эффективным инструментом, который может быть альтернативой всему, с чем вы познакомились до сих пор.

