Урок №5

Содержание

1. Что такое LINQ

LINQ (**Language Integrated Query**) - это специальный язык запросов для **.NET Framework**. Он позволяет выполнять запросы к объектам находящимся в памяти, в типизированной базе данных и в **XML** документе: **LINQ**, **DLINQ XLINQ** соответственно.

Этот язык присутствует в **.NET Framework** начиная с версии 3.5. На данный момент полностью **LINQ** поддерживают только два языка программирования: **C#** и **Visual Basic.NET**.

Особенности **LINQ** позволяют использовать **SQL** подобный синтаксис непосредственно в коде программы, написанной, например, на языке **C#**.

2. Цели и задачи LINQ.

На данный момент наиболее распространен объектно-ориентированый подход к разработке. Это достаточно удобное решение для большинства практических случаев. Однако оно не является панацеей, и далеко не для всех сценариев представляет собой оптимальный выбор. Это хорошо видно на примере баз данных – несмотря на повальное применение **ООП**, для не реляционных хранилищ данный подход так и не сумел завоевать хоть какую-нибудь популярность.

Итак, задача формулируется следующим образом: надо добавить в **ООП**-язык средства работы с коллекциями. Для достижения этой цели было решено перенять положительный опыт реляционных **СУБД**, которые, по сути, и являются набором коллекций. Одной из основных причин успеха современных **РСУБД**, использующих **SQL**, является декларативность языка запросов. Он не требует описывать, «как» достичь результата, а требует лишь указать, «что» надо получить, а «как» – это уже забота **СУБД**. Это позволяет не заботиться о конкретном алгоритме и предоставляет **СУБД** максимум свободы при выборе эффективного решения.

Современные языки программирования до сих пор не предоставляли декларативных средств обработки данных (подобных **SQL**). Однако достаточно давно существуют функциональные языки программирования, в которых обработка данных (и особенно списков) достигла высокого уровня и фактически выглядит декларативной. В **C#** ввели новый декларативный синтаксис и функциональность для работы с коллекциями, но не стали скрывать внутренности механизма реализации этой функциональности, позволяющей данному языку запросов эффективно работать с разными источниками данных. Это сделало решение более расширяемым.

3. Использование LINQ и коллекций.

LINQ позволяет использовать:

- Вывод типов
- Анонимные типы
- Расширяющие методы
- Лямбда-выражения
- Стандартные операторы языка запросов
- Дерево выражений
- Ленивые вычисления

Вывод типов

При объявлении переменных, часто получается громоздкий и избыточный код. Ведь в большинстве случаев переменная имеет такой же тип, что и инициализирующее ее выражение. Стало быть, указание типа переменной только засоряет код и отнимает время. Для облегчения жизни в подобных случаях в **C#** 3.0 введена такая функциональность, как «вывод типов» (**type inference**). Суть довольно проста – если компилятор может вычислить тип из правой части выражения, то декларировать его в левой части не обязательно.

Рассмотрим пример:

```
var i = 1;
var s = "SomeString";
var z = i + i * i;
```

```
var c = 'c';

Console.WriteLine(i.GetType()); // System.Int32
Console.WriteLine(s.GetType()); // System.String
Console.WriteLine(c.GetType()); // System.Char
Console.WriteLine(z.GetType()); // System.Int32
```

Ключевое слово «**var**» вместо типа переменной говорит компилятору о том, что тип выражения надо вычислить из правой части. Обратите внимание – именно компилятору, то есть тип вычисляется не во время выполнения кода, а на этапе компиляции. Язык по-прежнему, статически типизирован, и все ошибки, связанные с неправильным использованием типов, будут отловлены на этапе компиляции.

```
static void Main(string[] args)
        {
            //Определяем массив чисел
            int[] Mas = {1,2,3,4,5,6,7,8,9,10};
             foreach (var val in Mas)
            Console.WriteLine(val);

        }
```

На экране мы увидим все элементы массива.

Теперь немного усложним пример:

```
static void Main(string[] args)
        {
            //Определяем массив чисел
            int[] Mas = {1,2,3,4,5,6,7,8,9,10};
            //Выполняем запрос к массиву Mass и получим его элементы,
которые больше 4, но меньше 8
            var Mas1 = from i in Mas
            where i > 4 && i < 8
            select i;
            foreach (var val in Mas1)
            Console.WriteLine(val);
        }
```

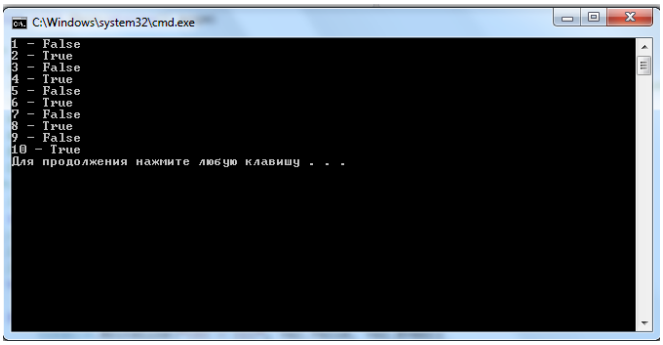
На экране мы увидим числа 5 6 7.

Теперь попробуем выполнить преобразование данных при получении результата. Для этого в разделе **select** запроса **LINQ** необходимо указать, что необходимо сделать с данными.

```
//Определяем массив чисел
            int[] Mas = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            //Выполняем запрос к массиву Mas и выполняем преобразование в
объект
            //содержащий два поля Value - число и
```

```
            //Even значение boolean показывающее является ли число четным
            var Mas1 = from i in Mas
                           select new { Value = i, Even = (i % 2 == 0) };
            foreach (var val in Mas1)

                Console.WriteLine("{0} - {1}", val.Value, val.Even);
```

После выполнения данного куска кода мы увидим на консоли:



Анонимные типы

При полноценной работе с коллекциями тип результата операции или набора операций над коллекциями может сильно отличаться от типов обрабатываемых коллекций. Данный факт порождает следующую проблему: в господствующих статически типизированных языках (таких, как **C++**, **C#**, **Java**) мы обязаны сначала описать тип данных, а потом уже использовать переменные этого типа. Таким образом, при классическом подходе у нас есть два пути.

Во-первых, можно создавать новый тип каждый раз, когда необходимо выполнить преобразование, но есть опасность, что просто не хватит фантазии на имена.

Во-вторых, можно создать один большой тип со всеми полями, которые только могут получиться в результате операций. Но это тоже не выход, так как данный тип будет постоянно проинициализирован лишь частично, что, по сути, означает отказ от статической типизации – если мы попытаемся обратиться не к тому полю, то ошибка всплывет только во время выполнения. Частным случаем такого подхода являются **DataSet**-ы, но их нельзя назвать удачным решением проблемы.

Чтобы выйти из этой ситуации, и ввели такое понятие, как «анонимные типы» (**anonymous type**). 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.

Рассмотрим пример:

```
    static void Main(string[] args)
            {
```

```
            // Вот так выглядит объявление нового типа:
            var anon = new { age = 18, name = "vasa" };

            // дальше можно спокойно использовать
            Console.WriteLine("age = " + anon.age + "\tname = " +
anon.name);

            // а так можно посмотреть, что же создалось:
            Console.WriteLine(anon.GetType());

        }
```

У созданного подобным образом типа нет имени, потому он, собственно, и анонимный, и различается только по сигнатуре входящих в него полей, что хорошо видно при попытке означенное имя запросить.

Как легко можно заметить, здесь также не обошлось без функциональности, описанной в предыдущем разделе. Причем вывод типов помог здесь два раза. Во-первых, при объявлении переменной. Без этого нам пришлось бы описывать тип дважды, и все удовольствие от анонимности типа нивелировалось бы утомительностью его объявления. А во-вторых, при описании полей их тип нигде не указан и вычисляется из типа инициализирующего выражения.

У анонимного типа есть один существенный недостаток, серьезно ограничивающий возможности его использования. Его нельзя экспортировать за пределы метода, в котором его создали.

Рассмотрим пример с применением **LINQ** к коллекциям:

```
class Program
    {
        class Tovar
        {//Новая возможность C#3.0 автоматические свойства
            public string Name { set; get; }
            public float Price { set; get; }
        }
        static void Main(string[] args)
        {
            //создаем список
            var Mas = new List<Tovar>();
            //заполняем список тремя позициями
            Mas.Add( new Tovar() { Name = "картошка", Price = 2.04f });
            Mas.Add(new Tovar() { Name = "огурцы", Price = 7.20f });
            Mas.Add(new Tovar() { Name = "помидоры", Price = 11.30f });
            //Выбираем все товары с ценой от 7 до 20
            var Tovars = from t in Mas
                         where t.Price>=7 && t.Price<=20
                         select t;
            foreach (var t in Tovars)
                Console.WriteLine("{0} - {1}", t.Name, t.Price);
```

```
        }

    }
```

Как видим, работа с коллекциями не отличается от работы с массивами, но необходимо помнить, что **LINQ** умеет работать только с типизированными коллекциями (**Generic**).

Расширяющие методы

Расширяющие методы (**Extension Methods**) - это возможность добавлять новые методы без изменения класса объекта.

Например, используя расширяющие методы, можно в класс **Object** добавить новый метод:

```
class Program
    {
        static void Main()
        {
            string myString = "Hi!";
            myString.PrintMe(2); //Используем Extension Method
        }
    }

    // Класс в котором объявлены Extension Methods должен быть static
     public static class MyClass
    {
        // Определяем Extension Method PrintMe
        public static void PrintMe(this object obj, int count)
        {
            for (int i = 0; i < count; i++)
            {
                Console.WriteLine(obj.ToString());
            }
        }

    }
```

Расширяющие методы объявляются в **static** классах, и естественно сами должны быть **static**. При объявлении данного метода, в передаваемых параметрах, необходимо использовать ключевое слово **this**, после него идет тип, для которого создается расширяющий метод.

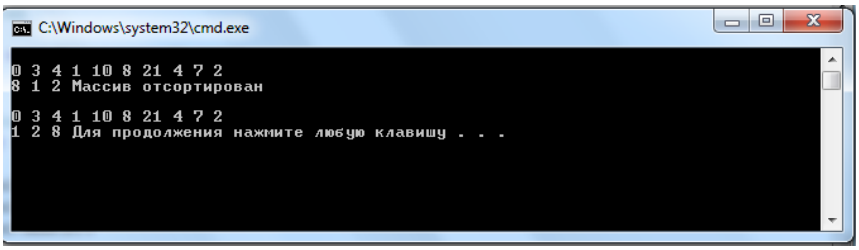
Работает этот механизм следующим образом. Увидев вызов метода объекта, компилятор сначала проверяет, есть ли необходимый метод у класса этого объекта, и если он отсутствует, пытается найти расширяющий метод. При компиляции расширяющего метода компилятор помечает этот метод специальным атрибутом. В дальнейшем, просматривая список открытых пространств имен, компилятор ищет статические классы, и, если находит, анализирует их на наличие расширяющих

методов.

Рассмотрим пример:

```
static class PrintArrays
    {
        public static void Print(this IEnumerable<int> mas)
        {
            Console.WriteLine();
            foreach (var i in mas)
            {
                Console.Write("{0} ",i);
            }
        }

        public static void Sort(this IEnumerable<int> mas)
        {
            var mas1 = from t in mas
                       orderby t
                       select t;
            mas = mas1;

            Console.WriteLine("Массив отсортирован");

        }
    }

    class Program
    {
        static void Main(string[] args)
        {

            IEnumerable<int> mas1 = new int[10] { 0, 3, 4, 1, 10, 8, 21,
4, 7, 2 };
            var mas2 = new List<int>();

            mas2.Add(8);
            mas2.Add(1);
            mas2.Add(2);

            mas1.Print();
            mas2.Print();
            mas1.Sort(); //вызывается расширяющий метод
            mas2.Sort(); //вызывается стандартный метод сортировки List
            mas1.Print();
            mas2.Print();

        }


    }
```

Выполняя этот пример, получаем следующий результат:



Введением расширяющих методов удалось достичь следующего: во-первых, мы избавились от необходимости явно писать имя класса, содержащего расширяющую функцию; во-вторых, подобный синтаксис гораздо понятнее, если придется записывать несколько операций подряд; в-третьих, **Visual Studio** сама подскажет о наличии подобных методов расширения.

При этом добавление **this** не отменяет старого способа обращения к утилитному методу, его вполне можно вызвать как раньше:

```
PrintArrays.Print(mas1);
```

Лямбда-выражения

Лямбда-выражение (**Lambda Expression**) — это анонимная функция, которая содержит выражения и операторы и может использоваться для создания делегатов или типов дерева выражений.

Во всех лямбда-выражениях используется лямбда-оператор =>, который читается как "переходит в". Левая часть лямбда-оператора определяет параметры ввода (если таковые имеются), а правая часть содержит выражение или блок оператора. Лямбда-выражение x => x * x читается как "x переходит в x x раз". Это выражение может быть назначено типу делегата следующим образом:

```
class Program
    {
        delegate int del(int i);
        static void Main(string[] args)
        {
            del myDelegate = x => x * x;
            int i = myDelegate(5);
            Console.WriteLine(i);
        }

    }
```

Стандартные операторы запросов

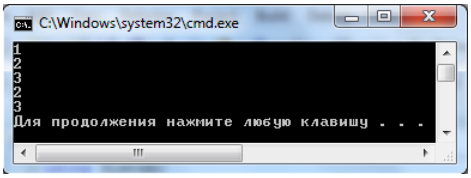
Стандартные операторы запросов (**Standard Query Operators**) - это расширенные методы (**Extension methods**) которые находятся в классе **System.Linq.Enumerable**. Они расширяют функционал объектов, реализующих интерфейс **IEnumerable**.

Оператор Where

Оператор **Where** возвращает все элементы массива, удовлетворяющие определенному условию.

```
static void Main(string[] args)
        {
            //Определяем массив из 10 чисел
            int[] Mas = { 1, 2, 3, 4, 5, 2, 3, 8, 9, 10 };
            //Отбираем элементы, удовлетворяющие условию
            var Mas1 = Mas.Where(i=>i<=3);
            foreach (var num in Mas1)
            {
                Console.WriteLine(num);
            }

        }
```

Результат будет такой:



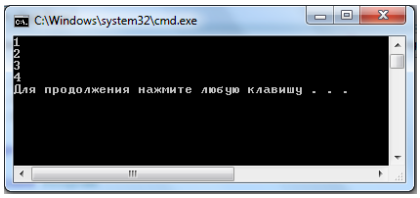
Оператор Take

Оператор **Take** возвращает указанное количество элементов массива, начиная с первого.

Например:

```
static void Main(string[] args)
        {
            //Определяем массив из 10 чисел
            int[] Mas = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            //Выбераем из массива первых 4 элемента
            var Mas1 = Mas.Take(4); //Выбераем из массива первых 4
элемента
            foreach (var n in Mas1)
            {
                Console.WriteLine(n);
            }

        }
```

Результат будет такой:

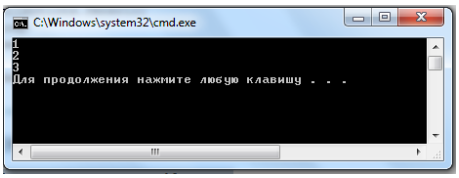


Оператор TakeWhile

Оператор **TakeWhile** возвращает все элементы массива, удовлетворяющие определенному условию.

```
static void Main(string[] args)
        {
            //Определяем массив из 10 чисел
            int[] Mas = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            //Выбераем из массива первых 4 элемента
            var Mas1 = Mas.TakeWhile(i => i <= 3);
            foreach (var n in Mas1)
            {
                Console.WriteLine(n);
            }

        }
```

Результат будет такой:

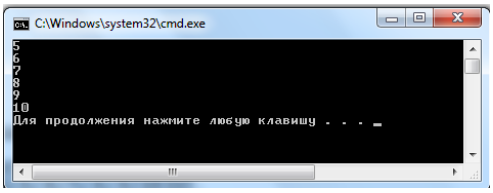


Оператор Skip

Оператор **Skip**, наоборот выдает все элементы массива, пропуская указанное количество начальных элементов.

```
static void Main(string[] args)
        {
            //Определяем массив из 10 чисел
            int[] Mas = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
            var Mas1 = Mas.Skip(4); //Пропускаем первые 4 элемента
            foreach (var n in Mas1)
            {
                Console.WriteLine(n);
            }

        }
```

Результат будет такой:

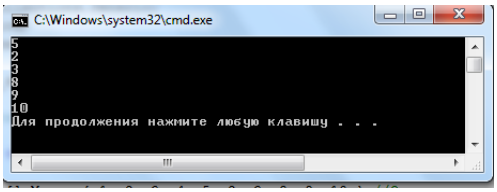


Оператор SkipWhile

Оператор **SkipWhile** выбирает все элементы массива, пропускаю все начальные элементы, до тех пор, пока не будет найден первый элемент, для которого условие перестало выполняться.

```
static void Main(string[] args)
        {
           //Определяем массив из 10 чисел
            int[] Mas = { 1, 2, 3, 4, 5, 2, 3, 8, 9, 10 };
           //Пропускаем элементы, пока выполняется условие
            var Mas1 = Mas.SkipWhile(i => i <= 4);
            foreach (var num in Mas1)
            {
                Console.WriteLine(num);
            }
        }
```

Результат будет такой:

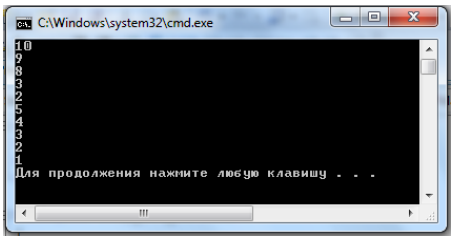


Оператор Reverse

Оператор **Reverse**, переворачивает исходный массив.

```
static void Main(string[] args)
        {
            //Определяем массив из 10 чисел
            int[] Mas = { 1, 2, 3, 4, 5, 2, 3, 8, 9, 10 };
            //Переворачиваем массив
            var Mas1 = Mas.Reverse();
            foreach (var num in Mas1)
            {
                Console.WriteLine(num);
            }
```

Результат будет такой:



Дерево выражений (Expression Tree)

Описанные ранее лямбда-выражения позволяют сказать компилятору, что именно мы хотим сделать, а уже компилятор сам определяет, как это сделать. Такой алгоритм идеален для коллекций, но для других источников данных такой механизм не подходит. Это связано с тем что текст функции храниться в откомпилированном виде, а нам необходимо разобрать любым другим интерпретатором указанное лямбда-выражение и реализовать запрос к определенным данным. ... Выход придумали следующий – ввели специальный тип **Expression<T>**, экземпляру которого можно присвоить лямбда-выражение. При компиляции тело лямбды не компилируется, а сохраняется в виде данных, представляющих собой Абстрактное Синтаксическое Дерево – **AST (Abstract Syntactic Tree)** – вышеупомянутой лямбды.

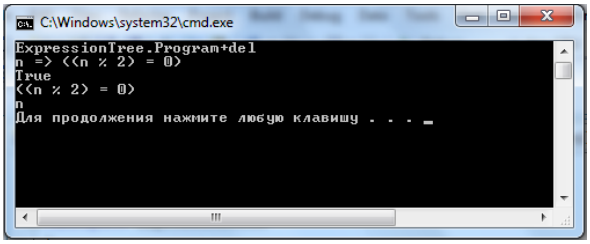
Тип **Expression<T>** лежит в пространстве имен **System.Linq.Expressions**.

Рассмотрим пример:

```
using System;
using System.Linq.Expressions;

namespace ExpressionTree
{
    class Program
    {
        delegate bool del(int i);
        static void Main(string[] args)
        {
            // Обычная лямбда
            del isEven = n => (n % 2 == 0);
            Console.WriteLine(isEven); // ExpressionTree.Program+del

            // Expression
            Expression<del> exIsEven = n => n % 2 == 0;
            Console.WriteLine(exIsEven); // n => ((n % 2) = 0)

            Console.WriteLine(isEven(2));   // true
           // Console.WriteLine(exIsEven(2)); // Compilation error
           // тело лямбды
            Console.WriteLine(exIsEven.Body); // ((n % 2)=0

            // имя параметра
            Console.WriteLine(exIsEven.Parameters[0]);
```

```
            }
        }
}
```

В результате выполнения примера, получим следующее:



Именно по такому принципу и работают различные реализации **LINQ**, например, **LINQ 2 SQL** и **LINQ 2 XML**.

Для удобства работы с **AST Expression** предоставляет довольно богатый **API** – набор методов, с помощью которых переданный участок кода можно как разобрать, так и «собрать», и даже скомпилировать для последующего выполнения. Выглядит это примерно так:

```
Console.WriteLine(exIsEven.Equals(5));
```

Новое выражение можно построить и динамически:

```
using System;
using System.Linq.Expressions;

namespace DinamicExpressionTree
{
    class Program
    {
        static void Main(string[] args)
        {
            // создать параметр лямбды
            var parameterN = Expression.Parameter(typeof(int), "n");

            //  собственно построение выражения
            /* Предика - это функция, принимающая один параметр
             и возвращающая логическое выражение*/
            var expression = Expression.Lambda<Predicate<int>>(
              Expression.Equal(
                Expression.Modulo(
                parameterN,
                Expression.Constant(2)),
                Expression.Constant(0)
              ),
              parameterN);

            // выводим, что получилось
```

```
            Console.WriteLine(expression); // n => ((n % 2) =0)

            // теперь это можно скомпилировать
            Predicate<int> compiledExpression = expression.Compile();

            // и использовать
            Console.WriteLine();
            Console.WriteLine(" 2 is Even ? " + compiledExpression(2));
// true
            Console.WriteLine(" 3 is Even ? " + compiledExpression(3));
// false

        }
    }

}
```

Ленивые вычисления (Lazy Evaluation)

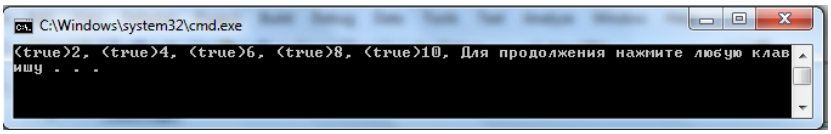
Суть явления под названием «ленивые вычисления» заключается в том, что вычисления откладываются до тех пор, пока не понадобится их результат. Такой подход позволяет гарантировать, что вычисляться будут только те данные, которые требуются для получения конечного результата, и тем самым позволяет описывать только зависимости функций друг от друга, и не следить за тем, чтобы не осуществлялось «лишних вычислений».

Рассмотрим пример:

```
namespace LazyEvaluation
{
    // создадим класс с расширяющией функцией
    public static class Help
    {
        public static IEnumerable<T> LazyFilter<T>(
          this IEnumerable<T> enumerable,
          Predicate<T> predicate)
        {
            foreach (var e in enumerable)
                if (predicate(e))
                {
                    Console.Write("(true)");
                    yield return e;
                }
        }
    }
    class Program
    {
       static void Main(string[] args)
        {
         // создаем массив
        IEnumerable<int> array = new [] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
            //задаем фильтр
        var array1 = array.LazyFilter(n => n % 2 == 0);
        //выводим содержимое массива
        foreach (var elem in array1)
          Console.Write(elem + ", ");

        }

    }
```

Результат будет такой:



То есть, реальное обращение к элементам коллекции **array** произошло не в тот момент, когда была применена фильтрация и объявлена переменная «**array1**», а когда нам понадобился результат этой фильтрации. Причина такого поведения в реализации утилитного метода **LazyFilter<T>**. Если бы он был описан не через **yield return**, а прямым перебором, то фильтрация произошла бы в момент использования.

Все стандартные расширяющие методы из библиотек **LINQ**, реализованы именно по принципу ленивых вычислений. Сделано это для того, чтобы была возможность применять несколько последовательных операций в разных выражениях, прежде чем реально будет выполнен запрос.

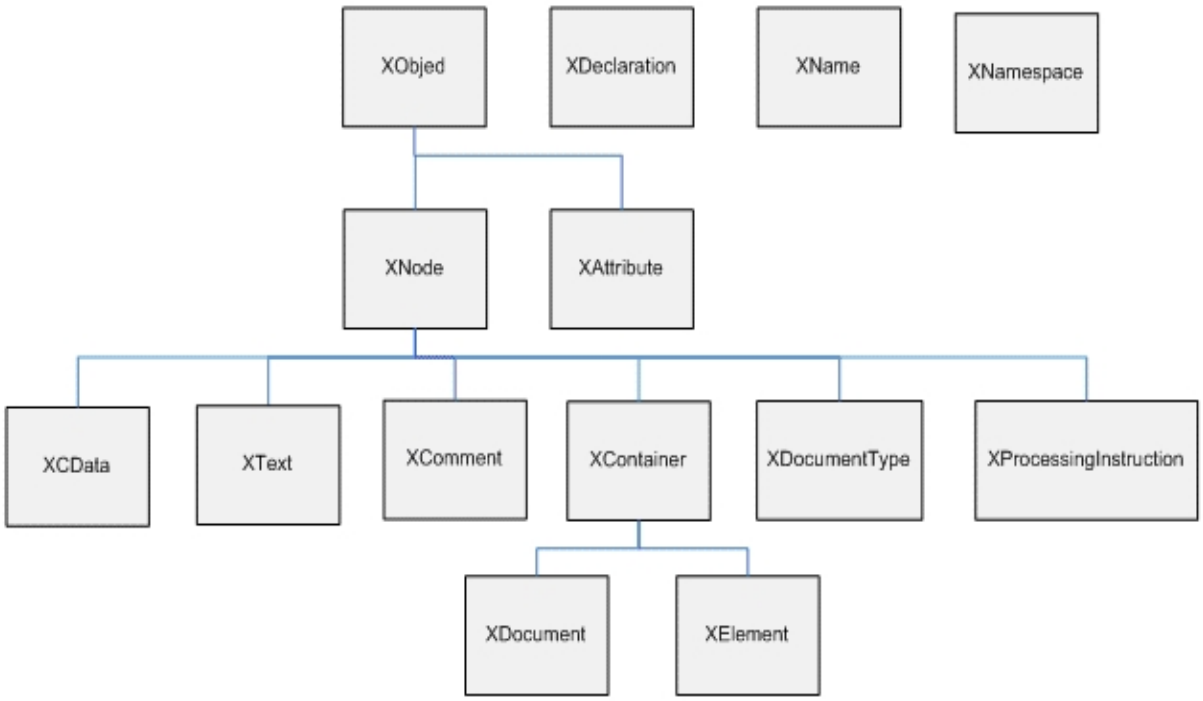
4. LINQ и XML

Обзор классов пространства имен system.xml.linq

Класс	Описание
XObject	Описывает узел или атрибут в **XML** дереве.
XDeclaration	Представляет **XML** описание.
XName	Описывает имя узла или атрибута в **XML** дереве.
XNamespace	Описывает пространство имен.
XNode	Представляет собой абстрактный узел в **XML** дереве (элемент, комментарий, текст, управляющая инструкция).
XAttribute	Описывает **XML** атрибут.
XCData	Представляет собой текстовый узел (тип данных **CDATA**).
	Описывает текстовый узел.

XText	
XComment	Описывает **XML** комментарий.
XContrainer	Описывает **XML** узел, содержащий внутри другие элементы.
XDocumentType	Описывает **XML Document Type Definition (DTD).**
XProcessingInstruction	Описывает **XML** инструкции.
XDocument	Описывает **XML** документ.
XElement	Описывает **XML** элемент.

Ниже представлена иерархия классов пространства имен **system.xml.linq**



Класс XDocument

Основные методы класса **XDocument**:

Метод	**Описание**
Add	Добавляет указанное содержимое как дочерние элементы текущего

	XContainer.
AddAfterSelf	Добавляет указанное содержимое непосредственно после данного узла.
AddBeforeSelf	Добавляет указанное содержимое непосредственно перед данным узлом.
AddFirst	Добавляет заданное содержимое как первые дочерние элементы данного документа или элемента.
Ancestors	Возвращает всю или отфильтрованную коллекцию элементов предков данного узла.
CreateReader	Создает **XmlReader** для данного узла.
CreateWriter	Создает **XmlWriter,** который можно использовать для добавления узлов в **XContainer.**
DescendantNodes	Возвращает коллекцию подчиненных узлов для документа или элемента в порядке следования узлов документа.
Descendants	Возвращает всю или отфильтрованную коллекцию подчиненных узлов для данного документа или элемента в порядке следования узлов документа.
Element	Получает первый (в порядке следования узлов документа) дочерний элемент с заданным **XName.**
Elements	Возвращает всю или отфильтрованную коллекцию дочерних элементов для данного документа или элемента в порядке следования узлов документа.
ElementsAfterSelf	Возвращает всю или отфильтрованную коллекцию родственных элементов после данного узла в документном порядке.
ElementsBeforeSelf	Возвращает всю или отфильтрованную коллекцию родственных элементов перед данным узлом в документном порядке.
IsAfter	Определяет, появляется ли текущий узел после указанного узла при документном порядке.
IsBefore	Определяет, появляется ли текущий узел перед указанным узлом при документном порядке.
Load	Создает новый **XDocument** из файла, указанного с помощью **URI,** из **TextReader** или из **XmlReader.** Функция

	статическая
Nodes	Возвращает коллекцию дочерних узлов для данного документа или элемента в порядке следования узлов документа
NodesAfterSelf	Возвращает коллекцию родственных узлов после данного узла в документном порядке.
NodesBeforeSelf	Возвращает коллекцию родственных узлов перед данным узлом в документном порядке
Parse	Создает новый **XDocument** из строки, при необходимости оставляя пустое пространство, задавая базовый URI и сохраняя сведения о строках.
Remove	Удаляет данный узел из родительского объекта.
RemoveNodes	Удаляет дочерние элементы из данного документа или элемента.
Save	Сериализует данный **XDocument** в файл, используя TextWriter или XmlWriter.
WriteTo	Записывает данный документ в **XmlWriter**.

Основные свойства класса **XDocument**:

Свойство	**Описание**
BaseUri	Получение базового **URI** для данного **XObject**.
Declaration	Возвращает или задает объявление **XML** для этого документа.
Document	Получение **XDocument** для данного **XObject**.
DocumentType	Получает определение типа документов (**DTD**) для этого документа.
FirstNode	Получает первый дочерний узел от данного узла.
LastNode	Получает последний дочерний узел от данного узла.
NextNode	Получает следующий родственный узел данного узла.
NodeType	Получает тип данного узла.
Parent	Получение родительского **XElement** данного элемента.

PreviousNode	Получает предыдущий родственный узел данного узла.
Root	Получает корневой элемент дерева **XML** для этого документа.

Класс XNode

Основные методы класса **XNode**:

Метод	**Описание**
AddAfterSelf	Добавляет указанное содержимое непосредственно после данного узла.
AddBeforeSelf	Добавляет указанное содержимое непосредственно перед данным узлом.
Ancestors	Возвращает всю или отфильтрованную коллекцию элементов предков данного узла.
CreateReader	Создает **XmlReader** для данного узла.
CompareDocumentOrder	Сравнивает два узла с целью определения их относительного порядка документа **XML.**
DeepEquals	Сравнивает значения двух узлов, включая значения всех подчиненных узлов.
ElementsAfterSelf	Возвращает всю или отфильтрованную коллекцию родственных элементов после данного узла в документном порядке.
ElementsBeforeSelf	Возвращает всю или отфильтрованную коллекцию родственных элементов перед данным узлом в документном порядке.
IsAfter	Определяет, появляется ли текущий узел после указанного узла при документном порядке.
IsBefore	Определяет, появляется ли текущий узел перед указанным узлом при документном порядке.
NodesAfterSelf	Возвращает коллекцию родственных узлов после данного узла в документном порядке.
NodesBeforeSelf	Возвращает коллекцию родственных узлов перед данным узлом в документном порядке
ReadFrom	Создает **XNode** из **XmlReader.**

Remove	Удаляет данный узел из родительского объекта.
ReplaceWith	Заменяет данный узел на указанное содержимое.
WriteTo	Записывает данный узел в **XmlWriter.**

Основные свойства класса **XNode**:

Свойство	**Описание**
BaseUri	Получение базового **URI** для данного **XObject.**
Document	Получение **XDocument** для данного **XObject.**
DocumentOrderComparer	Получает интерфейс **IComparer,** который может сравнить относительную позицию двух узлов.
EqualityComparer	Получает интерфейс **IComparer,** который сравнивает два узла для проверки равенства значений.
NextNode	Получает следующий родственный узел данного узла.
NodeType	Получает тип данного узла.
Parent	Получение родительского **XElement** данного элемента.
PreviousNode	Получает предыдущий родственный узел данного узла.

Класс XAttribute

Основные методы класса **XAttribute**:

Метод	**Описание**
Remove	Удаляет данный атрибут из родительского элемента.
SetValue	Задать значение атрибуту.

Основные свойства класса **XAttribute**:

Свойство	**Описание**
BaseUri	Получение базового **URI** для данного **XObject.**
Document	Получение **XDocument** для данного

	XObject.
EmptySequence	Получение пустой коллекции атрибутов. Функция статическая.
IsNamespaceDeclaration	Определяет, является ли этот атрибут объявлением пространства имен.
Name	Получение развернутого имени этого атрибута.
NextAttribute	Получение следующего атрибута родительского элемента.
NodeType	Получает тип данного узла.
Parent	Получение родительского **XElement** данного элемента.
PreviousAttribute	Получение предыдущего атрибута родительского элемента.
Value	Возвращает или задает значение этого атрибута.

Примеры

Для начала давайте рассмотрим пример, который создает XML файл во время выполнения программы.

```
using System;
using System.Linq;
using System.Xml.Linq;
using System.Xml;

namespace GenerateXML
{
    class Program
    {
        static void Main(string[] args)
        {
            XElement node = new XElement ( "Контакты",
            new XElement ( "Контакт",
            new XElement ( "Имя", "Патрик Хайнс"),
            new XElement ( "Телефон", "206-555-0144"),
            new XElement ( "Адрес",
            new XElement ( "Улица", "123 Main St"),
            new XElement ( "Город", "Mercer Island"),
            new XElement ( "Государство", "WA"),
            new XElement ( "Индекс", "68042") ) ) );
            XmlWriter w = XmlWriter.Create("1.xml");
            node.WriteTo(w);
            w.Close();
            Console.WriteLine("файл 1.xml создан");
```

```
            }
        }

}
```

В результате выполнения этого примера будет сформирован следующий XML файл:

```
<?xml version="1.0" encoding="utf-8"?>
<Контакты>
  <Контакт>
    <Имя>Патрик Хайнс</Имя>
    <Телефон>206-555-0144</Телефон>
    <Адрес>
      <Улица>123 Main St</Улица>
      <Город>Mercer Island</Город>
      <Государство>WA</Государство>
      <Индекс>68042</Индекс>
    </Адрес>
 </Контакт>
</Контакты>
```

Теперь давайте рассмотрим пример, который вычитывает данные из XML документа и делает выборку из них.

Для начала сформируем XML документ

```
<?xml version = "1.0"?>
<library>
 <disk>
  <name>Albom1</name>
  <siner>Beledi</siner>
  <year>2006</year>
  <studia>
      <namef>Artek</namef>
      <address>Kiev</address>
      <telefon>7450789</telefon>
  </studia>
 </disk>

 <disk>
  <name>Albom3</name>
  <siner>Barabani</siner>
  <year>2005</year>
  <studia>
      <namef>Artek</namef>
```

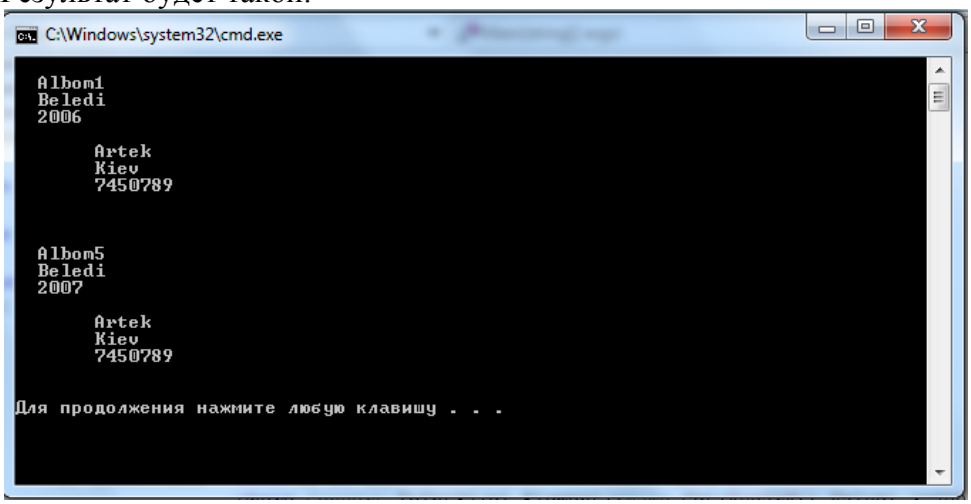
```
    <address>Kiev</address>
    <telefon>7450789</telefon>
  </studia>
 </disk>

<disk>
  <name>Albom5</name>
  <siner>Beledi</siner>
  <year>2007</year>
  <studia>
    <namef>Artek</namef>
    <address>Kiev</address>
    <telefon>7450789</telefon>
  </studia>
 </disk>

</library>
```

Теперь напишем программу, которая отобразит все альбомы выпущенные позднее 2006 года

```
using System;
using System.Linq;
using System.Xml.Linq;
using System.Xml;

namespace SelectFromXml
{
    class Program
    {
        static void Main(string[] args)
        {
            //создаем reader
            XmlReader reader = XmlReader.Create("albom.xml");
            //загружаем данные их документа
            XDocument doc = XDocument.Load(reader);

            //Формируем выборку
            var collection = from el in
doc.Descendants(XName.Get("disk"))
                             where
Convert.ToInt32(el.Element(XName.Get("year")).Value) >=2006
                             select el;

            //Выводим полученный результат
            foreach (var el in collection)
            {
                Console.WriteLine(el.Value);
            }
        }
```

```
            }
        }
```

Результат будет такой:



5. Использование LINQ и SQL

Для работы с LINQ в SQL в проект нужно добавить ссылки на две сборки: System.Data.Linq и Microsoft.SqlServer.Types.

Первый шаг в использовании LINQ to SQL — это создание классов-отображений для таблиц базы данных. Для одной таблицы – один класс-отображение.

Рассмотрим пример:

Создадим базу данных Countries и создадим таблицу Country.

```
create table country
(
  id int not null primary key identity,
  name nvarchar(40),
  square float
  )

  insert into country  values('Украина',603.7);
  insert into country  values('Россия',17075.4);
  insert into country  values('Польша',312.7);
```

Разработаем класс

```
using System.Data.Linq;
using Microsoft.SqlServer.Types;
using System.Data.Linq.Mapping;
```

```
namespace TableToClass
{
    [Table()]
    public sealed class Country
    {
     [Column(AutoSync = AutoSync.OnInsert, DbType = "int",
         IsPrimaryKey = true, IsDbGenerated = true, UpdateCheck =
UpdateCheck.Never)]
      public int Id;
    [Column(DbType = "nvarchar(40)", CanBeNull = true, Name="Name")]
       public string CountryName;
          [Column(DbType = "float",  CanBeNull = true)]
          public float Square;
       }
}
```

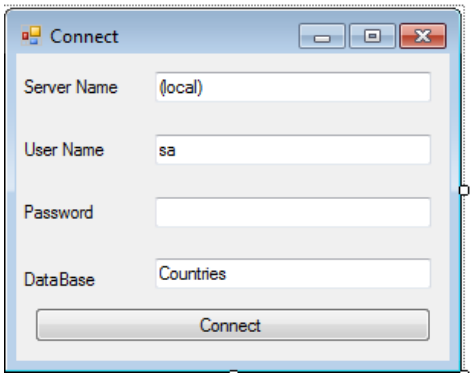
Над объявлением класса и над полями расставлены атрибуты. Например, атрибут **Table** указывает, что этот класс ассоциирован с таблицей в базе данных. Если имя класса совпадает с именем таблицы, то атрибут можно записывать без параметров, а если нет, то нужно будет указать дополнительное свойство **Name: [Table(Name = «Country»)].** То же самое справедливо и для полей класса, если его имя совпадает с название столбца, то параметр **Name** в атрибуте можно опустить.

Ещё один класс нужен для создания контекста базы данных. Мы можем использовать уже имеющийся **DataContext**, но лучше сделать своего наследника для строгой типизации.

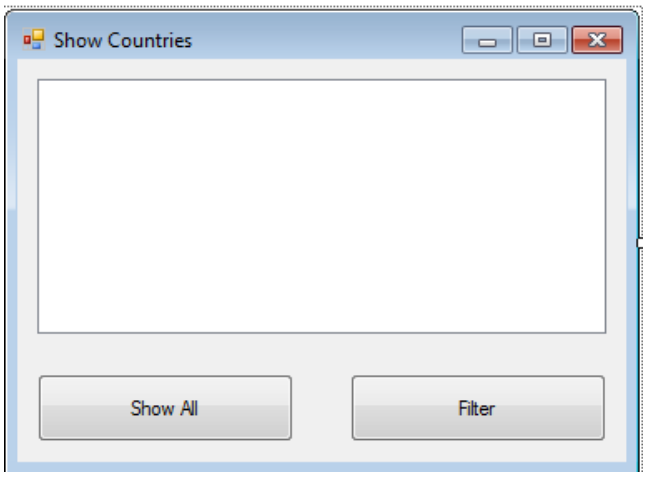
```
public class ExampleDatabase: DataContext
       {
          public Table<Country> Country;
          public ExampleDatabase(string connectionString)
           : base(connectionString)
          {

          }


       }
```

Теперь отобразим все страны из нашей таблицы, которые содержат букву «о»:

Для этого добавим в проект две формы:



```
public partial class Form2 : Form
    {
        public static string StrConnect;
        public Form2()
        {
            InitializeComponent();
        }

        private void Connect_Click(object sender, EventArgs e)
        {
            StrConnect = "Data Source=" + DataBase.Text +
                ";User Id=" + User.Text +
                ";Password=" + Password.Text +
                ";Initial Catalog=" + DataBase.Text;
            this.DialogResult = DialogResult.OK;
        }

    }
```

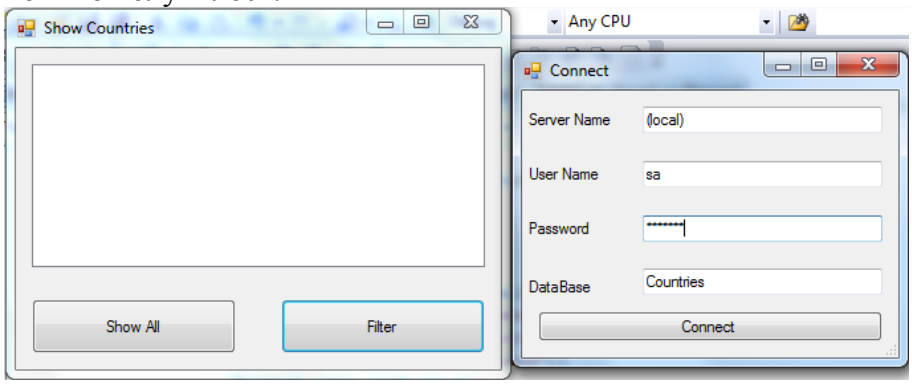


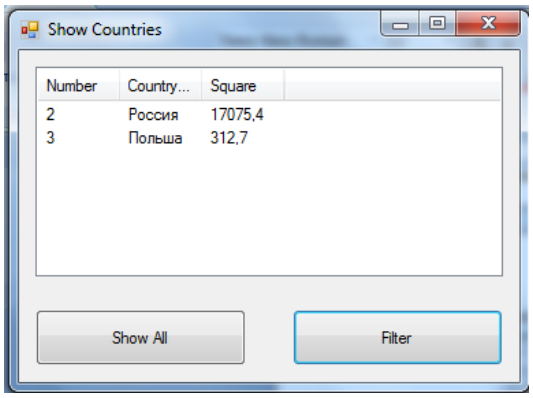

```
public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

        }
        private void Filter_Click(object sender, EventArgs e)
        {
            try
            {
                Form2 f = new Form2();
                if (f.ShowDialog() == DialogResult.OK)
                {
                    ExampleDatabase db = new
ExampleDatabase(Form2.StrConnect);

                    var q = from item in db.Country
                            where item.CountryName.Contains("o")
                            select item;
```

```
                    listView1.Clear();

                    listView1.View = View.Details;
                    listView1.Columns.Add("Number");
                    listView1.Columns.Add("Country_Name");
                    listView1.Columns.Add("Square");
                    foreach (var item in q)
                    {
                        ListViewItem it =
listView1.Items.Add(item.Id.ToString());
                        it.SubItems.Add(item.CountryName);
                        it.SubItems.Add(item.Square.ToString());

                    }
                }
            }
            catch (System.Exception e1)
            {
                MessageBox.Show(e1.Message);
            }

        }

        private void ShowAll_Click(object sender, EventArgs e)
        {
            try
            {
                Form2 f = new Form2();
                if (f.ShowDialog() == DialogResult.OK)
                {
                    ExampleDatabase db = new
ExampleDatabase(Form2.StrConnect);

                    var q = from item in db.Country
                            select item;

                    listView1.Clear();

                    listView1.View = View.Details;
                    listView1.Columns.Add("Number");
                    listView1.Columns.Add("Country_Name");
                    listView1.Columns.Add("Square");
                    foreach (var item in q)
                    {
                        ListViewItem it =
listView1.Items.Add(item.Id.ToString());
                        it.SubItems.Add(item.CountryName);
                        it.SubItems.Add(item.Square.ToString());

                    }
                }
            }
            catch (System.Exception e1)
            {
                MessageBox.Show(e1.Message);
            }

        }
```

```
        }
```

Вот что получилось:

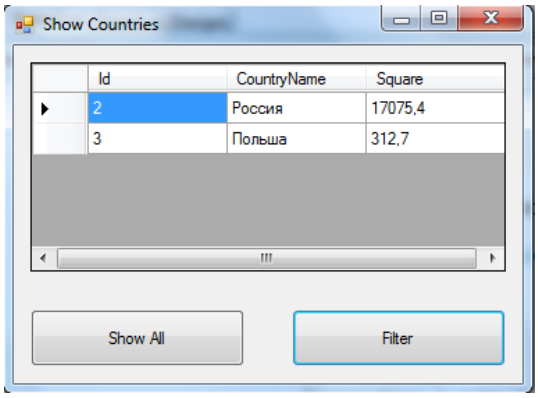




Теперь немного модифицируем пример, заменим у первой формы (**Show Countries**) элемент управления **ListView** на **DataGridView**. И внесем соответственно изменения в обработчики кнопок **Filter** и **Show All**.

```
private void Filter_Click(object sender, EventArgs e)
        {
            try
            {
                Form2 f = new Form2();
                if (f.ShowDialog() == DialogResult.OK)
                {
                    ExampleDatabase db = new
ExampleDatabase(Form2.StrConnect);

                    var q = from item in db.Country
                            where item.CountryName.Contains("o")
                            select new
                            {
                                Id = item.Id,
                                CountryName = item.CountryName,
                                Square = item.Square
                            };

                    dataGridView1.DataSource = q.ToList();
```

```
                }
            }
            catch (System.Exception e1)
            {
                MessageBox.Show(e1.Message);
            }

        }

        private void ShowAll_Click(object sender, EventArgs e)
        {
            try
            {
                Form2 f = new Form2();
                if (f.ShowDialog() == DialogResult.OK)
                {
                    ExampleDatabase db = new
ExampleDatabase(Form2.StrConnect);

                    var q = from item in db.Country
                            select new
                            {
                                Id = item.Id,
                                CountryName = item.CountryName,
                                Square = item.Square
                            };

                    dataGridView1.DataSource = q.ToList();


                }
            }
            catch (System.Exception e1)
            {
                MessageBox.Show(e1.Message);
            }


        }
```



6. Домашнее задание

1. Разработайте таблицу расписание, с полями (ФИО сотрудника, начало периода работы (дата, например 1.01.2009), конец периода (дата, например 30.04.2009)) и таблицу дни работы, содержащую (номер дня недели, ид_рассписания). Необходимо написать программу, которая по фамилии сотрудника отобразит список дат, в которые он работал.
2. Разработайте XML файл, содержащий описание драгоценных камней (название, цвет, признак прозрачности (да/нет), тип (поделочный, драгоценный, полудрагоценный), описание) . Напишите программу, которая будет отображать список камней по выбранному цвету.
3. Разработайте класс телефонный справочник. Для хранения экземпляров класса воспользуйтесь любой коллекцией. Напишите программу, которая вычитывает данные из файла, сохраняет их, а также позволяет просматривать записи, сортировать их по имени или номеру телефона, редактировать записи, добавлять и удалять их. Для сохранения данных в файл воспользуйтесь сериализацией.