Урок 4

Объектно - Ориентированное Программирование

Списки. Обобщенные списки. Лямбда-выражения. Linq

Generic. Collection. Необобщенные коллекции

Необобщенные коллекции вошли в состав среды .NET Framework ещё в версии 1.0. Они определяются в пространстве имён System.Collections. Необобщенные коллекции представляют собой структуры данных общего назначения, оперирующие ссылками на объекты. Таким образом, они позволяют манипулировать объектом любого типа, хотя и нетипизированным способом. В этом состоит их преимущество и в то же время недостаток. Благодаря тому, что необобщенные коллекции оперируют ссылками на объекты, в них можно хранить разнотипные данные. Это удобно в тех случаях, когда требуется манипулировать совокупностью разнотипных объектов, или же когда типы хранящихся в коллекции объектов заранее неизвестны. Но, если коллекция предназначается для хранения объекта конкретного типа, то необобщенные коллекции не обеспечивают типовую безопасность, которую можно обнаружить в обобщенных коллекциях.

Пример:

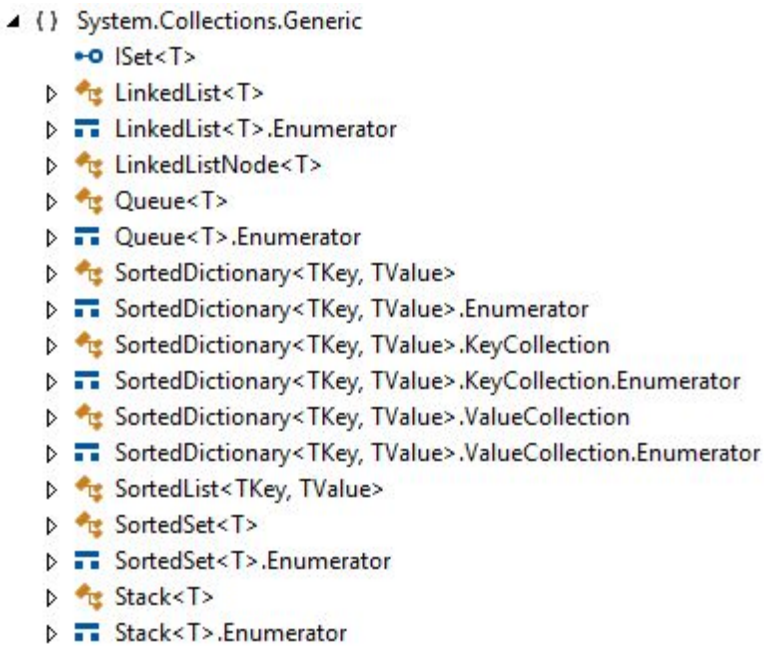
```
using System;
using System.Collections;
class Program
{
    static void Main(string[] args)
    {
        ArrayList list = new ArrayList();
        list.Add(1);
        list.Add(3.14);
        list.Add("Строка");
        list.Add(new int[] { 1,2,3});
        foreach (object element in list)
            Console.WriteLine(element);
        Console.ReadKey();
    }
}
```

Вывод программы:

```
1
3,14
Строка
System.Int32[]
```

Обобщенные коллекции

Обобщенные объекты .Net Framework легко отличить по угловым скобкам после их названия.



В таблице описаны основные обобщенные интерфейсы, с которыми придётся иметь дело при работе с обобщенными классами коллекций

Интерфейс System. Collections.Generic	Назначение
ICollection<T>	Определяет общие характеристики (например, размер, перечисление и безопасность к потокам) для всех типов обобщенных коллекций.
IComparer<T>	Определяет способ сравнения объектов.
IDictionary<TKey, TValue>	Позволяет объекту обобщенной коллекции представлять своё содержимое посредством пар “ключ/значение”.
IEnumerable<T>	Возвращает интерфейс IEnumerator<T> для заданного объекта.
IEnumerator<T>	Позволяет выполнять итерацию в стиле foreach по элементам коллекции.
IList<T>	Обеспечивает поведение добавления, удаления и индексации элементов в последовательном списке объектов.
ISet<T>	Предоставляет базовый интерфейс для абстракции множеств.

Список обобщенных коллекций

Обобщенный класс	Поддерживаемые основные интерфейсы	Назначение
Dictionary<TKey, TValue>	ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>	Представляет обобщенную коллекцию ключей и значений.
LinkedList<T>	ICollection<T>, IEnumerable<T>	Представляет двух-связный список.
List<T>	ICollection<T>, IEnumerable<T>, IList<T>	Последовательный список элементов с динамически изменяемым размером.
Queue<T>	ICollection, IEnumerable<T>	Обобщенная реализация очереди — списка, работающего по алгоритму “первый вошел — первый вышел” (FIFO).
SortedDictionary<TKey, TValue>	ICollection<T>, IDictionary<TKey, TValue>, IEnumerable<T>	Обобщенная реализация словаря — отсортированного множества пар “ключ/значение”.
SortedSet<T>	ICollection<T>, IEnumerable<T>, ISet<T>	Представляет коллекцию объектов, поддерживаемых в сортированном порядке без дублирования.
Stack<T>	ICollection, IEnumerable<T>	Обобщенная реализация стека — списка, работающего по алгоритму “последний вошел — первый вышел” (LIFO).

Класс List<T>

В классе List<T> реализуется обобщенный динамический массив. Он ничем принципиально не отличается от класса необобщенной коллекции ArrayList.

В этом классе реализуются интерфейсы ICollection, ICollection<T>, IList, IList<T>, IEnumerable и IEnumerable<T>. У класса List<T> имеются следующие конструкторы.

```
public List()

public List(IEnumerable<T> collection)

public List(int capacity)
```

Первый конструктор создаёт пустую коллекцию класса List с выбираемой по умолчанию первоначальной емкостью. Второй конструктор создаёт коллекцию типа List с количеством инициализируемых

элементов, которое определяется параметром collection и равно первоначальной ёмкости массива. Третий конструктор создаёт коллекцию типа List, имеющую первоначальную ёмкость, задаваемую параметром capacity. В данном случае ёмкость обозначает размер базового массива, используемого для хранения элементов коллекции. Ёмкость коллекции, создаваемой в виде динамического массива, может увеличиваться автоматически по мере добавления в неё элементов.

Среди методов коллекции List<T> можно выделить следующие:

- void Add(T item): добавление нового элемента в список;
- void AddRange(ICollection collection): добавление в список коллекции или массива;
- int BinarySearch(T item): бинарный поиск элемента в списке. Если элемент найден, то метод возвращает индекс этого элемента в коллекции. При этом список должен быть отсортирован.
- int IndexOf(T item): возвращает индекс первого вхождения элемента в списке;
- void Insert(int index, T item): вставляет элемент item в списке на позицию index;
- bool Remove(T item): удаляет элемент item из списка, и если удаление прошло успешно, то возвращает true;
- void RemoveAt(int index): удаление элемента по указанному индексу index;
- void Sort(): сортировка списка.

Структура KeyValuePair<TKey, TValue>

В пространстве имён System.Collections. Generic определена структура KeyValuePair<TKey, TValueX Она служит для хранения ключа и его значения и применяется в классах обобщенных коллекций, в которых хранятся пары "ключ-значение", как, например, в классе Dictionary<TKey, TValue>. В этой структуре определяются два следующих свойства:

```
public TKey Key { get; };
public TValue Value { get; };
```

В этих свойствах хранятся ключ и значение соответствующего элемента коллекции.

Для построения объекта типа KeyValuePair<TKey, TValue> служит конструктор:

```
public KeyValuePair(TKey key, TValue value)
```

где key обозначает ключ, а value — значение.

Класс Dictionary<TKey, TValue>

Класс Dictionary<TKey, TValue> позволяет хранить пары "ключ-значение" в коллекции как в словаре. Значения доступны в словаре по соответствующим ключам. В этом отношении данный класс аналогичен необобщенному классу Hashtable.

В классе Dictionary<TKey, TValue> реализуются интерфейсы IDictionary, IDictionary<TKey, TValue>, ICollection, ICollection<KeyValuePair<TKey, TValue>>, IEnumerable, IEnumerable<KeyValuePair<TKey, TValue», ISerializable и IDeserializationCallback. В двух последних интерфейсах поддерживается сериализация списка. Словари имеют динамический характер, расширяясь по мере необходимости.

В классе Dictionary<TKey, TValue> предоставляется немало конструкторов.

Ниже перечислены наиболее часто используемые из них.

```
public Dictionary()
public Dictionary(IDictionary<TKey, TValue> dictionary)
public Dictionary(int capacity)
```

В первом конструкторе создаётся пустой словарь с выбираемой по умолчанию первоначальной ёмкостью. Во втором конструкторе создаётся словарь с указанным количеством элементов dictionary. А в третьем конструкторе с помощью параметра capacity указывается ёмкость коллекции, создаваемой в виде словаря. Если размер словаря заранее известен, то, указав ёмкость создаваемой коллекции, можно исключить изменение размера словаря во время выполнения, что, как правило, требует дополнительных затрат вычислительных ресурсов.

Инициализация коллекции

Как инициализировать коллекцию в момент объявления:

```
List<int> list = new List<int>() { 1, 2, 3, 4, 5 };
Dictionary<int, string> dict = new Dictionary<int, string>() { { 1, "One" }, {2, "Two" } };
```

Перебор элементов коллекции без привязки к ее реализации

Как перебрать все элементы коллекции, независимо от того, как она реализована. Для этого, вместо того, чтобы писать циклы с обращением к элементам коллекции по индексу или ключу, воспользуйтесь конструкцией foreach. Она позволяет обратиться к объектам любого класса, реализующие интерфейс IEnumerable (или IEnumerable<T>)

```
using System;
using System.Collections.Generic;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            int[] array = { 1, 2, 3, 4, 5 };
            foreach (int n in array)
            {
                Console.Write("{0} ", n);
            }
            Console.WriteLine();
            List<DateTime> times = new List<DateTime>(new DateTime[] { DateTime.Now, DateTime.UtcNow });
            foreach (DateTime time in times)
            {
                Console.WriteLine(time);
            }
            Dictionary<int, string> numbers = new Dictionary<int, string>();
            numbers[1] = " One"; numbers[2] = " Two"; numbers[3] = " Three";
            foreach (KeyValuePair<int, string> pair in numbers)
            {
                Console.WriteLine("{0}", pair);
            }
        }
    }
}
```

Изменение порядка элементов массива на обратный

1 способ. Самостоятельно

```
private static void Reverse<T>(T[] array)
{
    int left = 0, right = array.Length - 1;
    while (left < right)
    {
        T temp = array[left];
        array[left] = array[right];
        array[right] = temp;
        left++;
        right--;
    }
}
```

2 способ. Используя метод расширения Reverse

```
int[] array = new int[5] { 1, 2, 3, 4, 5 };
IEnumerable<int> reversed = array.Reverse<int>();
```

Этот код будет корректно работать на любой коллекции с интерфейсом IEnumerable<T>, однако возвращает он не массив, а объект-итератор (привет делегатам!), который будет перебирать элементы оригинальной коллекции в обратном порядке.

Какой способ предпочесть, вы решаете, исходя из своих потребностей.

Извлечение уникальных элементов из коллекции

Задача. У вас имеется коллекция объектов, на основе которой вы хотите сгенерировать новую, содержащую по одной копии каждого объекта.

Решение. Чтобы сгенерировать коллекцию без повторяющихся элементов, вы должны отслеживать все элементы оригинальной коллекции и добавлять в новую лишь те, которые раньше не встречались. Рассмотрим пример:

```
private static ICollection<T> GetUniques<T>(ICollection<T> list)
{
   // Для отслеживания элементов используйте словарь
   Dictionary<T, bool> found = new Dictionary<T, bool> ();
   List<T> uniques = new List<T>();
   // Этот алгоритм сохраняет оригинальный порядок элементов
   foreach (T val in list)
   {
      if (!found.ContainsKey(val))
      {
         found[val] = true;
         uniques.Add(val);
      }
   }
   return uniques;
}
```

Лямбда-выражения

C# поддерживает способность обрабатывать события “встроенным образом”, назначая блок операторов кода непосредственно событию с использованием анонимных методов вместо построения отдельного метода, подлежащего вызову делегатом. Лямбда-выражения — это всего лишь лаконичный способ записи анонимных методов, который в конечном итоге упрощает работу с типами делегатов .NET.

Чтобы подготовить фундамент для изучения лямбда-выражений, создадим новое консольное приложение по имени SimpleLambdaExpressions. Теперь займемся методом FindAll () обобщенного типа List<T>. Этот метод может быть вызван, когда нужно извлечь подмножество элементов из коллекции, и он имеет следующий прототип:

```
// Метод класса System.Collections.Generic.List<T>.

public List<T> FindAll(Predicate<T> match)
```

Как видите, этот метод возвращает объект List<T>, представляющий подмножество данных. Также обратите внимание, что единственный параметр FindAll () — обобщенный делегат типа System.Predicate<T>. Этот делегат может указывать на любой метод, возвращающий bool и принимающий единственный параметр:

```
// Этот делегат используется методом FindAll() для извлечения подмножества.
public delegate bool Predicate<T>(T obj);
```

Когда вызывается FindAll (), каждый элемент в List<T> передаётся методу, указанному объектом Predicate<T>. Реализация этого метода будет производить некоторые вычисления для проверки соответствия элемента данных указанному критерию, возвращая в результате true или false. Если метод вернет true, то текущий элемент будет добавлен в List<T>, представляющий искомое подмножество. Прежде чем посмотреть, как лямбда-выражения упрощают работу с FindAll (), давайте решим эту задачу в длинной нотации, используя объекты делегатов непосредственно. Добавим в класс Program метод (по имени TraditionalDelegateSyntax()), который взаимодействует с System.Predicate<T> для обнаружения четных чисел в списке List<T> целочисленных значений:

```
using System;
using System.Collections.Generic;
// Эндрю Троелсон. Язык программирования C#5.0
// Понятие лямбда-выражения
namespace Лямбда_выражения
{
   class Program
   {
      static void Main(string[] args)
      {
         Console.WriteLine("***** Fun with Lambdas *****\n");
         TraditionalDelegateSyntax();
         Console.ReadLine();
      }
      static void TraditionalDelegateSyntax()
      {
 // Создать список целых чисел.
         List<int> list = new List<int>();
         list.AddRange(new int[] { 20, 1, 4, 8, 9, 4, 4 });
// Вызов FindAll() с использованием традиционного синтаксиса делегатов.
// Создаём обобщенный экземпляр обобщенного делегата используя встроенный делегат Predicate
         Predicate<int> predicate = new Predicate<int>(IsEvenNumber);
 // Создаём список целых чисел, используя метод FindAll, в который передаём делегат
         List<int> evenNumbers = list.FindAll(predicate);
         Console.WriteLine("Здесь только четные числа:");
         foreach (int evenNumber in evenNumbers)
         {
            Console.Write("{0}\t", evenNumber);
         }
         Console.WriteLine();
      }
// Цель для делегата Predicate<>.
      static bool IsEvenNumber(int i)
      {
 // Это чётное число?
         return (i % 2) == 0;
      }
   }
}
```

Хотя этот традиционный подход к работе с делегатами функционирует ожидаемым образом, метод IsEvenNumber () вызывается только при очень ограниченных условиях; в частности, когда вызывается FindAll (), который взваливает на нас все заботы относительно определения метода. Если бы вместо этого использовался анонимный метод, код стал бы существенно яснее. Рассмотрим следующий новый метод в классе Program:

```
using System;
using System.Collections.Generic;
// Эндрю Троелсон. Язык программирования C#5.0
// Понятие лямбда-выражения
namespace Лямбда_выражения_00200
{
   class Program
   {
      static void Main(string[] args)
      {
      }
      static void AnonymousMethodSyntax()
      {
// Создать список целых.
         List<int> list = new List<int>();
         list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
// Теперь использовать анонимный метод.
         List<int> evenNumbers = list.FindAll(delegate (int i)
                                 { return (i % 2) == 0; });
// Вывод чётных чисел.
         Console.WriteLine("Here are your even numbers:");
         foreach (int evenNumber in evenNumbers)
         {
            Console.Write("{0}\t", evenNumber);
         }
         Console.WriteLine();
      }
   }
}
```

Для дальнейшего упрощения вызова FindAll () можно применять лямбда-выражения, Используя этот новый синтаксис, вообще не приходится иметь дело с лежащим в основе объектом делегата. Рассмотрим следующий новый метод в классе Program:

```
static void LambdaExpressionSyntax()
{
// Создать список целых.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });
// Теперь использовать лямбда-выражение C#.
    List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
// Вывод чётных чисел.
    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
        Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

Здесь обратите внимание на довольно странный оператор кода, передаваемый методу FindAll (), который в действительности и является лямбда-выражением. В этой модификации примера вообще нет никаких следов делегата Predicate<T> (как и ключевого слова delegate). Всё, что указано вместо них — это лямбда-выражение: i => (i % 2) == 0

Лямбда-выражения могут применяться везде, где используется анонимный метод или строго типизированный делегат (обычно в более лаконичном виде). “За кулисами” компилятор C# транслирует лямбда-выражение в стандартный анонимный метод, использующий тип делегата Predicate<T>

Использование преимуществ контравариантности

Задача. В предыдущих версиях .NET(до 4) возникали ситуации, в которых делегаты вели себя неожиданным образом. Например, делегат с параметром-типом базового класса, по идее, должен легко присваиваться делегатам с производными параметрами-типами, поскольку любой делегат, вызываемый из базового класса, должен позволять вызывать себя из производного класса. Ситуацию иллюстрирует код, приведенный ниже.

```
using System;
namespace ConsoleApplication1
{
   class Program
   {
      class Shape
      {
         public void Draw() { Console.WriteLine("Drawing shape"); }
      };
      class Rectangle : Shape
      {
         public void Expand() { /*...*/ }
      };
// A делегат и метод определены так:
      delegate void ShapeAction<T>(T shape);
      static void DrawShape(Shape shape)
      {
         if (shape != null)
         {
            shape.Draw();
         }
      }
      static void Main(string[] args)
      {
      }
   }
}
```

Решение. В версии .NET 4 контравариантность делегатов решила проблему, и вы можете присваивать менее специфичные делегаты более специфичным. Теперь параметр-тип T определён с модификатором in, и это означает, что делегат не возвращает T. В коде, приведённом далее, параметр-тип делегата модифицирован ключевым словом in:

```
using System;
namespace ConsoleApplication1
{
    class Shape
    {
        public void Draw() { Console.WriteLine("Drawing shape"); }
    };
    class Rectangle : Shape
    {
        public void Expand() { /*...*/ }
    };
    class Program
    {
        delegate void ShapeAction<in T>(T shape);
        static void DrawShape(Shape shape)
        {
            if (shape != null)
            {
                shape.Draw();
            }
        }
        static void Main(string[] args)
        {
// Очевидно, что этот код должен работать
        ShapeAction<Shape> action = DrawShape;
            action(new Rectangle());
/* Интуитивно понятно, что любой метод, удовлетворяющий делегату ShapeAction<Shape>, должен
работать  с объектом Rectangle, потому что Rectangle является  производным от Shape. Всегда есть
возможность присвоить менее специфичный _метод более специфичному делегату, но до появления
версии .NET 4  нельзя было присвоить менее специфичный делегат_ более специфичному делегату.
Это очень важное различие. Теперь это можно, поскольку параметр-тип помечен  модификатором "in"
*/
// Следующие действия были возможны до появления .NET 4
            ShapeAction<Rectangle> rectActionI = DrawShape;
            rectActionI(new Rectangle());
// Take Advantage of Contravariance 293
// А это было невозможно до появления .NET 4
            ShapeAction<Rectangle> rectAction2 = action;
            rectAction2(new Rectangle());
            Console.ReadKey();
        }
    }
}
```

Пример метода расширений

```
using System;
using ExtensionMethods;
// Пример метода расширений (слово this перед параметром)
// https://msdn.microsoft.com/ru-ru/library/bb383977.aspx
namespace ExtensionMethods
{
   public static class MyExtensions
   {
// Создали метод, который добавляет (расширяет список методов) метод WordCount в список методов типа String
      public static int WordCount(this String str)
      {
         return str.Split(new char[] { ' ', '.', '?' },
                     StringSplitOptions.RemoveEmptyEntries).Length;
      }
   }
}
namespace MyProgram
{
   class Program
   {
      static void Main()
      {
         string s = "Hello Extension Methods";
// Теперь в классе s появился новый метод
         int i = s.WordCount();
      }
   }
}
```

Linq

Linq - использует следующие программные конструкции:

- неявная типизация локальных переменных;
- синтаксис инициализации объектов и коллекций;
- лямбда-выражения;
- расширяющие методы;
- анонимные типы.

В основу LINQ положено понятие запроса, в котором определяется информация, получаемая из источника данных. Например, запрос списка рассылки почтовых сообщений заказчикам может потребовать предоставления адресов всех заказчиков, проживающих в конкретном городе; запрос базы данных товарных запасов — список товаров, запасы которых исчерпались на складе; а запрос журнала, регистрирующего интенсивность использования Интернета, — список наиболее часто посещаемых веб-сайтов. И хотя все эти запросы отличаются в деталях, их можно выразить, используя одни и те же синтаксические элементы LINQ. Как только запрос будет сформирован, его можно выполнить. Это делается, в частности, в цикле foreach. В результате выполнения запроса выводятся его результаты. Поэтому использование запроса может быть разделено на две главные стадии. На первой стадии запрос формируется, а на второй — выполняется. Таким образом, при формировании запроса

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

Для обращения к источнику данных по запросу, сформированному средствами LINQ, в этом источнике должен быть реализован интерфейс IEnumerable. Он имеет две формы: обобщённую и необобщённую. Как правило, работать с источником данных легче, если в нём реализуется обобщённая форма IEnumerable<T>, где T обозначает обобщённый тип перечисляемых данных. Здесь и далее предполагается, что в источнике данных реализуется форма интерфейса IEnumerable<T>. Этот интерфейс объявляется в пространстве имён System. Collections . Generic. Класс, в котором реализуется форма интерфейса IEnumerable<T>, поддерживает перечисление, а это означает, что его содержимое может быть получено по очереди или в определённом порядке. Форма интерфейса IEnumerable<T> поддерживается всеми массивами в C#. Поэтому на примере массивов можно наглядно продемонстрировать основные принципы работы LINQ. Следует, однако, иметь в виду, что применение LINQ не ограничивается одними массивами.

Простой LINQ запрос

```
// Сформировать простой запрос LINQ
using System;
// Обязательно подключить
using System.Linq;
class SimpQuery
{
   static void Main()
   {
// Все массивы в C# неявным образом преобразуются в форму интерфейса IEnumerable<T>.
// Благодаря этому любой массив в C# может служить в качестве источника данных, извлекаемых
// по запросу LINQ.
      int[] nums = { 1, -2, 3, 0, -4, 5 };
// Создадим запрос, получающий только положительные числа
// posNums - переменная запроса.
// Ей присваивается результат выполнения запроса
      var posNums = from n // n -переменная диапазона (как в foreach)
                    in nums      // источник данных
                 where n > 0   // предикат (условие) - фильтр данных
                 select n;        //какое данное получаем. В сложных запросах мы здесь можно указать,
                                  // например, фамилию адресата вместо всего адреса
      Console.Write("Положительные числа: ");
// Выполняем запрос и выводим положительные числа на экран
      foreach (int i in posNums) Console.Write(i + " ");
      Console.WriteLine();
   }
}
```

Два where (условия)

```
using System;
using System.Linq;
class TwoWheres
{
   static void Main()
   {
      int[] nums = { 1, -2, 3, -3, 0, -8, 12, 19, 6, 9, 10 };
// Создаём запрос на выборку положительных чисел меньше 10
      var posNums = from n in nums
                    where n > 0
                    where n < 10
                    select n;
      Console.Write("Положительные числа меньше 10: ");
      foreach (int i in posNums) Console.Write(i + " ");
      Console.WriteLine();
   }
}
```

Еще один пример с where

```
// Еще один where
using System;
using System.Linq;
class WhereDemo2 {
  static void Main() {
    string[] strs = { ".com", ".net", "hsNameA.com", "hsNameB.net",
                      "test", ".network", "hsNameC.net", "hsNameD.com" };
// Create a query that obtains Internet addresses that end with .net
// Создадим запрос, который получает все Интернет-адреса, заканчивающиеся на .net
    var netAddrs = from addr in strs
                   where addr.Length > 4
                   && addr.EndsWith(".net", StringComparison.Ordinal)
// Он  возвращает логическое значение true, если вызывающая его строка оканчивается
// последовательностью символов, указываемой в качестве аргумента этого метода.
// Сортировка результатов запроса с помощью оператора orderby
                   select addr;
// Выполним запрос и выведем результаты
    foreach(var str in netAddrs) Console.WriteLine(str);
  }
}
```

Демонстрация OrderBy и преобразования в список

```
// Демонстрация OrderBy.
using System;
using System.Collections.Generic;
using System.Linq;
class OrderbyDemo
{
   static void Main()
   {
      int[] nums = { 10, -19, 4, 7, 2, -5, 0 };
// Запрос, который получает значения в отсортированном порядке
      var posNums = from n in nums
                    where n>0  orderby n descending
                    select n;
      Console.Write("Значения по возрастанию: ");
// Преобразовать в список (для примера такой возможности)
      List<int> a = posNums.ToList<int>();
// Execute the query and display the results.
      foreach (int i in posNums) Console.Write(i + " ");
      nums[1] = 10;
      Console.WriteLine("\n"+posNums.Sum());
      Console.WriteLine();
   }
}
```

Демонстрация использования LINQ с массивом пользовательских данных

```
using System;
using System.Linq;
class EmailAddress
{
   public string Name { get; set; }
   public string Address { get; set; }
   public EmailAddress(string n, string a)
   {
      Name = n;
      Address = a;
   }
}
class SelectDemo2
{
   static void Main()
   {
      EmailAddress[] addrs = {
       new EmailAddress("Herb", "Herb@HerbSchildt.com"),
       new EmailAddress("Tom", "Tom@HerbSchildt.com"),
       new EmailAddress("Sara", "Sara@HerbSchildt.com")
   };
      // Create a query that selects e-mail addresses.
      var eAddrs = from entry in addrs
                   select entry.Address;
      Console.WriteLine("The e-mail addresses are");
      // Execute the query and display the results.
      foreach (string s in eAddrs) Console.WriteLine("  " + s);
   }
}
```

Практика

“Астероиды” с коллекциями

Изменим программу так, чтобы можно было стрелять очередями.

Добавим в файл Game.cs пространство имён :

```
using System.Collections.Generic;
```

Вместо одной пули создадим коллекцию пуль:

```
static List<Bullet> bullets=new List<Bullet>();
```

При нажатии на выстрел пуля будет добавляться в коллекцию:

```
if (e.KeyCode == Keys.ControlKey)  bullets.Add(new Bullet(new Point(ship.Rect.X + 10, ship.Rect.Y + 4),
new Point(4, 0), new Size(4, 1)));
```

В методе Draw класс Game теперь нужно выводить все пули:

```
foreach(Bullet b in bullets)  b.Draw();
```

В методе Update:

```
foreach (Bullet b in bullets) b.Update();
```

Так же существенно изменится столкновение, поэтому приведем код Update класса Game полностью:

```
static public void Update()
{
        foreach (BaseObject obj in objs) obj.Update();
        foreach (Bullet b in bullets) b.Update();
        for (int i=0;i<asteroids.Length;i++)
        {
                if (asteroids[i] != null)
                {
                        asteroids[i].Update();
                        for(int j=0;j<bullets.Count;j++)
                         if (asteroids[i]!=null && bullets[j].Collision(asteroids[i]))
                        {
                                System.Media.SystemSounds.Hand.Play();
                                asteroids[i] = null;
                                        bullets.RemoveAt(j);
                                        j--;
                                continue;
                        }
                        if (asteroids[i]!=null && ship.Collision(asteroids[i]))
                        {
                                ship.EnergyLow(rnd.Next(1, 10));
                                System.Media.SystemSounds.Asterisk.Play();
                                if (ship.Energy <= 0) ship.Die();
                        }
                }
        }
}
```

Домашнее задание

1. Добавить в программу коллекцию астероидов. Как только она заканчивается (все астероиды сбиты), то формируется новая коллекция, в которой на 1 астероид больше.
2. Дана коллекция List<T>, требуется подсчитать, сколько раз каждый элемент встречается в данной коллекции.
   а) для целых чисел;
   б) *для обобщенной коллекции;
   в)**используя Linq
3. *Дан фрагмент программы:

```
Dictionary<string, int> dict = new Dictionary<string, int>()
 {
   {"four",4 },
   {"two",2 },
   { "one",1 },
   {"three",3 },
 };
    var d = dict.OrderBy(delegate(KeyValuePair<string,int> pair) { return pair.Value; });
    foreach (var pair in d)
   {
     Console.WriteLine("{0} - {1}", pair.Key, pair.Value);
   }
```

а) Свернуть обращение к OrderBy с использованием лямбда-выражения.
б) *Развернуть обращение к OrderBy с использованием делегата Predicate<T>

Дополнительные материалы

1. XNA
2. Ковариантность и контравариантность (Википедия)

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. “Программирование на языке высокого уровня”, Т.А. Павловская, 2009 г.
2. “Язык программирования C# 5.0 и платформа .NET 4.5”. Эндрю Троелсен, Питер 2013 г.
3. “C# 4.0. Полное руководство”, Г. Шилдт, 2011,
4. “C# 4.0 на примерах”, “БХВ-Петербург”, Ватсон Б.С.,”Вильямс”, 2011
5. MSDN