Технология LINQ to Object

Доцент каф. Евдошенко О.И.

Введение в коллекции

Основой для создания всех коллекций является реализация интерфейсов IEnumerator и IEnumerable (и их обобщенных двойников IEnumerator<T> и IEnumerable<T>).

Интерфейс IEnumerator представляет перечислитель, с помощью которого становится возможен последовательный перебор коллекции, например, в цикле foreach.

Интерфейс IEnumerable через свой метод GetEnumerator предоставляет перечислитель всем классам, реализующим данный интерфейс. Поэтому интерфейс IEnumerable (IEnumerable<T>) является базовым для всех коллекций.

```
// необобщенная коллекция ArrayList
ArrayList objectList = new ArrayList() { 1, 2, "string", 'c', 2.0f };

object obj = 45.8;
objectList.Add(obj);
objectList.Add("string2");
objectList.RemoveAt(0); // удаление первого элемента
```

Необобщенные коллекции

Необобщенные или простые коллекции определены в пространстве имен **System.Collections**. Их особенность состоит в том, что их функциональность, функциональные возможности описываются в интерфейсах, которые также находятся в этом пространстве имен.

Основные интерфейсы:

IEnumerable: определяет метод GetEnumerator. Данный метод возвращает перечислитель - то есть некоторый объект, реализующий интерфейс IEnumerator.

IEnumerator: реализация данного интерфейса позволяет перебирать элементы коллекции с помощью цикла foreach

ICollection: является основой для всех необобщенных коллекций, определяет основные методы и свойства для всех необобщенных коллекций (например, метод CopyTo и свойство Count). Данный интерфейс унаследован от интерфейса IEnumerable, благодаря чему базовый интерфейс также реализуется всеми классами необобщенных коллекций

IList: позволяет получать элементы коллекции по порядку. Также определяет ряд методов для манипуляции элементами: Add (добавление элементов), Remove/RemoveAt (удаление элемента) и ряд других.

IComparer: определяет метод int Compare(object x, object y) для сравнения двух объектов

Необобщенные коллекции

Интерфейсы реализуются следующими классами коллекций в пространстве имен **System.Collections**:

ArrayList: класс простой коллекции объектов. Реализует интерфейсы IList, ICollection, IEnumerable

BitArray: класс коллекции, содержащей массив битовых значений. Реализует интерфейсы ICollection, IEnumerable

Hashtable: класс коллекции, представляющей хэш-таблицу и хранящий набор пар "ключ-значение"

Queue: класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы ICollection, IEnumerable

SortedList: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы ICollection, IDictionary, IEnumerable

Stack: класс стека

Обобщенные коллекции

Классы обобщенных коллекций находятся в пространстве имен **System.Collections.Generic**. Функционал коллекций также по большей части описывается в обобщенных интерфейсах.

Основные интерфейсы обобщенных коллекций:

IEnumerable<T>: определяет метод GetEnumerator, с помощью которого можно получать элементы любой коллекции. Реализация данного интерфейса позволяет перебирать элементы коллекции с помощью цикла foreach

IEnumerator<T>: определяет методы, с помощью которых потом можно получить содержимое коллекции по очереди

ICollection<T>: представляет ряд общих свойств и методов для всех обобщенных коллекций (например, методы CopyTo, Add, Remove, Contains, свойство Count)

IList<T>: предоставляет функционал для создания последовательных списков

IComparer<T>: определяет метод Compare для сравнения двух однотипных объектов

IDictionary<TKey, TValue>: определяет поведение коллекции, при котором она должна хранить объекты в виде пар ключ-значение: для каждого объекта определяется уникальный ключ типа, указанного в параметре TKey, и этому ключу соответствует определенное значение, имеющее тип, указанный в параметре TValue

Обобщенные коллекции

Интерфейсы реализуются следующими классами коллекций в пространстве имен **System.Collections.Generic:**

List<T>: класс, представляющий последовательный список. Реализует интерфейсы IList<T>, ICollection<T>, IEnumerable<T>

Dictionary<TKey, TValue>: класс коллекции, хранящей наборы пар "ключ-значение". Реализует интерфейсы ICollection<T>, IEnumerable<T>, IDictionary<TKey, TValue>

LinkedList<T>: класс двухсвязанного списка. Реализует интерфейсы ICollection<T> и IEnumerable<T>

Queue<T>: класс очереди объектов, работающей по алгоритму FIFO("первый вошел -первый вышел"). Реализует интерфейсы ICollection, IEnumerable<T>

SortedSet<T>: класс отсортированной коллекции однотипных объектов. Реализует интерфейсы ICollection<T>, ISet<T>, IEnumerable<T>

SortedList<TKey, TValue>: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. Реализует интерфейсы ICollection<T>, IEnumerable<T>, IDictionary<TKey, TValue>

SortedDictionary<TKey, TValue>: класс коллекции, хранящей наборы пар "ключ-значение", отсортированных по ключу. В общем похож на класс SortedList<TKey, TValue>, основные отличия состоят лишь в использовании памяти и в скорости вставки и удаления

Stack<T>: класс стека однотипных объектов. Реализует интерфейсы ICollection<T> и IEnumerable<T>

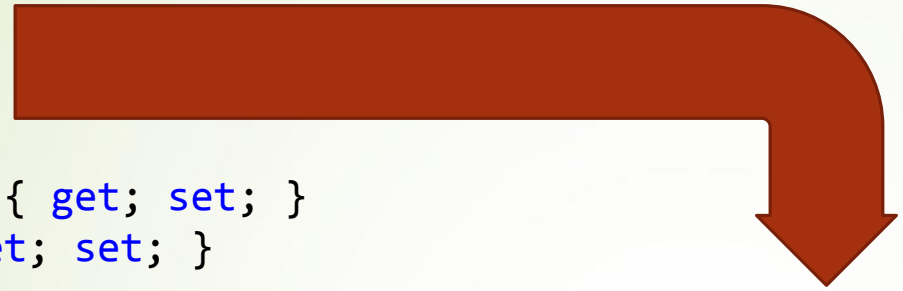
```
List<string> countries = new List<string>()
        { "Россия", "США", "Великобритания", "Китай" };
```

Использование собственного класса для типов элементов коллекции

```
public class Student

{
    public string Name { get; set; }
    public int Age { get; set; }

}
```



```
List<Student> theStudent = new List<Student>
 {
  new Student() { Name="Сергей", Age=17},
  new Student() { Name="Лида", Age=15},
  new Student() { Name="Виталий", Age=16},
  new Student() { Name="Ольга", Age=17}
  };

foreach (var ST in theStudent)
{
        Console.WriteLine(ST.Name + " " + ST.Age); }
}
```

Использование собственного класса для типов элементов коллекции

```
public class Студент
    {
        public string Фамилия { get; set; }
        public List<int> Оценки { get; set; }
    }
```

```
List<Студент> Студенты = new List<Студент>
      {
            new Студент {Фамилия="Иванов", Оценки= new List<int> {5, 4, 4, 5}},
            new Студент {Фамилия="Стороженко", Оценки= new List<int> {3, 3, 2, 3}},
            new Студент {Фамилия="Николаев", Оценки= new List<int> {3, 4, 4, 5}},
            new Студент {Фамилия="Петров", Оценки= new List<int> {2, 4, 3, 2}},
            new Студент {Фамилия="Левочкин", Оценки= new List<int> {3, 3, 4, 3}}
      };
```

```
Студент student = new Студент ();
        student.Фамилия = "Захаров";
        student.Оценки = new List<int>();
        student.Оценки.Add(5);
        student.Оценки.Add(2);
        Студенты.Add(student);
```

Фильтрация данных в LINQ-запросе

```
var СписокДвоечников = from Студент in Студенты
                       from Оценка in Студент.Оценки
                       where Оценка <= 2
                       orderby Студент.Фамилия
                       select new { Студент.Фамилия, Оценка};
```

```
textBox1.Text += string.Format("Студент {0} " +
          "имеет оценку: {1}\r\n", Учащиеся.Фамилия,  Учащиеся.Оценка);
```

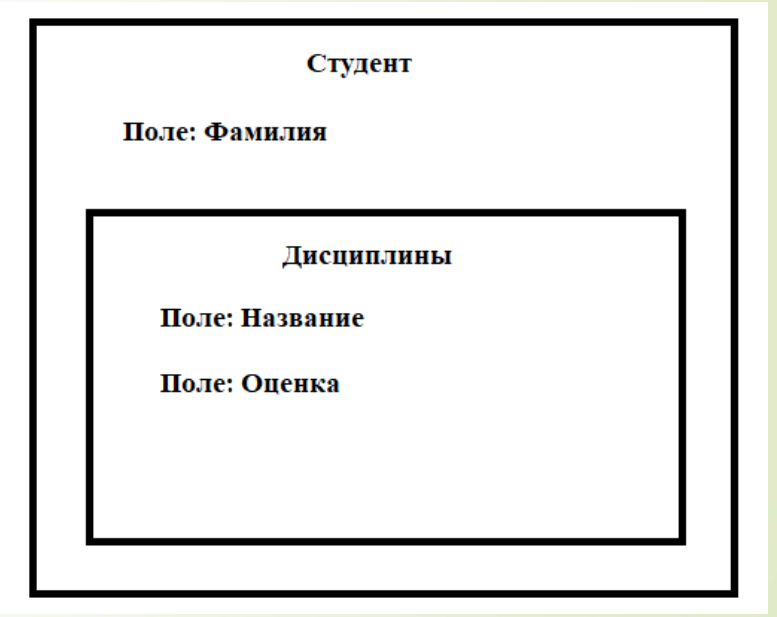
```
 var СписокДвоечников = from Студент in Студенты
                        from Оценка in Студент.Оценки
                        where Оценка <= 2
                        orderby Студент.Фамилия
                        select new {Студент,Оценка};
```

```
textBox1.Text += string.Format("Студент {0} " +
          "имеет оценку: {1}\r\n", Учащиеся.Студент.Фамилия,  Учащиеся.Оценка);
```

Использование собственного класса для типов элементов коллекции

```
public class Студент
    {
        public string Фамилия { get; set; }
        public List<Subject> Дисциплины { get; set; }
    }
```

```
public class Subject
    {
        public string Название { get; set; }
        public int Оценка { get; set; }
    }
```



Использование собственного класса для типов элементов коллекции

```
List<Студент> Студенты = new List<Студент>
        {
            new Студент {Фамилия="Зиборов",
          Дисциплины= new List<Subject>
       {
          new Subject {Название="Математика",Оценка=2},
         new Subject {Название="Русский язык",Оценка=2},
          new Subject {Название="Литература",Оценка=3}
       }
                          },

            new Студент {Фамилия="Стороженко",
           Дисциплины=new List<Subject>
       {
           new Subject {Название="Математика",Оценка=5},
           new Subject {Название="Русский язык",Оценка=3},
           new Subject {Название="Литература",Оценка=4}
       }
                           }
          }
```

Добавление данных в коллекцию

```
Студент student_mark = new Студент();

student_mark.Фамилия = "Захаров";
student_mark.Дисциплины = new List<Subject>();
student_mark.Дисциплины.Add(new Subject { Название = "Информатика", Оценка = 2 });

student_mark.Дисциплины.Add(new Subject { Название = "Физика", Оценка = 3 });

student_mark.Дисциплины.Add(new Subject { Название = "Программирование", Оценка = 3 });

            Студенты.Add(student_mark);
```

Фильтрация данных в LINQ-запросе

```
var СписокДвоечников = from Студент in Студенты
                       from Оценки in Студент.Дисциплины
                       where Оценки.Оценка <= 2
                       orderby Студент.Фамилия
                   select new { Студент.Фамилия, Оценки.Название, Оценки.Оценка };
```

```
foreach (var Учащиеся in СписокДвоечников)
      textBox1.Text += string.Format("Студент {0} по дисциплине - {1} " +
"имеет оценку: {2}\r\n", Учащиеся.Фамилия, Учащиеся.Название, Учащиеся.Оценка);
```

```
var СписокДвоечников = from Студент in Студенты
                       from Оценки in Студент.Дисциплины
                       where Оценки.Оценка <= 2
                       where Студент.Фамилия=="Левочкин"
                       orderby Оценки.Название
                   select new { Студент.Фамилия, Оценки.Название, Оценки.Оценка };
```

Операции SelectMany

Операция **SelectMany** используется для создания выходной последовательности с проекцией "один ко многим" из входной последовательности.

```
List<Employee> a1 = new List<Employee> { };

a1.Add(new Employee { id = 1, firstName = "Наталья", lastName = "Иванова",departament=1 });
a1.Add(new Employee { id = 2, firstName = "Ирина", lastName = "Золотова",departament=2 });
a1.Add(new Employee { id = 3, firstName = "Ольга", lastName = "Зощенко",departament=1 });
a1.Add(new Employee { id = 4, firstName = "Дмитрий", lastName = "Варлаков",departament=2 });
a1.Add(new Employee { id = 101, firstName = "Иван", lastName = "Павлов",departament=1 });


List<Departament> a2 = new List<Departament> { };

a2.Add(new Departament { id = 1, Name = "D1" });
a2.Add(new Departament { id = 2, Name = "D2" });
```

Операции SelectMany

```
var employee = a1
               .SelectMany(e => a2
                           .Where(eo => eo.id == e.departament)
                           .Select(eo => new
                           {
                               fam = e.lastName,
                               departament = eo.Name
                           }));


foreach (var item in employee)
               MessageBox.Show(item.fam+" "+item.departament);
```

Операции SelectMany

```
List<User> users = new List<User>
{
   new User {Name="Том", Age=23, Languages = new List<string> {"английский", "немецкий" }},
   new User {Name="Боб", Age=27, Languages = new List<string> {"английский", "французский" }},
   new User {Name="Джон", Age=29, Languages = new List<string> {"английский", "испанский" }},
   new User {Name="Элис", Age=24, Languages = new List<string> {"испанский", "немецкий" }}
};
```

```
var selectedUsers = from user in users
                    from lang in user.Languages
                    where user.Age < 28
                    where lang == "английский"
                    select user;

var selectedUsers = users.SelectMany(u => u.Languages,
                                     (u, l) => new { User = u, Lang = l })
                                     .Where(u => u.Lang == "английский" && u.User.Age < 28)
                                     .Select(u => u.User);
```

Метод **SelectMany()** в качестве первого параметра принимает последовательность, которую надо проецировать, а в качестве второго параметра - функцию преобразования, которая применяется к каждому элементу. На выходе она возвращает 8 пар "пользователь - язык" (new { User = u, Lang = l }), к которым потом применяется фильтр с помощью Where.

Операция Join

Операция **Join** выполняет внутреннее соединение по эквивалентности двух последовательностей на основе ключей, извлеченных из каждого элемента этих последовательностей.

Метод Join() принимает четыре параметра:

- второй список, который соединяем с текущим
- свойство объекта из текущего списка, по которому идет соединение
- свойство объекта из второго списка, по которому идет соединение
- новый объект, который получается в результате соединения

```
var res = from em in a1
          join d in a2 on em.departament equals d.id
          select new { Name = em.lastName, Departament = d.Name};

var res = a1.Join(a2, // второй набор
                  p => p.departament, // свойство-селектор объекта из первого набора
                  t => t.id, // свойство-селектор объекта из второго набора
                  (p, t) => new { Name = p.lastName, Departament = t.Name }); // результат
```

```
Сотрудник: Иванова, отдел: D1
Сотрудник: Золотова, отдел: D2
Сотрудник: Зощенко, отдел: D1
Сотрудник: Варлаков, отдел: D2
Сотрудник: Павлов, отдел: D1
```

Группировка данных по полю

```
var Запрос1 = from П in Продукты
              group П by П.Отдел  into g
              select g;
```

```
foreach (var Группа in Запрос1)
                {
                        textBox1.Text += Группа.Key.ToString();

foreach (var Товар in Группа)
                        {
                            textBox1.Text += Товар.Наименование;
                        }
                }
```

Группировка данных по критерию

```
var Запрос = from П in Продукты
             group П by new
             { Критерий = П.Цена > 90}
             into g
             select g;
```

```
foreach (var Группа in Запрос)
        {
            if (Группа.Key.Критерий == false)
                textBox1.Text += "\r\nЦены 90 руб или меньше:";
            else
                textBox1.Text += "\r\nЦены больше 90 руб:";
            foreach (var Прод in Группа)
            {  textBox1.Text += String.Format("\r\n{0} - {1}",
                                    Прод.Наименование, Прод.Цена);  }
        }
```

Использование агрегатных функций в группировке

```
var Запрос =  from p in Продукты
              group p by p.Цена > 90 into g
              select new
              {
                    g.Key,
                  СредЦенаПоГруппе = g.Average(p => p.Цена)
               };

Single СредЦенаПоГруппе1 =Запрос.ElementAt(0).СредЦенаПоГруппе;
Single СредЦенаПоГруппе2 = Запрос.ElementAt(1).СредЦенаПоГруппе;
```

```
var categories = from p in products
                 group p by p.Category into g
                  select new
                        {
                    Category = g.Key,
                    MaximumPrice = g.Max(p => p.UnitPrice),
                    MinimumPrice = g.Min(p => p.UnitPrice),
                    AveragePrice = g.Average(p => p.UnitPrice)
                      };
```

Операция GroupBy

```
Name="ИТ-31", Departament ="ЦТиК"
Name="ИТ-41", Departament ="ЦТиК"
Name="ФФ-31", Departament ="ФФ"
```

```
var rezult =
                groups.GroupBy(o => o.Departament);

            foreach (var departament in rezult)
            {
                Console.WriteLine($"Факультет: {departament.Key}");
                foreach (var group in departament)
                {
                    Console.WriteLine($"Группа: {group.Name}");
                }
            }
```

```
Факультет: ЦТиК
Группа: ИТ-31
Группа: ИТ-41
Факультет: ФФ
Группа: ФФ-31
```

Операция GroupBy

```
List<Group> groups = new List<Group>() {
                new Group {Name="ИТ-31", Departament="ЦТиК", count=19},
                new Group {Name="ИТ-41", Departament="ЦТиК", count=17},
                new Group {Name="ФФ-31", Departament="ФФ", count=15}
            };
var rezult =
groups.GroupBy(o => o.Departament).Select(g => new { Name = g.Key, Sum = g.Sum(n => n.count) });
```

```
Факультет: ЦТиК, количество студентов: 36
Факультет: ФФ, количество студентов: 15
```

```
var rezult = groups.GroupBy(p => p.Departament)
                        .Select(g => new
                        {
                            Name = g.Key,
                            Sum = g.Sum(n => n.count),
                            Group = g.Select(p => p)
                        });
```

```
Факультет: ЦТиК, количество студентов: 36
Группа: ИТ-31, количество студентов: 19
Группа: ИТ-41, количество студентов: 17

Факультет: ФФ, количество студентов: 15
Группа: ФФ-31, количество студентов: 15
```

Операция GroupJoin

Операция **GroupJoin** выполняет групповое соединение двух последовательностей на основе ключей, извлеченных из каждого элемента последовательностей.

```
var res = a2.GroupJoin(
          a1, // второй набор
          d => d.id, // свойство-селектор объекта из первого набора
          em => em.departament, // свойство-селектор объекта из второго набора
          (dep, emp) => new  // результирующий объект
          {
            Name = dep.Name,
            Employee = emp.Select(p=>p),
          });
```

```
Отдел: D1
ФИО сотрудника: Иванова
ФИО сотрудника: Зощенко
ФИО сотрудника: Павлов

Отдел: D2
ФИО сотрудника: Золотова
ФИО сотрудника: Варлаков
```

```
foreach (var spisok in res)
        {
            Console.WriteLine($"Отдел: {spisok.Name}");

            foreach (var employee in spisok.Employee)
            {
                Console.WriteLine($"ФИО сотрудника: {employee.lastName}");
            }
            Console.WriteLine();

        }
```

Класс Dictionary

Словарь (dictionary) представляет собой сложную структуру данных, позволяющую обеспечить доступ к элементам по ключу.

Add()
Добавляет в словарь пару "ключ-значение", определяемую параметрами key и value. Если ключ key уже находится в словаре, то его значение не изменяется, и генерируется исключение ArgumentException

ContainsKey()

Возвращает логическое значение true, если вызывающий словарь содержит объект key в качестве ключа; а иначе — логическое значение false

ContainsValue()

Возвращает логическое значение true, если вызывающий словарь содержит значение value; в противном случае — логическое значение false

Remove()

Удаляет ключ key из словаря. При удачном исходе операции возвращается логическое значение true, а если ключ key отсутствует в словаре — логическое значение false

Класс Dictionary

```
Dictionary<int, string> dic = new Dictionary<int, string>();
Console.WriteLine("Введите имя сотрудника: \n");
string s;
          for (int j = 0; j < 10; j++)
            {
                Console.Write("Name{0} --> ", j);
                s = Console.ReadLine();
                dic.Add(j, s);
                Console.Clear();
            }

// Получить коллекцию ключей
ICollection<int> keys = dic.Keys;
```

Использование собственного класса для типов элементов коллекции

```
  class Student
      {
          public string fio;
          public Dictionary<string, int> subject = new Dictionary<string, int>();
      }

 List<Student> clients = new List<Student>
        {
            new Student {fio = "Петров С.В.",
subject=new Dictionary<string, int> {{"Математика", 3},{"Русский язык", 4},
{"Литература",4}}},

            new Student {fio = "Антонов В.Д.",
subject=new Dictionary<string, int> {{"Математика", 2},{"Русский язык", 4},
{"Литература",4}}},

             new Student {fio = "Самоилик Д.Т.",
subject=new Dictionary<string, int> {{"Математика", 3},{"Русский язык", 2},
{"Литература",4}}}

        };
```

Фильтрация данных в LINQ-запросе

```
var СписокДвоечниковСловарь = from Студент in clients
                              from Оценки in Студент.subject
                              where Оценки.Value == 2
                              orderby Студент.fio
                              orderby Оценки.Key
                          select new { Студент.fio, Оценки.Key, Оценки.Value };


foreach (var Учащиеся in СписокДвоечниковСловарь)
                  textBox1.Text += string.Format("Студент {0} по дисциплине -
{1} " + "имеет оценку: {2}\r\n", Учащиеся.fio, Учащиеся.Key, Учащиеся.Value);
```

Решение задач

Дана целочисленная последовательность A. Сгруппировать элементы последовательности A, оканчивающиеся одной и той же цифрой, и на основе этой группировки получить последовательность строк вида «D:S», где D — ключ группировки (т. е. некоторая цифра, которой начинается хотя бы одно из чисел последовательности A), а S — среднее арифметическое четных чисел из A, которые начинаются цифрой D.

Решение задач

```
int[] a1 = new int[10] { 11, 30, 21, 116, 89, 79, 100, 22, 57, 20 };

            var query = a1.GroupBy(n =>

                n.ToString().Substring(0,1)).Select(g => new
            {
                d = g.Key,
                average = g.Where (n=>n%2==0).DefaultIfEmpty(0).Average()

            }).OrderBy(g => g.d);
```

```
1: 108

2: 21

3: 30

5: 0

7: 0

8: 0
```

Очередь Queue<T>

Класс Queue<T> представляет обычную очередь, работающую по алгоритму FIFO ("первый вошел - первый вышел").

У класса Queue<T> можно отметить следующие методы:

Dequeue: извлекает и возвращает первый элемент очереди

Enqueue: добавляет элемент в конец очереди

Peek: просто возвращает первый элемент из начала очереди без его удаления

```
Queue<int> numbers = new Queue<int>();

numbers.Enqueue(3); // очередь 3
numbers.Enqueue(5); // очередь 3, 5
numbers.Enqueue(8); // очередь 3, 5, 8

// получаем первый элемент очереди
int queueElement = numbers.Dequeue(); //теперь очередь 5, 8
```


```
Queue<Person> persons = new Queue<Person>();
persons.Enqueue(new Person() { Name = "Арбуз" });
persons.Enqueue(new Person() { Name = "Дыня" });
persons.Enqueue(new Person() { Name = "Тыква" });

// получаем первый элемент без его извлечения
Person pp = persons.Peek();
```

Коллекция Stack<T>

Класс Stack<T> представляет коллекцию, которая использует алгоритм LIFO ("последний вошел - первый вышел")
В классе **Stack** можно выделить три основных метода, которые позволяют управлять элементами:

Push: добавляет элемент в стек на первое место

Pop: извлекает и возвращает первый элемент из стека

Peek: просто возвращает первый элемент из стека без его удаления

```
Stack<int> numbers = new Stack<int>();

numbers.Push(3); // в стеке 3
numbers.Push(5); // в стеке 5, 3
numbers.Push(8); // в стеке 8, 5, 3

// так как вверху стека будет находиться число 8, то оно и извлекается
int stackElement = numbers.Pop(); // в стеке 5, 3
```

```
    Stack<Person> persons = new Stack<Person>();
    persons.Push(new Person() { Name = "Tom" });
    persons.Push(new Person() { Name = "Bill" });
    persons.Push(new Person() { Name = "John" });
    // Первый элемент в стеке
    Person person = persons.Pop(); // теперь в стеке Bill, Tom
```