# Абстрактные типы данных. Коллекции.

1. Коллекции. Абстрактные типы данных.
2. Пространство имен System.Collections.

## Абстрактные типы данных. Коллекции.

**Коллекция** в программировании — программный объект, содержащий в себе набор значений одного или различных типов, и позволяющий обращаться к этим значениям.

Назначение коллекции — служить хранилищем объектов и обеспечивать доступ к ним. Обычно коллекции используются для хранения групп однотипных объектов, подлежащих стереотипной обработке. Для обращения к конкретному элементу коллекции могут использоваться различные методы, в зависимости от того, какой абстрактный тип данных она реализует.

Различают:

* + **логическое** (абстрактный тип данных)
  + и **физическое** (реализация абстрактного типа данных) представление данных.

**Абстрактный тип данных** — это тип данных, который предоставляет для работы с элементами этого типа определённый набор функций, а также возможность создавать элементы этого типа при помощи специальных функций. Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения — в этом и заключается суть абстракции. Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями. Конкретные реализации АТД называются структурами данных.

**Основные АТД.**

1. **Множество.**

Множество ­ это неупорядоченная совокупность элементов. Для множеств определены операции:

1. проверки принадлежности элемента множеству,
2. включения элемента,
3. исключения элемента,
4. объединения множеств,
5. пересечения множеств,
6. вычитания множеств.
7. **Словарь.**

Ассоциативный массив, или словарь ­ это массив, доступ к элементам которого осуществляется не по номеру, а по некоторому ключу, т.е. это таблица, состоящая из пар ключ-значение.

В качестве ключа могут использоваться значения различных типов, единственное ограничение – тип ключа должен допускать сравнение на равенство.

Может вводиться запрет на дублирование ключей в коллекции. Если такого ограничения нет, то при обращении по дублирующемуся ключу может выдаваться либо n-е найденное значение, либо все значения с данным ключом.

Операции, определенные над словарем:

1. Добавление пары ключ значение.
2. Удаление пары ключ значение по ключу.
3. Получение значения по ключу.

**3. Очередь**

Очередь – набор данных, реализующий принцип хранения «FIFO» («первым пришёл – первым вышел»). В очереди постоянно доступен только один элемент — тот, который был добавлен самым первым из имеющихся. При добавлении нового элемента он попадает в очередь. Текущий элемент можно удалить — тогда текущим станет элемент, добавленный первым из оставшихся.

Операции, определенные над очередью:

1. Помещение элемента в очередь.
2. Извлечение элемента из очереди.
3. **Стек**

Стек – набор данных, реализующий принцип хранения «LIFO» («последним пришёл – первым вышел»). В стеке постоянно доступен только один элемент – тот, который был добавлен последним. Новый элемент может быть добавлен в стек, он станет текущим. Текущий элемент всегда можно удалить («взять») из стека, после этого становится доступен элемент, который был добавлен непосредственно перед ним.

Операции определенные над стеком

1. Помещение элемента в стек.
2. Удаление элемента из стека.
3. **Список**

Список – набор данных, к которым возможен последовательный доступ. В любой момент доступен первый и последний элемент списка. От любого элемента списка можно получить доступ к следующему и к предыдущему по порядку, таким образом, можно последовательно дойти до любого желаемого. Новый элемент может добавляться в начало или в конец списка.

Операции определенные над списком:

1. Добавление элемента в начало списка.
2. Добавление элемента в конец списка.
3. Получение элемента из начала списка.
4. Получение элемента из конца списка.
5. Получение следующего элемента списка.

**Физическое представление данных**

Структура данных – это программная единица, позволяющая хранить и обрабатывать какие либо связанные данные. Используется для реализации каких-либо абстрактных типов данных. Структура данных определяет то, как данные будут размещены в памяти и соответственно время, необходимое для выполнения тех или иных операций над хранимыми данными.

1. **Массив**

Массив – это структура данных имеющая постоянную длину, порядок расположения элементов в массиве совпадает с порядком расположения элементов в памяти компьютера. Операции, связанные с изменением размера массива, как правило, требуют перераспределения памяти под новый массив и переноса элементов из старого массива в новый.

Все массивы в С# имеют общий базовый класс Array, определенный в пространстве имен System. В нем есть несколько полезных методов, упрощающих работу с массивами, например, методы получения размерности, сортировки и поиска.

Все массивы в С# построены на основе базового класса Array, который содержит полезные для программиста свойства и методы, часть из которых перечислены в табл.

|  |  |  |
| --- | --- | --- |
| Имя | Вид | Описание |
| Length | Свойство | Количество элементов массива (по всем размерностям) |
| Rank | Свойство | Количество размерностей |
| BinarySearch | Статический метод | Бинарный поиск |
| Clear | Статический метод | Присваивание значений элементам массива по умолчанию |
| Copy | Статический метод | Копирование заданного диапазона элементов из одного массива в другой |
| CopyTo | Метод | Копирование всех элементов текущего одномерного массива в другой массив |
| GetValue | Метод | Получение значения элемента массива |
| IndexOf | Статический метод | Поиск первого вхождения элемента в массив |
| LastIndexOf | Статический метод | Поиск последнего вхождения элемента в массив |
| Reverse | Статический метод | Переворот массива |
| SetValue | Метод | Установка значения элемента массива |
| Sort | Статический метод | Сортировка |

Методы Sort, IndexOf и BinarySearch являются статическими, поэтому к ним обращаются через имя класса, а не экземпляра, и передают в них имя массива. Двоичный поиск можно применять только для упорядоченных массивов.

Оператор foreach применяется для перебора элементов в специальным образом организованной группе данных. Массив является именно такой группой. Удобство этого вида цикла заключается в том, что нам не требуется определять количество элементов в группе и выполнять их перебор по индексу: мы просто указываем на необходимость перебрать все элементы группы.

Синтаксис оператора:

foreach ( тип имя in выражение ) тело цикла

Пример

int[] а = {24, 50, 18, 3, 16, -7, 9, -1 } ;

Вывод этого массива на экран с помощью оператора foreach выглядит следующим образом:

foreach ( int х in а ) Console.WriteLine( х );

1. **Связный список**

Связный список – это структура данных, состоящая из узлов, каждый из которых содержит как собственные данные, так и одну или две ссылки на следующие и/или предыдущие узлы списка. Расположение элементов списков в памяти компьютера не совпадает с расположением элементов в списке. Обращение к произвольному элементу списка требует прохода по списку. Операции добавления и удаления элементов из списка не требуют перераспределения памяти под всю структуру данных.

**Пример 14\_1**

class LPoint

{

public int data;

public LPoint next;

public LPoint(int number)

{

data = number;

next = null;

}

public override string ToString()

{

return data.ToString();

}

}

class Program

{

static LPoint MakeList(int size)

{

Random rnd = new Random();

LPoint beg = new LPoint(rnd.Next(0,100));//первый элемeнт списка

for (int i = 1; i < size; i++)

{

LPoint newPoint = new LPoint(rnd.Next(0,100));

newPoint.next = beg;

beg = newPoint;

}

return beg;

}

static void PrintList(LPoint beg)

{

if (beg == null)

{

Console.WriteLine("Список пустой!");

return;

}

LPoint currentPoint = beg;

while (currentPoint != null)

{

Console.Write(currentPoint.ToString() + "\t");

currentPoint = currentPoint.next;

}

Console.WriteLine();

}

static LPoint DelPoint(LPoint beg, int key)

{

if (beg == null) return null;

LPoint point = beg;

if (beg.data == key)

{

beg = beg.next;

return beg;

}

while (point.next!=null&&point.next.data != key) point = point.next;

if (point.next != null)

{

LPoint temp = point.next;

point.next = temp.next;

}

else Console.WriteLine("Такого элемента в списке нет");

return beg;

}

static LPoint AddPoint(LPoint beg, int num)

{

Console.WriteLine("Введите значение элемента ");

int key = Convert.ToInt32(Console.ReadLine());

LPoint point=new LPoint(key);//добавим

if(num==1)

{

point.next=beg;

beg=point;

return beg;

}

LPoint current=beg;

for (int i = 1; i < num-1 && current.next != null; i++)

current = current.next;

if(current.next!=null)

{

point.next=current.next;

current.next=point;

}

else

{

Console.WriteLine("Нет элемента с этим номером! Добавляем в конец списка!");

current.next = point;

}

return beg;

}

static void Main(string[] args)

{

int size;

Console.WriteLine("Введите количество элементов в списке");

size = Convert.ToInt32(Console.ReadLine());

LPoint beg = MakeList(size);

PrintList(beg);

bool ok = true;

int answer;

while (ok)

{

try

{

Console.WriteLine("1.Удалить элемент из списка ");

Console.WriteLine("2. Добавить элемент в список ");

Console.WriteLine("3. Конец");

answer = Convert.ToInt32(Console.ReadLine());

switch (answer)

{

case 1:

{

Console.WriteLine("Введите значение удаляемого элемента ");

int key = Convert.ToInt32(Console.ReadLine());

beg = DelPoint(beg, key);

PrintList(beg);

break;

}

case 2:

{

Console.WriteLine("Введите номер элемента для добавления ");

int number = Convert.ToInt32(Console.ReadLine());

beg = AddPoint(beg, number);

PrintList(beg);

break;

}

case 3:

{

ok = false;

break;

}

}//switch

}//try

catch (Exception e)

{

Console.WriteLine(e.Message);

ok = true;

}

}//while

}

}

1. **Хеш-таблица**

Хеш-таблица – это структура данных представляющая собой комбинацию массива и списков. Каждый элемент массива представляет собой указатель на начало списка. При помещении элемента в структуру сначала вычисляется хеш-функция (функция осуществляющая преобразование входного массива данных произвольной длины в выходную битовую строку фиксированной длины) и в зависимости от полученного результата элемент помещается в один из списков, полученное значение является индексом массива. Число хранимых элементов, делённое на размер массива (число возможных значений хэш-функции) называется коэффициентом заполнения хэш-таблицы и является важным параметром, от которого зависит среднее время выполнения операций. Также хеш-функция должна обеспечивать равномерное распределение получаемых значений.

**Пример 14\_2**

class LPoint

{

public int key;

public string value;

public LPoint next;

static Random rnd = new Random();

public LPoint(string s)

{

value = s;

key = GetHashCode();

next = null;

}

public override string ToString()

{

return key + ":" + value.ToString();

}

public override int GetHashCode()

{

int code = 0;

foreach (char c in value)

code += (int)c;

return code;

}

}

class HTable

{

public LPoint[] table;

public int Size;

public HTable()

{

Size = 10;

table = new LPoint[10];

}

public bool Add(string s)

{

LPoint point = new LPoint(s);

if (s == null) return false;

int index = Math.Abs(point.GetHashCode()) % Size;

if (table[index] == null) table[index] = point;

else

{

LPoint cur = table[index];

if (string.Compare(cur.ToString(), point.ToString()) == 0)

return false;//одинаковые не добавляем

while (cur.next != null)

{

if (string.Compare(cur.ToString(), point.ToString()) == 0)

return false;//одинаковые не добавляем

cur = cur.next;

}

cur.next = point;

}

return true;

}

public void Print()

{

if (table == null) { Console.WriteLine("Таблица пустая!"); return; }

for (int i = 0; i < Size; i++)

{

if (table[i] == null) Console.WriteLine(i + " : ");

else

{

Console.Write(i + " : ");

LPoint p = table[i];

while (p != null)

{

Console.Write(p.ToString() + "\t");

p = p.next;

}

Console.WriteLine();

}

}

}

}

class Program

{

static void Main(string[] args)

{

string[] arr = new string[22];

arr[0] = "aaa";

arr[1] = "aab"; arr[2] = "aba"; arr[3] = "baa"; arr[4] = "abb";

arr[5] = "bba"; arr[6] = "aac"; arr[7] = "caa"; arr[8] = "aca"; arr[9] = "abc";

arr[10] = "bca"; arr[11] = "acc"; arr[12] = "cca"; arr[13] = "bcb"; arr[14] = "bbc";

arr[15] = "ccb"; arr[16] = "bbb"; arr[17] = "ccc"; arr[18] = "acb"; arr[19] = "cba";

arr[20] = "ccb"; arr[21] = "acc";

HTable ht = new HTable();

foreach (string s in arr)

ht.Add(s);

ht.Print();

}

}

## 2. Пространство имен System.Collections.

В С# под коллекцией понимается группа объектов. Пространство имен System.Collections содержит множество интерфейсов и классов, которые определяют и реализуют коллекции различных типов.

**Подключение:** using System.Collections;

Коллекции упрощают программирование, предлагая уже готовые решения для построения структур данных, разработка которых "с нуля" отличается большой трудоемкостью. Речь идет о встроенных коллекциях, которые поддерживают, например, функционирование стеков, очередей и хеш-таблиц. Коллекции пользуются большой популярностью у всех С#-программистов.

В пространстве имен System.Collections определены наборы стандартных коллекций и интерфейсов, которые реализованы в этих коллекциях.

1. **Интерфейсы**

|  |  |
| --- | --- |
| **Интерфейс** | **Назначение** |
| ICollection | Определяет общие характеристики (например, размер) для набора элементов |
| IComparer | Позволяет сравнивать два объекта |
| IDictionary | Позволяет представлять содержимое объекта в виде пар «имя-­значение» |
| IDictionaryEnumenator | Используется для нумерации содержимого объекта, поддерживающего интерфейс IDictionary. |
| IEnumerator | Обычно используется для поддержки оператора foreach в отношении объектов |
| IEnumerable | Возвращает интерфейс IEnumerator для указанного объекта |
| IHashCodeProvider | Возвращает хеш-код для реализации типа с применением выбранного пользователем алгоритма хеширования |
| IList | Поддерживает методы добавления, удаления и индексирования элементов в списке объектов |

1. **Интерфейс ICollection** содержит основные методы и свойства, без которых не может обойтись ни одна коллекция:
   * свойство Count - количество элементов, хранимых в коллекции в данный момент.
   * метод void CopyTo (Array target, int startIdx) - обеспечивает переход от коллекции к стандартному С#-массиву.
   * метод IEnumerator GetEnumerator () - возвращает нумератор коллекции.
2. **Интерфейс IList** наследует интерфейс ICollection и определяет поведение коллекции, доступ к элементам которой разрешен посредством индекса с отсчетом от нуля. Помимо методов, определенных в интерфейсе ICollection, интерфейс IList определяет и собственные методы.

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| int Add(object obj) | Добавляет объект objв вызывающую коллекцию. Возвращает индекс, по которому этот объект сохранен |
| void Clear() | Удаляет все элементы из вызывающей коллекции |
| bool Contains(object obj) | Возвращает значение true , если вызывающая коллекция содержит объект, переданный в параметре obj*,* и значение false в противном случае |
| int IndexOf(object obj) | Возвращает индекс объекта obj,если он (объект) содержится в вызывающей коллекции. Если объект obj не обнаружен, метод возвращает -1 |
| void Insert(int idx, object obj) | Вставляет в вызывающую коллекцию объект objпо индексу, заданному параметром idx.Элементы, находившиеся до этого по индексу idx и далее, смещаются вперед, чтобы освободить место для вставляемого объекта obj. |
| void Remove(object obj) | Удаляет из вызывающей коллекции объект, расположенный по индексу, заданному параметром idx. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы ликвидировать образовавшуюся "брешь" |
| void RemoveAt(int idx) | Удаляет первое вхождение объекта objиз вызывающей коллекции. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы ликвидировать образовавшуюся "брешь" |

В классе IList также определены свойства:

* + bool IsFixedSize { get; } – если свойство равно true, то в такую коллекцию нельзя вставлять и удалять элементы .
  + bool IsReadOnly { get; } - если свойство равно true, то содержимое коллекции не меняется.

1. **Интерфейс IDictionary** определяет поведение коллекции, которая устанавливает соответствие между уникальными ключами и значениями. Ключ - это объект, который используется для получения соответствующего ему значения. Следовательно, коллекция, которая реализует интерфейс IDictionary, служит для хранения пар ключ/значение. Сохраненную однажды пару можно затем извлечь по заданному ключу. Интерфейс IDictionary наследует интерфейс ICollection.

|  |  |
| --- | --- |
| **Метод** | **Описание** |
| void Add( object k, object v) | Добавляет в вызывающую коллекцию пару ключ/значение, заданную параметрами *к* и *v.* Ключ *к* не должен быть нулевым |
| void Clear() | Удаляет все пары ключ/значение из вызывающей коллекции |
| bool Contains (object k) | Возвращает значение true , если вызывающая коллекция содержит объект *к* в качестве ключа. В противном случае возвращает значение false |
| iDictionaryEnumerator GetEnumerator() | Возвращает нумератор для вызывающей коллекции |
| void Remove (object *k)* | Удаляет элемент, ключ которого равен значению *k* |

**Свойства:**

* + bool isFixedSize - равно значению true, если словарь имеет фиксированный размер.
  + bool isReadOnly - равно значению true , если словарь предназначен только для чтения.
  + iCollection Keys - получает коллекцию ключей.
  + iCollection Values - получает коллекцию значений.

Для хранения пары ключ/значение, используются объекты структуры **DictionaryEntry.** В ней определены свойства:

* + public object Key { get; set; }
  + public object Value { get; set; }

1. **Коллекции**

Классы коллекций делятся на три основных категории:

* + **общего назначения,**
  + специализированные,
  + ориентированные на побитовую организацию данных.

Классы общего назначения можно использовать для хранения объектов любого типа. Битовые – предназначены для хранения битовой информации. Коллекции специального назначения разрабатываются для обработки данных конкретного типа. Мы будем рассматривать классы коллекций общего назначения:

|  |  |  |
| --- | --- | --- |
| **Класс** | **Назначение** | **Интерфейсы** |
| ArrayList | Массив, динамически  изменяющий свой размер | IList, ICollection, IEnumerable, ICloneable |
| Hashtable | Хеш-таблица | IDictionary, ICollection, IEnumerable,  ICloneable |
| Queue | Очередь | ICollection, ICloneable, IEnumerable |
| SortedList | Коллекция, отсортированная  по ключам. Доступ  к элементам — по ключу  или по индексу | IDictionary, ICollection, IEnumerable,  ICloneable |
| Stack | Стек | ICollection, IEnumerable |

### 2.1. Класс ArrayList

**Класс ArrayList** предназначен для поддержки динамических массивов, которые при необходимости могут увеличиваться или уменьшаться. Объект класса ArrayList представляет собой массив переменной длины, элементами которого являются объектные ссылки.

**Конструкторы:**

* + public ArrayList() – создает создания пустой массив с начальной емкостью, равной 8 элементам.
  + public ArrayList(ICollection с) - создает массив, который инициализируется элементами и емкостью коллекции, заданной параметром с.
  + public ArrayList(int capacity) - создает массив с заданным начальным размером.

В С# стандартные массивы имеют фиксированную длину, которая не может измениться во время выполнения программы. Это означает, что программист должен знать заранее, сколько элементов будет храниться в массиве. Но иногда до выполнения программы нельзя точно сказать, какого размера массив понадобится. В таких случаях и используется класс ArrayList.

**Основные элементы класса ArrayList**

|  |  |  |
| --- | --- | --- |
| **Элемент** | **Вид** | **Назначение** |
| Capacity | Свойство | Количество элементов, которые могут  храниться в массиве |
| Count | Свойство | Фактическое количество элементов массива |
| Item | Свойство | Получить или установить значение элемента по заданному индексу |
| Add | Метод | Добавление элемента в конец массива |
| AddRange | Метод | Добавление серии элементов в конец массива |
| BinarySearch | Метод | Двоичный поиск в отсортированном массиве или его части |
| Clear | Метод | Удаление всех элементов из массива |
| Clone | Метод | Поверхностное копирование1 элементов одного массива в другой массив |
| CopyTo | Метод | Копирование всех или части элементов массива в одномерный массив |
| GetRange | Метод | Получение значений подмножества элементов массива в виде объекта типа ArrayList |
| Index | Метод | Поиск первого вхождения элемента в массив (возвращает индекс найденного элемента или -1, если элемент не найден) |
| Insert | Метод | Вставка элемента в заданную позицию (по заданному индексу) |
| InserRange | Метод | Вставка группы элементов, начиная с заданной позиции |
| LastIndexOf | Метод | Поиск последнего вхождения элемента в одномерный массив |
| Remove | Метод | Удаление первого вхождения заданного элемента в массив |
| RemoveAt | Метод | Удаление элемента из массива по заданному индексу |
| RemoveRange | Метод | Удаление группы элементов из массива |
| Reverse | Метод | Изменение порядка следования элементов на обратный |
| SetRange | Метод | Установка значений элементов массива в заданном диапазоне |
| Sort | Метод | Упорядочивание элементов массива или его части |
| TrimToSize | Метод | Установка емкости массива равной фактическому количеству элементов |

**Пример 14\_3.**

static void Main(string[] args)

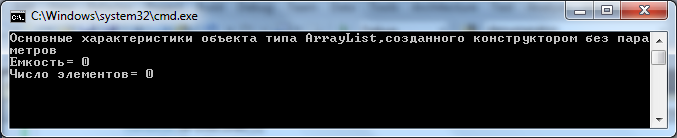
{

ArrayList al1 = new ArrayList();

Console.WriteLine("Основные характеристики объекта типа ArrayList, созданного конструктором без параметров");

Console.WriteLine("Емкость= "+al1.Capacity);

Console.WriteLine("Число элементов= " + al1.Count);



Console.WriteLine("Добавляем 5 элементов");

for (int i = 0; i < 5; i++)

al1.Add(i + 1);

Console.WriteLine("Емкость= " + al1.Capacity);

Console.WriteLine("Число элементов= " + al1.Count);

foreach (int num in al1)

Console.Write(num + "\t");

Console.WriteLine();



Console.WriteLine("Добавляем еще 5 элементов");

for (int i = 5; i < 10; i++)

al1.Add(i + 1);

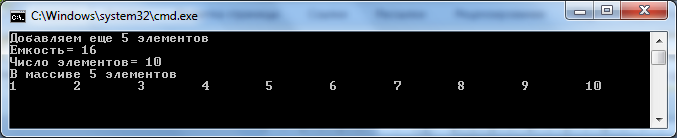
Console.WriteLine("Емкость= " + al1.Capacity);

Console.WriteLine("Число элементов= " + al1.Count);

foreach (int num in al1)

Console.Write(num + "\t");

Console.WriteLine();



Console.WriteLine("Удаляем элемент=1");

al1.Remove(1);

Console.WriteLine("Удаляем элемент c индексом 5");

al1.RemoveAt(5);

Console.WriteLine("Меняем элемент с индексом 5 на значение 100");

al1[5] = 100;

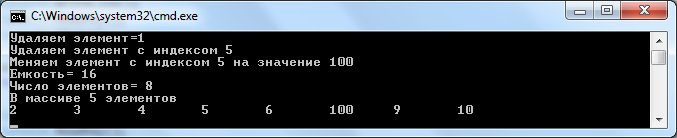
Console.WriteLine("Емкость= " + al1.Capacity);

Console.WriteLine("Число элементов= " + al1.Count);

foreach (int num in al1)

Console.Write(num + "\t");

Console.WriteLine();



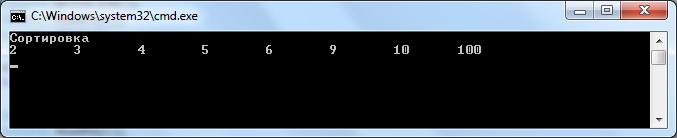
Console.WriteLine("Сортировка");

al1.Sort();

foreach (int num in al1)

Console.Write(num + "\t");

Console.WriteLine();



Console.WriteLine("Бинарный поиск. Ищем число 6");

int number=al1.BinarySearch(6);

if (number >= 0) Console.WriteLine("Нашли! Его номер=" + (number+1));

else Console.WriteLine("Не нашли! ");

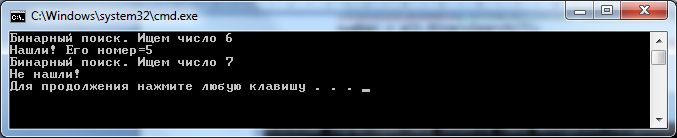
Console.WriteLine("Бинарный поиск. Ищем число 7");

number = al1.BinarySearch(7);

if (number >= 0) Console.WriteLine("Нашли! Его номер=" + (number+1));

else Console.WriteLine("Не нашли! ");

}



### 2.2. Класс Hashtable

**Класс Hashtable** предназначен для создания коллекции, в которой для хранения объектов используется хеш-таблица. Реализует интерфейсы IDictionary, ICollection, IEnumerable, ISerializable, IDeserializationCallback и ICloneable.

**Конструкторы:**

* public Hashtable() - создает стандартный объект класса Hashtable.
* public Hashtable(IDictionary с) – создает коллекцию и инициализирует ее элементами заданной коллекции с.
* public Hashtable(int capacity) – создает коллекцию и инициализирует емкость значением capacity.
* public Hashtable(int capacity, float fillRatio) - инициализирует емкость значением capacity), а коэффициент заполнения значением fillRatio. Значение коэффициента заполнения (коэффициента нагрузки), которое должно попадать в диапазон 0,1-1,0, определяет степень заполнения хеш-таблицы, после чего ее размер увеличивается. В частности, размер таблицы увеличится, когда количество элементов станет больше емкости таблицы, умноженной на ее коэффициент заполнения.

**Основные элементы класса Hashtable**

|  |  |  |
| --- | --- | --- |
| Элемент | Вид | Назначение |
| Keys | Свойство | Получить коллекцию ключей |
| Values | Свойство | Получить коллекцию значений |
| ContainsKey | Метод | Возвращает true , если в вызывающей  коллекции содержится ключ, заданный параметром*.* В противном случае возвращает значение false. |
| ContainsValue | Метод | Возвращает true , если в вызывающей  коллекции содержится значение, заданное параметром*.* В противном случае возвращает значение false. |

**Пример 14\_4.**

static void Main(string[] args)

{

Hashtable ht = new Hashtable();

Console.WriteLine("Основные характеристики объекта типа Hashtable,созданного конструктором без параметров");

Console.WriteLine("Число элементов= " + ht.Count);

Console.WriteLine("Добавляем 6 элементов");

ht.Add("cat", "кошка");

ht.Add(" dog", "собака");

ht.Add("house", "дом");

ht.Add("table", "стол");

ht.Add("program", "программа");

ht.Add("student", "студент");

Console.WriteLine("Число элементов= " + ht.Count);

ICollection c = ht.Keys;

foreach (string s in c)

Console.WriteLine(s +" : "+ ht[s]);

Console.WriteLine("Добавляем 4 элемента");

ht["time"] = "время";

ht["book"]="книга";

ht["timetable"]="расписание";

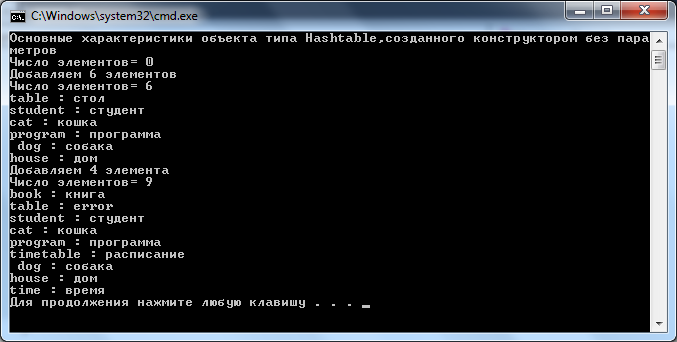
ht["table"]="error";

Console.WriteLine("Число элементов= " + ht.Count);

foreach (string s in c)

Console.WriteLine(s +" : "+ ht[s]);

}



### 2.3. Класс SortedList

**Класс SortedList** предназначен для создания коллекции, которая хранит пары ключ/значение в упорядоченном виде, а именно отсортированы по ключу.

**Конструкторы:**

* public SortedList() - создает пустую коллекцию емкостью 16 элементов.
* public SortedList(IDictionary c) - создает коллекцию, которая инициализируется элементами и емкостью коллекции, заданной параметром с.
* public SortedList(int capacity) *-* создает коллекцию, которая инициализируется емкостью, заданной параметром capacity*.*
* public SortedList(IComparer *comp) -*  создает коллекцию и позволяет задать метод сравнения, который должен использоваться для сравнения объектов коллекции.

Емкость SortedList-коллекции увеличивается автоматически, если в этом возникает необходимость, при добавлении элементов. Если окажется, что текущая емкость может быть превышена, она удваивается.

|  |  |  |
| --- | --- | --- |
| **Элемент** | **Вид** | **Назначение** |
| Keys | Свойство | Получить коллекцию ключей |
| Values | Свойство | Получить коллекцию значений |
| ContainsKey | Метод | Возвращает значение true , если в коллекции содержится ключ, заданный параметром*.* В противном случае возвращает значение false |
| ContainsValue | Метод | Возвращает значение true , если в коллекции содержится значение, заданное параметром*.* В противном случае возвращает значение false |
| GetBylndex | Метод | Возвращает значение, индекс которого задан параметром |
| GetKey | Метод | Возвращает ключ, индекс которого задан параметром |
| GetKeyList() | Метод | Возвращает iList-коллекцию ключей, хранимых в вызывающей SortedList-коллекции |
| GetValueList() | Метод | Возвращает iList-коллекцию значений, хранимых в вызывающей SortedList-коллекции |
| IndexOfKey() | Метод | Возвращает индекс ключа, заданного параметром *к.* Возвращает значение - 1 , если в списке нет заданного ключа |
| IndexOfValue() | Метод | Возвращает индекс первого вхождения значения, заданного параметром *v.* Возвращает -1, если в списке нет заданного ключа |
| SetByIndex(int *idx,* object *v)* | Метод | Устанавливает значение по индексу, заданному параметром ***idx,*** равным значению, переданному в параметре *v* |
| TrimToSize() | Метод | Устанавливает свойство capacity равным значению свойства Count |

**Пример 14\_5.**

static void Main(string[] args)

{

SortedList sl = new SortedList();

Console.WriteLine("Основные характеристики объекта типа SortedList,созданного конструктором без параметров");

Console.WriteLine("Емкость= " + sl.Capacity);

Console.WriteLine("Число элементов= " + sl.Count);

Console.WriteLine("Добавляем 6 элементов");

sl.Add("cat", "кошка");

sl.Add("dog", "собака");

sl.Add("house", "дом");

sl.Add("table", "стол");

sl.Add("program", "программа");

sl.Add("student", "студент");

Console.WriteLine("Число элементов= " + sl.Count);

Console.WriteLine("Емкость= " + sl.Capacity);

ICollection c = sl.Keys;

foreach (string s in c)

Console.WriteLine(s + " : " + sl[s]);

Console.WriteLine("Добавляем 4 элемента");

sl["time"] = "время";

sl["book"] = "книга";

sl["timetable"] = "расписание";

sl["table"] = "error";

Console.WriteLine("Число элементов= " + sl.Count);

Console.WriteLine("Емкость= " + sl.Capacity);

foreach (string s in c)

Console.WriteLine(s + " : " + sl[s]);

Console.WriteLine("Список, полученый с помощью целочисленных индексаторов");

for (int i = 0; i < sl.Count; i++)

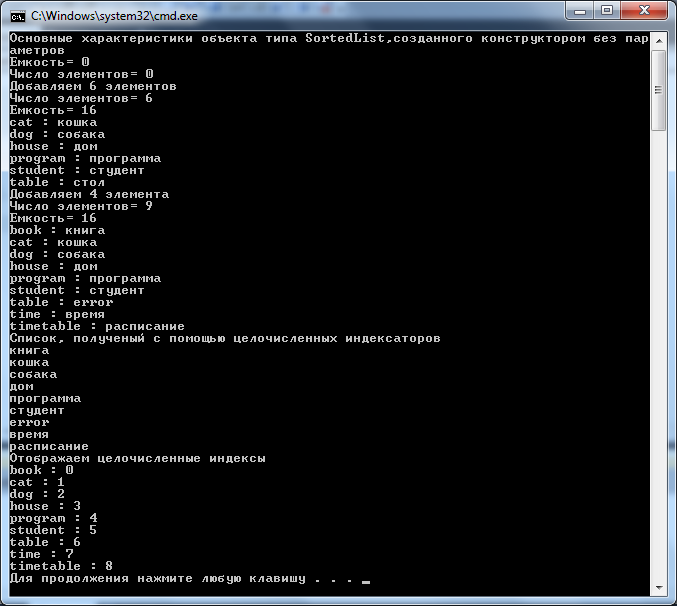
Console.WriteLine(sl.GetByIndex(i) + " ");

Console.WriteLine("Отображаем целочисленные индексы");

foreach (string s in c)

Console.WriteLine(s + " : " + sl.IndexOfKey(s));

}



## 2.4. Класс Stack

**Класс Stack** представляет собой список, добавление и удаление элементов к которому осуществляется по принципу "последним пришел – первым обслужен" (last-in, first-out, LIFO). Реализует интерфейсы ICollection, IEnumerable и ICloneable. Стек – это динамическая коллекция, которая при необходимости увеличивается, чтобы принять для хранения новые элементы, причем каждый раз, когда стек должен расшириться, его емкость удваивается.

**Конструкторы:**

* public Stack() - предназначен для создания пустого стека с начальной емкостью, равной 10 элементам.
* public Stack(int capacity)- создает пустой стек с начальной емкостью, заданной параметром capacity
* public Stack(ICollection c) - служит для построения стека, который инициализируется элементами и емкостью коллекции, заданной параметром с.

|  |  |  |
| --- | --- | --- |
| **Элемент** | **Вид** | **Назначение** |
| Contains(object v) | Метод | Возвращает значение true , если объект *v* содержится в вызывающем стеке. В противном случае возвращает значение false |
| Clear() | Метод | Устанавливает свойство count равным нулю, тем самым эффективно очищая стек |
| Peek() | Метод | Возвращает элемент, расположенный в вершине стека, но не удаляет его |
| Pop() | Метод | Возвращает элемент, расположенный в вершине стека, и удаляет его |
| Push(object *v)* | Метод | Помещает объект *v* в стек |
| ToArray() | Метод | Возвращает массив, который содержит копии элементов вызывающего стека |

**Пример 14\_6.**

class Program

{

static void ShowPush(Stack st, int elem)

{

st.Push(elem);

Console.WriteLine("Помещаем элемент {0} в стек",elem);

Console.WriteLine("Стек:");

foreach(int x in st)

Console.Write(x+" ");

Console.WriteLine();

}

static void ShowPop(Stack st)

{

if (st.Count == 0) Console.WriteLine("Стек пуст");

else

{

int elem = (int)st.Pop();

Console.WriteLine("Удаляем элемент {0} из стекa", elem);

Console.WriteLine("Стек:");

foreach (int x in st)

Console.Write(x + " ");

Console.WriteLine();

}

}

static void Main(string[] args)

{

Stack st = new Stack();

Console.WriteLine("В стеке {0} элементов", st.Count);

Console.WriteLine();

ShowPush(st, 22);

ShowPush(st, 65);

ShowPush(st, 91);

ShowPop(st);

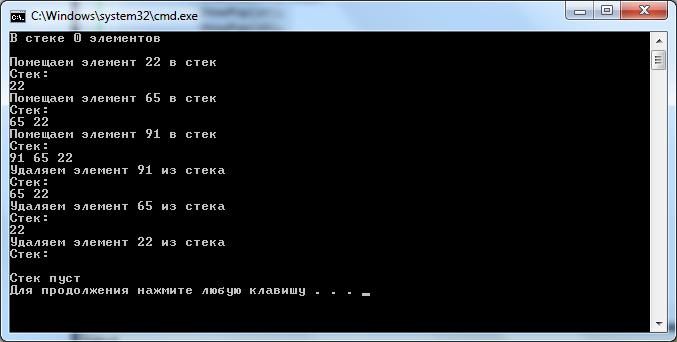
ShowPop(st);

ShowPop(st);

ShowPop(st);

}

}



### 2.5. Класс Queue

Класс коллекции, предназначенный для поддержки **очереди**, называется Queue. Добавление элементов в очередь и удаление их из нее осуществляется по принципу "первым пришел – первым обслужен" (first-in, first-out — FIFO). Он реализует интерфейсы iCollection, IEnumerable и ICloneable. Очередь – это динамическая коллекция, которая при необходимости увеличивается, чтобы принять для хранения новые элементы, причем каждый раз, когда такая необходимость возникает, текущий размер очереди умножается на коэффициент роста, который по умолчанию равен значению 2,0.

В классе Queue определены следующие конструкторы:

* public Queue() - предназначен для создания пустой очереди с начальной емкостью, равной 32 элементам, и коэффициентом роста 2,0.
* public Queue (int *capacity)* - создает пустую очередь с начальной емкостью, заданной параметром *capacity,* и коэффициентом роста 2,0.
* public Queue (int *capacity,* float *growFact)-* позволяет задать коэффициент роста посредством параметра *growFact.*
* public Queue (ICollection c) - служит для создания очереди, которая инициализируется элементами и емкостью коллекции, заданной параметром с.

Помимо методов, определенных в интерфейсах, которые реализует класс Queue, также определены собственные методы.

|  |  |  |
| --- | --- | --- |
| **Элемент** | **Вид** | **Назначение** |
| Contains (object *v)* | Метод | Возвращает значение true , если объект *v* содержится в вызывающей очереди. В противном случае возвращает значение false |
| Clear() | Метод | Устанавливает свойство Count равным нулю, тем самым эффективно очищая очередь |
| Dequeue () | Метод | Возвращает объект из начала вызывающей очереди, Возвращаемый объект из очереди удаляется |
| Enqueue(object *v)* | Метод | Добавляет объект *v* в конец очереди |
| Peek () | Метод | Возвращает объект из начала вызывающей очереди, но не удаляет его |
| тоАггау () | Метод | Возвращает массив, который содержит копии элементов из вызывающей очереди |
| TrimToSize() | Метод | Устанавливает емкость равной значению свойства Count |

**Пример 14\_7**

static void ShowEnq(Queue q, int elem)

{

q.Enqueue(elem);

Console.WriteLine("Помещаем элемент {0} в очередь", elem);

Console.WriteLine("Очередь:");

foreach (int x in q) Console.Write(x + " ");

Console.WriteLine();

}

static void ShowDeq(Queue q)

{

if (q.Count == 0) Console.WriteLine("Очередь пустая");

else

{

int elem = (int)q.Dequeue();

Console.WriteLine("Удаляем элемент {0} из очереди", elem);

Console.WriteLine("Очередь:");

foreach (int x in q) Console.Write(x + " ");

Console.WriteLine();

}

}

static void Main(string[] args)

{

Queue q = new Queue();

Console.WriteLine("В очереди {0} элементов", q.Count);

Console.WriteLine();

ShowEnq(q, 22);

ShowEnq(q, 65);

ShowEnq(q, 91);

ShowDeq(q);

ShowDeq(q);

ShowDeq(q);

ShowDeq(q);

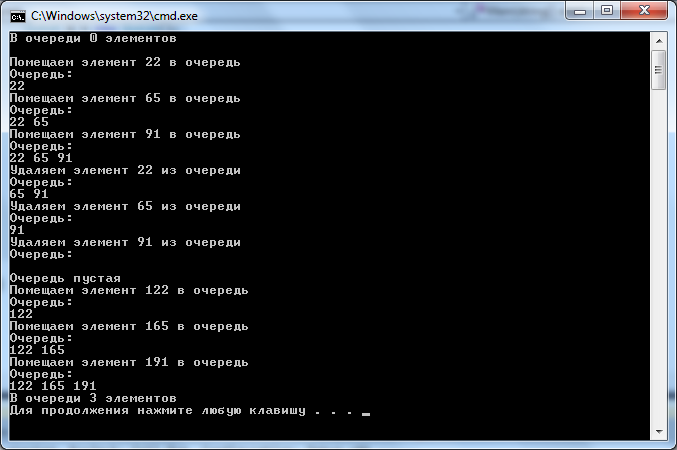
ShowEnq(q, 122);

ShowEnq(q, 165);

ShowEnq(q, 191);

q.TrimToSize();

Console.WriteLine("В очереди {0} элементов", q.Count);

}

### 2.6. Доступ к коллекциям с помощью нумератора

Часто при работе с коллекциями возникает необходимость в циклическом опросе ее элементов. **Нумератор** позволяет выполнять циклический опрос элементов коллекции.

**Нумератор** – это объект, который реализует интерфейс IEnumerator.

**Свойства:**

* Current - позволяет получить элемент, соответствующий текущему значению нумератора.

**Методы:**

* bool MoveNext() - текущая позиция нумератора перемещается к следующему элементу коллекции.
* void Reset() - устанавливает нумератор в начало коллекции.

Для опроса элементов коллекции, в которой хранятся пары ключ/значение используется интерфейс IDictionaryEnumerator.

Класс DictionaryEnumerator является производным от класса IEnumerator и дополнительно определяет три свойства.

* Entry - позволяет получить следующую пару ключ/значение в форме структуры типа DictionaryEntry.
* Key - позволяет получить прямой доступ к ключу.
* Value - позволяет получить прямой доступ к значению.

Интерфейс IDictionaryEnumerator используется подобно обычному нумератору за исключением того, что текущие значения элементов здесь можно получить с помощью свойств Entry, Key или Value.

**Пример 14\_8.**

static void Main(string[] args)

{

ArrayList al = new ArrayList(1);

for (int i = 0; i < 10; i++)

al.Add(i);

IEnumerator ienum = al.GetEnumerator();

while (ienum.MoveNext())

Console.Write(ienum.Current + " ");

Console.WriteLine();

//еще раз

ienum.Reset();

while (ienum.MoveNext())

Console.Write(ienum.Current + " ");

Console.WriteLine();

Hashtable ht = new Hashtable();

ht.Add("Иванов И.И.", "222-22-22");

ht.Add("Сидоров С.С.", "233-33-33");

ht.Add("Петров П.П.", "244-44-44");

IDictionaryEnumerator etr = ht.GetEnumerator();

Console.WriteLine("Отображаем информацию с помощью свойства Entry.");

while (etr.MoveNext())

Console.WriteLine(etr.Entry.Key + ": " + etr.Entry.Value);

Console.WriteLine();

Console.WriteLine("Отображаем информацию с помощью " +"свойств Key и Value.");

etr.Reset();

while (etr.MoveNext())

Console.WriteLine(etr.Key + ": " +etr.Value);

}

