КОЛЛЕКЦИИ

Ранее мы познакомились с такими АТД как списки, деревья и графы, а также рассмотрели их программную реализацию и использование при решении практических задач.

Вместе с тем C# предлагает готовые коллекции, которые представляют собой достаточно мощный и удобный инструмент для хранения и обработки данных. Коллекции упрощают реализацию многих задач программирования, предлагая уже готовые решения для построения структур данных. Все коллекции разработаны на основе четко определенных интерфейсов, поэтому стандартизируют способ обработки групп объектов. Среда .NET Framework поддерживает три основных типа коллекций: общего назначения, специализированные и ориентированные на побитовую организацию данных.

Коллекции общего назначения определены в пространстве имен System.Collection и реализуют такие структуры данных, как стек, очередь, динамический массив, словарь (хеш-таблица, предназначенная для хранения пар ключ/значение), отсортированный список для хранения пар ключ/значение. Коллекции общего назначения работают с данными типа object, поэтому их можно использовать для хранения данных любого типа.

Дополнительно в пространстве имен System.Collection определена единственная к*оллекция BitArray, ориентированная на побитовую организацию данных*, которая служит для хранения групп битов и поддерживает такой набор операций, который не характерен для коллекций других типов.

Коллекции специального назначения определены в пространстве имен System.Collection.Specialized и ориентированы на обработку данных конкретного типа, или на обработку данных уникальным способом. Например, существуют специализированные коллекции, предназначенные только для обработки строк.

Интерфейсы коллекций

В пространстве имен System.Collection, кроме коллекций общего назначения, определено несколько интерфейсов, определяющих функциональность многих стандартных классов в C#, в том числе и коллекций. Поэтому изучение пространства System.Collection мы начнем с изучения интерфейсов.

Интерфейсы, которые поддерживают работу с коллекциями, приведены в следующей таблице:

Интерфейс	**Описание**
IEnumerator	Содержит методы, которые позволяют поэлементно получать содержимое коллекции.
IEnumerable	Определяет коллекцию, к которой можно обратиться с помощью нумератора.
ICollection	Определяет общие характеристики для набора элементов.
IComparer	Определяет метод Compare(), который выполняет сравнение объектов, хранимых в коллекции.
IList	Определяет коллекцию, к которой можно получить доступ посредством индексатора.
IDictionary	Определяет коллекцию (словарь), которая состоит из пар

Интерфейс	Описание
	ключ/значение.
IDictionaryEnumerator	Определяет нумератор для коллекции, которая реализует интерфейс IDictionary.
IHashCodeProvider	Определяет хеш-функцию.

Основополагающим для всех коллекций является реализация *перечислителя (нумератора),* который поддерживается интерфейсами ***IEnumerator и IEnumerable***.

В интерфейсе IEnumerator определено единственное свойство:

```
object Current { get; }
```

И два метода:

```
bool MoveNext()
void Reset()
```

Свойство Current позволяет получить элемент, соответствующий текущему значению нумератора. Поскольку свойство Current предназначено только для чтения, нумератор можно использовать только для считывания значения объекта в коллекции, а не для его модификации.

При каждом обращении к методу MoveNext() текущая позиция нумератора перемещается к следующему элементу коллекции. Метод возвращает значение true, если к следующему элементу можно получить доступ, или значение false, если достигнут конец коллекции. До выполнения первого обращения к методу MoveNext() значение свойства Current не определено.

Установить нумератор в начало коллекции можно с помощью метода Reset(). После вызова метода Reset() нумерация элементов начнется с начала коллекции, и для доступа к первому ее элементу необходимо вызвать метод MoveNext().

Интерфейс IEnumerable должен быть реализован в любом классе, если в нем предполагается поддержка нумераторов. В интерфейсе IEnumerable определен единственный метод GetEnumerator(), который возвращает нумератор для коллекции.

Реализация интерфейсов IEnumerator и IEnumerable позволяет получить доступ к содержимому коллекции, в том числе и с помощью цикла foreach.

Интерфейс ***ICollection*** наследует интерфейс IEnumerable. В ICollection объявлены основные методы и свойства, без которых не может обойтись ни одна коллекция. Рассмотрим некоторые функциональные члены, определенные в интерфейсе ICollection:

Элемент интерфейса	Его тип	описание
int Count {get;}	Свойство	Определяет количество элементов коллекции в данный момент. Если коллекция пуста, то Count равно нулю.
void CopyTo (Array target, int startIdx)	Метод	Обеспечивает переход от коллекции к стандартному C#-массиву, копируя содержимое коллекции в массив, заданный параметром target, начиная с индекса, заданного параметром startIdx.

Так как интерфейс ICollection наследует интерфейс IEnumerable, то он включает в себя его единственный метод GetEnumerator (), который возвращает нумератор коллекции.

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:

Элемент интерфейса	**Его тип**	**Описание**
int Add(object obj)	Метод	Добавляет объект obj в вызывающую коллекцию. Возвращает индекс, по которому этот объект сохранен.
void Clear()	Метод	Удаляет все элементы из коллекции.
bool Contains(object obj)	Метод	Возвращает значение true, если коллекция содержит объект, переданный в параметре obj, и значение false в противном случае.
int IndexOf(object obj)	Метод	Возвращает индекс объекта obj, если он (объект) содержится в коллекции. Если объект obj не обнаружен, метод возвращает -1.
void Insert(int idx, object obj)	Метод	Вставляет в коллекцию объект obj по 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)	Метод	Удаляет первое вхождение объекта obj из коллекции. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы ликвидировать образовавшуюся "брешь".
void RemoveAt(int idx)	Метод	Удаляет из коллекции объект, расположенный по индексу, заданному параметром idx. Элементы, находившиеся до этого за удаленным элементом, смещаются, ликвидируя образовавшуюся "брешь".
bool IsFixedSize { get; }	Свойство	Принимает значение true, если коллекция имеет фиксированный размер. Это означает, что в такую коллекцию нельзя вставлять элементы и удалять их из нее.
bool IsReadOnly { get; }	Свойство	Принимает значение true, если коллекция предназначена только для чтения.
object this[int idx] { get; set; }	Индексатор	Используется для считывания или записи значения элемента с индексом idx. Нельзя применить для добавления в коллекцию нового элемента.

Интерфейс ***IDictionary*** наследует интерфейс ICollection. Он определяет поведение коллекции, которая устанавливает соответствие между уникальными ключами и значениями. Коллекции, реализующий интерфейс IDictionary называют *словарями*. Ключ – это объект, который

ЕРАМ Systems

используется для получения соответствующего ему значения. Сохраненную однажды пару можно затем извлечь по заданному ключу.

Элемент интерфейса	Его тип	Описание
void Add (object k, object v)	метод	Добавляет в словарь пару ключ/значение, заданную параметрами k и v. Ключ k не должен быть нулевым. При попытке задать нулевой ключ генерируют исключение типа NotSupportedException. Если окажется, что ключ k уже хранится в словаре, генерируется исключение типа ArgumentException.
void Clear ()	метод	Удаляет все пары ключ/значение из словаря.
bool Contains (object k)	метод	Возвращает значение true, если словарь содержит объект k в качестве ключа. В противном случае возвращает значение false.
IDictionaryEnumerator GetEnumerator()	метод	Возвращает нумератор словаря.
void Remove (object k)	метод	Удаляет элемент, ключ которого равен значению k.
bool isFixedSize { get }	свойство	Равно значению true, если словарь имеет фиксированный размер.
bool isReadOnly { get }	свойство	Равно значению true, если словарь предназначен только для чтения.
ICollection Keys { get }	свойство	Получает коллекцию ключей.
ICollection Values { get}	свойство	Получает коллекцию значений.
object this[object key] { get; set; }	индексатор	Этот индексатор можно использовать для получения или установки значения элемента, а также, в отличии от коллекций, для добавления в словарь нового элемента. "Индекс" в данном случае является ключом элемента.

Интерфейс ***IDictionaryEnumerator*** является производным от интерфейса IEnumerator и распространяет свои функциональные возможности нумератора на область словарей.

В интерфейсе ***IComparer*** определен метод Compare (), который позволяет сравнивать два объекта:

```
int Compare(object vl, object v2)
```

Метод Compare() возвращает положительное число, если значение vl больше значения v2, отрицательное, если vl меньше v2 и нуль, если сравниваемые значения равны. Этот интерфейс можно использовать для задания способа сортировки элементов коллекции.

После рассмотрения интерфейсов коллекций перейдем к изучению самих коллекций.

Коллекции общего назначения

Классы коллекций общего назначения:

Класс	Описание
Stack	Стек – частный случай однонаправленного списка, действующий по принципу: последним пришел – первым вышел.
Queue	Очередь – частный случай однонаправленного списка, действующего по принципу: первым пришел – первым вышел.
ArrayList	Динамический массив, т.е., массив который при необходимости может увеличивать свой размер.
Hashtable	Хеш-таблица для пар ключ/значение.
SortedList	Отсортированный список пар ключ/значение.

Рассмотрим данные коллекции более подробно.

Коллекция Stack

C# предлагает собственную реализацию АТД стек ввиде класса Stack, который реализует интерфейсы ICollection, IEnumerable и ICloneable (стандартный интерфейс, позволяющий создавать копию объекта).

Stack – это динамическая коллекция, размер которой может изменяться. В классе Stack определены следующие конструкторы:

```
//создает пустой стек, начальная вместимость которого равна 10
public Stack();
// создает пустой стек, начальная вместимость которого равна
// capacity
public Stack(int capacity);
//создает стек, который содержит элементы коллекции,
//заданной параметром c и аналогичной с ней вместимостью
public Stack(ICollection c);
```

Рассмотрим методы, реализуемые классом Stack:

Метод	Описание
public virtual bool Contains(object v)	Возвращает значение true, если объект v содержится в вызывающем стеке; в противном случае возвращает значение false.
public virtual void Clear()	Устанавливает свойство Count равным нулю, тем самым очищая стек.
public virtual object Peek()	Возвращает элемент, расположенный в вершине стека, но не извлекая его из стека.
public virtual object Pop()	Возвращает элемент, расположенный в вершине стека, и извлекает его из стека.
public virtual void Push(object v)	Помещает объект v в вершину стека.

Рассмотрим следующий пример использования стека:

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

EMI Systems

```
namespace Example
{
   class Program
   {
      public static void Main()
      {
         Console.Write("n= ");
         int n = int.Parse(Console.ReadLine());
         Stack intStack = new Stack();
         for (int i = 1; i <= n; i++)
         {
            intStack.Push(i);
         }
         Console.WriteLine("Размерность стека {0}", intStack.Count);
         Console.Write("n= ");
         n = int.Parse(Console.ReadLine());
         for (int i = 1; i <= n; i++)
         {
            intStack.Push(i);
         }
         Console.WriteLine("Размерность стека {0}", intStack.Count);
         Console.WriteLine("Верхний элемент стека = {0}",
                           intStack.Peek());
         Console.WriteLine("Размерность стека {0}", intStack.Count);
         Console.Write("Содержимое стека: ");
         while (intStack.Count != 0)
         {
            Console.Write("{0} ", intStack.Pop());
         }
         Console.WriteLine("\nРазмерность стека {0}",
                           intStack.Count);
      }
   }
}
```

EPAM Systems

Результат работы программы:

```
n=5
Размерность стека 5
n=8
Размерность стека 13
Верхний элемент стека 8
Размерность стека 13
Содержимое стека: 8 7 6 5 4 3 2 1 5 4 3 2 1
Размерность стека 0
```

При работе с коллекциями общего назначения нужно помнить, что они работают с данными типа object, поэтому их можно использовать для хранения данных любого типа. С одной стороны такая универсальность коллекции – это хорошо, а с другой стороны, работа с такими коллекциями это потенциальная возможность ошибок, так как при извлечении данных из коллекции нужно не забывать о преобразовании типов. Например:

```
using System;
using System.Collections;

namespace Example
{
   class Program
   {
      public static void Main()
      {
         Console.Write("n= ");
         int n = int.Parse(Console.ReadLine());
         Stack intStack = new Stack();

         for (int i = 0; i <= n; i++)
         {
            intStack.Push(i);
         }
         int sum = 0;
         while (intStack.Count > 1)
         {
            n = (int)intStack.Pop(); //явное преобразование типов
            sum += n;
            Console.Write("{0}+",n );
         }
         Console.WriteLine("{0}={1}", intStack.Pop(), sum);
      }
   }
}
```

Результат работы программы:

```
n=10
10+9+8+7+6+5+4+3+2+1+0=55
```

Задание

На основе текстового файла создайте стек, который бы содержал каждый символ файла только один раз. Содержимое полученного стека выведите на экран.

Класс Queue

C# предлагает собственную реализацию АТД очередь ввиде класса Queue, который, также как и коллекция Stack, реализует интерфейсы ICollection, IEnumerable и ICloneable.

Queue – это динамическая коллекция, размер которой изменяется. В классе Queue определены следующие конструкторы:

```
public Queue();  //создает пустую очередь, начальная вместимость
                 //которой равна 32
public Queue (int capacity);     //создает пустую очередь, начальная
                                 //вместимость которой равна
                                 //capacity
public Queue (ICollection c);    //создает очередь, которая содержит
                                 //элементы коллекции, заданной
                                 //параметром c, и аналогичной
                                 //с ней вместимостью
```

Рассмотрим методы, реализуемые классом Queue:

Метод	Описание
public virtual bool Contains (object v)	Возвращает значение true, если объект v содержится в вызывающей очереди; в противном случае возвращает значение false.
public virtual void Clear ()	Устанавливает свойство Count равным нулю, тем самым очищая очередь.
public virtual object Dequeue ()	Возвращает объект из начала вызывающей очереди, удаляя его из очереди.
public virtual object Peek ()	Возвращает объект из начала вызывающей очереди, не удаляя его из очереди.
public virtual void Enqueue(object v)	Добавляет объект v в конец очереди.
public virtual void TrimToSizeO	Устанавливает свойство Capacity равным значению свойства Count.

Рассмотрим следующий пример использования очереди:

```
using System;
using System.Collections;
namespace Example
{
   class Program
   {
      public static void Main()
      {
         Console.Write("n= ");
         int n = int.Parse(Console.ReadLine());
         Queue intQ = new Queue();
         for (int i = 1; i <= n; i++)
         {
            intQ.Enqueue(i);
         }
         Console.WriteLine("Размерность очереди {0}", intQ.Count);
         Console.Write("n= ");
         n=int.Parse(Console.ReadLine());
         for (int i = 1; i <= n; i++)
         {
            intQ.Enqueue(i);
         }
         Console.WriteLine("Размерность очереди {0}", intQ.Count);
         Console.WriteLine("Верхний элемент очереди = {0}",
                           intQ.Peek());
         Console.WriteLine("Размерность очереди {0}", intQ.Count);
         Console.Write("n= ");
         n = int.Parse(Console.ReadLine());
         if (intQ.Contains(n))
         {
            Console.WriteLine("Элемент {0} содержится в очереди", n);
```

EPAM Systems

```
            }
            else
            {
               Console.WriteLine("Элемент {0} не содержится в очереди",
                                 n);
            }
            Console.Write("Содержимое очереди: " );
            while (intQ.Count != 0)
               Console.Write("{0}  ", intQ.Dequeue());
            Console.WriteLine("\nРазмерность очереди {0}", intQ.Count);
         }
      }
   }
```

Результат работы программы:

```
n=5
Размерность очереди 5
n=8
Размерность очереди 13
Верхний элемент очереди 1
Размерность очереди 13
n=4
Элемент 4 содержится в очереди
Содержимое очереди: 1 2 3 4 5 1 2 3  4 5 6 7 8
Размерность очереди 0
```

Как мы уже говорили, коллекции общего можно использовать для хранения данных любого типа, в том числе и структур, и ссылок на другие объекты. Для демонстрации сказанного рассмотрим следующий пример:

В текстовом файле записана информация о людях (фамилия, имя, отчество, возраст, вес через пробел). Вывести на экран вначале информацию о людях младше 40 лет, а затем информацию о всех остальных.

```
using System;
using System.Collections;
using System.IO;
using System.Text;

namespace Example
{
   class Program
   {
      //структура для хранения данных об одном человеке
      public struct Person
      {
         public string firstName;
         public string secondName;
         public string lastName;
         public int age;
         public float massa;
         public void Show()
         {
            Console.WriteLine("{0}  {1}  {2}  {3}  {4}", firstName,
```

```
                                secondName, lastName, age, massa);
            }
        }
        public static void Main()
        {
            Queue people = new Queue();
            Console.WriteLine("ВОЗРАСТ МЕНЕЕ 40 ЛЕТ: ");
            Person a;
            using (StreamReader fileIn =
                     new StreamReader(@"d:/Example/input.txt",
                                Encoding.GetEncoding(1251)))
            {
                string line;
                //читаем построчно до конца файла
                while ((line = fileIn.ReadLine()) != null)
                {
                    string[] temp = line.Split(' ');
                    if (temp.Length != 5)
                    {
                        throw new Exception("Ошибка в файле");
                    }
                    else
                    {
                        a.firstName = temp[0];
                        a.secondName = temp[1];
                        a.lastName= temp[2];
                        a.age = int.Parse(temp[3]);
                        a.massa = float.Parse(temp[4]);
                        // если возраст меньше 40 лет, то выводим данные
                        //на экран, иначе помещаем их в
                        //очередь для временного хранения
                        if (a.age < 40)
                        {
                            a.Show();
                        }
                        else
                        {
                            people.Enqueue(a);
                        }
                    }
                }
            }
            Console.WriteLine("ВОЗРАСТ 40 ЛЕТ И СТАРШЕ:");
            while (people.Count != 0)     //извлекаем из очереди данные
            {
                //выполняем преобразование типов
                a = (Person)people.Dequeue();
                a.Show();
            }
        }
    }
}
```

EPAM Systems

Результат работы программы

```
input.txt
Иванов Сергей Николаевич 21 64
Петров Игорь Юрьевич 45 88
Семёнов Михаил Алексеевич 20 70
Пиманов Александр Дмитриевич 53 101
Вывод на экран:
ВОЗРАСТ МЕНЕЕ 40 ЛЕТ:
Иванов Сергей Николаевич 21 64
Семёнов Михаил Алексеевич 20 70
ВОЗРАСТ 40 ЛЕТ И СТАРШЕ:
Петров Игорь Юрьевич 45 88
Пиманов Александр Дмитриевич 53 101
```

Задание

Измените программу так, чтобы для каждой группы людей на экран выводился средний возраст.

Класс ArrayList

В C# стандартные массивы имеют фиксированную длину, которая не может изменяться во время выполнения программы. Класс ArrayList предназначен для поддержки динамических массивов, которые при необходимости могут увеличиваться и сокращаться.

Объект класса ArrayList представляет собой массив переменной длины, элементами которого являются объектные ссылки. Любой объект класса ArrayList создается с некоторым начальным размером. При превышении этого размера память под нее автоматически удваивается. В случае удаления объектов массив можно сократить.

Класс ArrayList реализует интерфейсы ICollection, IList, IEnumerable и ICloneable. В классе ArrayList определены следующие конструкторы:

```
public ArrayList();   //создает пустой массив с максимальной
                      //емкостью равной 16 элементам, при текущем
                      //числе элементов 0
public ArrayList(int capacity);  //создает массив с заданной
                                 //емкостью capacity, при
                                 //текущем числе элементов 0
public ArrayList(ICollection c); //строит массив, который
                                 //инициализируется элементами
                                 //коллекции c
```

Класс ArrayList реализует следующие методы:

Метод	**Описание**
public virtual void AddRange (ICollection c)	Добавляет элементы из коллекции *c* в конец массива.

Метод	Описание
public virtual int BinarySearch (object v)	В вызывающей отсортированной коллекции выполняет поиск значения, заданного параметром v. Возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращается отрицательное значение.
public virtual int BinarySearch (object v, IComparer comp)	В вызывающей отсортированной коллекции выполняет поиск значения, заданного параметром v, на основе метода сравнения объектов, заданного параметром comp. Возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращается отрицательное значение.
public virtual int BinarySearch (int startIdx, int count, object v, IComparer comp)	В вызывающей отсортированной коллекции выполняет поиск значения, заданного параметром v, на основе метода сравнения объектов, заданного параметром comp. Поиск начинается с элемента, индекс которого равен значению startIdx и включает count элементов. Метод возвращает индекс найденного элемента. Если искомое значение не обнаружено, возвращается отрицательное значение.
public virtual void CopyTo(Array ar, int startIdx)	Копирует содержимое вызывающей коллекции начиная с элемента, индекс которого равен значению startIdx, в массив, заданный параметром ar. Приемный массив должен быть одномерным и совместимым по типу с элементами коллекции.
public virtual void CopyTo(int srcIdx, Array ar, int destIdx, int count)	Копирует count элементов вызывающей коллекции, начиная с элемента, индекс которого равен значению srcIdx в массив, заданный параметром ar, начиная с элемента, индекс которого равен значению destIdx. Приемный массив должен быть одномерным и совместимым по типу с элементами коллекции.
public virtual ArrayList GetRange(int idx, int count)	Возвращает часть вызывающей коллекции типа ArrayList. Диапазон возвращаемой коллекции начинается с индекса idx и включает count элементов. Возвращаемый объект ссылается на те же элементы, что и вызывающий объект.
public static ArrayList FixedSize(ArrayList ar)	Превращает коллекцию ar в ArrayList-массив с фиксированным размером и возвращает результат.

ЕРАМ Systems

Метод	Описание
public virtual void InsertRange(int startIdx, ICollection c)	Вставляет элементы коллекции, заданной параметром c, в вызывающую коллекцию, начиная с индекса, заданного параметром startIdx.
public virtual int LastIndexOf(object v)	Возвращает индекс последнего вхождения объекта v в вызывающей коллекции. Если искомый объект не обнаружен, возвращает отрицательное значение.
public static ArrayList ReadOnly(ArrayList ar)	Превращает коллекцию ar в ArrayList-массив, предназначенный только для чтения.
public virtual void RemoveRange(int idx, int count)	Удаляет count элементов из вызывающей коллекции, начиная с элемента, индекс которого равен значению idx.
public virtual void Reverse()	Располагает элементы вызывающей коллекции в обратном порядке.
public virtual void Reverse(int startIdx, int count)	Располагает в обратном порядке count элементов вызывающей коллекции, начиная с индекса startIdx.
public virtual void SetRange(int startIdx, ICollection c)	Заменяет элементы вызывающей коллекции, начиная с индекса startIdx, элементами коллекции, заданной параметром c.
public virtual void Sort()	Сортирует коллекцию по возрастанию.
public virtual void Sort(IComparer comp)	Сортирует вызывающую коллекцию на основе метода сравнения объектов, заданного параметром comp. Если параметр comp имеет значение null, для каждого объекта используется стандартный метод сравнения.
Public virtual void Sort (int startIdx, int endIdx, IComparer comp)	Сортирует часть вызывающей коллекции на основе метода сравнения объектов, заданного параметром comp. Сортировка начинается с индекса startIdx и заканчивается индексом endIdx. Если параметр comp имеет значение null, для каждого объекта используется стандартный метод сравнения.
public virtual object [ ] ToArray ()	Возвращает массив, который содержит копии элементов вызывающего объекта.
public virtual Array ToArray (Type type)	Возвращает массив, который содержит копии элементов вызывающего объекта. Тип элементов в этом массиве задается параметром type.

EPAM Systems

Метод	Описание
public virtual void TrimToSize()	Устанавливает свойство Capacity равным значению свойства Count.

Свойство Capacity позволяет узнать, или установить емкость вызывающего динамического массива типа ArrayList. Емкость представляет собой количество элементов, которые можно сохранить в ArrayList-массиве без его увеличения.

Следует учесть, что при увеличении размера массива фактически происходит его пересоздание, поэтому если вам заранее известно, сколько элементов должно содержаться в ArrayList-массиве, то размерность массива можно установить используя свойство Capacity, экономя тем самым системные ресурсы.

Если нужно уменьшить размер ArrayList-массива, это можно сделать путем установки свойства Capacity. При этом следует помнить, что устанавливаемое значение не должно быть меньше значения свойства Count, иначе будет сгенерировано исключение ArgumentOutOfRangeException. Чтобы сделать емкость ArrayList-массива равной действительному количеству элементов, хранимых в нем в данный момент, установите свойство Capacity равным свойству Count. Того же эффекта можно добиться, вызвав метод TrimToSize().

Рассмотрим следующий пример использования динамического массива.

```
using System;
using System.Collections;
namespace Example
{
   class Program
   {
      static void ArrayPrint(ArrayList a)
      {
         Console.WriteLine("Емкость массива: {0}", a.Capacity);
         Console.WriteLine("Количество элементов: {0}", a.Count);
         Console.Write("Содержимое массива: ");
         foreach (int i in a)
         {
            Console.Write(i + " ");
         }
         Console.WriteLine();
      }
      static void Main()
      {
         ArrayList myArray = new ArrayList();
         ArrayPrint(myArray);
         Console.WriteLine("\nДобавили 5 чисел");
         for (int i = 0; i < 5; i++)
         {
            myArray.Add(i);
         }
         ArrayPrint(myArray);
         Console.WriteLine("\nОптимизируем емкость массива");
         myArray.Capacity = myArray.Count;
```

EPAM Systems

```
            ArrayPrint(myArray);
            Console.WriteLine("\nДобавляем элементы в массив");
            myArray.Add(10);
            myArray.Insert(1, 0);
            myArray.AddRange(myArray);
            ArrayPrint(myArray);
            Console.WriteLine("\nУдаляем элементы из массива");
            myArray.Remove(1);
            myArray.RemoveAt(7);
            ArrayPrint(myArray);
            Console.WriteLine("\nУдаляем весь массив");
            myArray.Clear();
            ArrayPrint(myArray);
        }
    }
}
```

Результат работы программы:

```
Емкость массива: 16
Количество элементов: 0
Содержимое массива:

Добавили 5 чисел
Емкость массива: 16
Количество элементов: 5
Содержимое массива: 0 1 2 3 4

Оптимизируем емкость массива
Емкость массива: 5
Количество элементов: 5
Содержимое массива: 0 1 2 3 4

Добавляем элементы в массив
Емкость массива: 20
Количество элементов: 14
Содержимое массива: 0 0 1 2 3 4 10 0 0 1 2 3 4 10

Удаляем элементы из массива
Емкость массива: 20
Количество элементов: 12
Содержимое массива: 0 0 2 3 4 10 0 1 2 3 4 10

Удаляем весь массив
Емкость массива: 20
Количество элементов: 0
Содержимое массива:
```

Рассмотрим еще один пример: В текстовом файле записана информация о людях (фамилия, имя, отчество, возраст, вес через пробел). Вывести на экран информацию о людях, отсортированную по возрасту.

```
using System;
using System.Collections;
using System.IO;
```

EAM Systems

```
using System.Text;
namespace Example
{
   class Program
   {
      //структура для хранения данных об одном человеке
      public struct Person
      {
         public string firstName;
         public string secondName;
         public string lastName;
         public int age;
         public float massa;
         public void Show()
         {
            Console.WriteLine("{0}  {1}  {2}  {3}  {4}", firstName,
                              secondName, lastName, age, massa);
         }
      }
      //реализация  стандартного интерфейса
      public class SortByAge : IComparer
      {
         //переопределение метода Compare
         int IComparer.Compare(object x, object y)
         {
            Person t1 = (Person)x;
            Person t2 = (Person)y;
            if (t1.age > t2.age) return 1;
            if (t1.age < t2.age) return -1;
            return 0;
         }
      }
      static void ArrayPrint(string s, ArrayList a)
      {
         Console.WriteLine(s);
         foreach (Person item in a)
         {
            item.Show();
         }
      }
      static void Main()
      {
         using (StreamReader fileIn =
                  new StreamReader(@"d:/Example/input.txt",
                           Encoding.GetEncoding(1251)))
         {
            string line;
            Person a;
            ArrayList people = new ArrayList();
            string[] temp = new string[3];
```

EBAM Systems

```
                //цикл для организации обработки файла
                while ((line = fileIn.ReadLine()) != null)
                {
                   temp = line.Split(' ');
                   if (temp.Length != 5)
                   {
                      throw new Exception("Ошибка в файле.");
                   }
                   else
                   {
                      a.firstName = temp[0];
                      a.secondName = temp[1];
                      a.lastName = temp[2];
                      a.age = int.Parse(temp[3]);
                      a.massa = float.Parse(temp[4]);
                      people.Add(a);
                   }
                }
                ArrayPrint("ИСХОДНЫЕ ДАННЫЕ: ", people);
                //вызов сортировки
                people.Sort(new Program.SortByAge());
                ArrayPrint("\nОТСОРТИРОВАННЫЕ ДАННЫЕ: ", people);
             }
          }
       }
    }
```

Результат работы программы:

```
ИСХОДНЫЕ ДАННЫЕ:
Иванов Сергей Николаевич 21 64
Петров Игорь Юрьевич 45 88
Семёнов Михаил Алексеевич 20 70
Пиманов Александр Дмитриевич 53 101

ОТСОРТИРОВАННЫЕ ДАННЫЕ:
Семёнов Михаил Алексеевич 20 70
Иванов Сергей Николаевич 21 64
Петров Игорь Юрьевич 45 88
Пиманов Александр Дмитриевич 53 101
```

Замечание

Обратите внимание на то, что в данном примере был разработан вложенный класс SortByAge, реализующий стандартный интерфейс IComparer. В этом классе был перегружен метод Compare, позволяющий сравнивать между собой два объекта типа Person. Созданный класс использовался для сортировки коллекции по заданному критерию.

Класс Hashtable

Класс Hashtable предназначен для создания коллекции, в которой для хранения объектов используется хеш-таблица. В хеш-таблице для хранения информации используется механизм, именуемый хешированием (hashing). Суть хеширования состоит в том, что для определения уникального значения, которое называется хеш-кодом, используется информационное содержимое соответствующего ему ключа. Хеш-код затем используется в качестве индекса,

по которому в таблице отыскиваются данные, соответствующие этому ключу. Преобразование ключа в хеш-код выполняется автоматически, т.е., сам хеш-код вы даже не увидите.

Преимущество хеширования заключается в том, что оно позволяет сокращать время выполнения таких операций, как поиск, считывание и запись данных, даже для больших объемов информации.

Класс Hashtable реализует стандартные интерфейсы IDictionary, ICollection, IEnumerable, ISerializable, IDeserializationCallback и ICloneable. Размер хеш-таблицы может динамически изменяться. Размер таблицы увеличивается тогда, когда количество элементов превышает значение, равное произведению вместимости таблицы и ее коэффициента заполнения, который может принимать значение на интервале от 0,1 до 1,0. По умолчанию установлен коэффициент равный 1,0.

В классе Hashtable определено несколько конструкторов:

```
public Hashtable()       //создает пустую хеш-таблицу
public Hashtable(IDictionary c)  //строит хеш-таблиц, которая
                                 //инициализируется элементами
                                 //коллекции c
public Hashtable(int capacity)   //создает хеш-таблицу с
                                 //вместимостью capacity
public Hashtable(int capacity, float n)     //создает хеш-таблицу
                                            //вместимостью capacity
                                            //и коэффициентом
                                            //заполнения n
```

Класс Hashtable реализует следующие методы:

Метод	**Описание**
public virtual bool ContainsKey (object k)	Возвращает значение true, если в хеш-таблице содержится ключ, заданный параметром k. В противном случае возвращает значение false.
public virtual bool ContainsValue (object v)	Возвращает значение true, если в хеш-таблице содержится значение, заданное параметром v. В противном случае возвращает значение false.
public virtual IDictionaryEnumerator GetEnumerator()	Возвращает для хеш-таблицы нумератор типа IDictionaryEnumerator.

В классе Hashtable, помимо свойств, определенных в реализованных им интерфейсах, определены два собственных public-свойства:

```
//позволяет получить коллекцию ключей
public virtual ICollection Keys { get; }
//позволяет получить коллекцию значений
public virtual ICollection Values { get; }
```

Для добавления элемента в хеш-таблицу необходимо вызвать метод *Add()*, который принимает два отдельных аргумента: ключ и значение. Важно отметить, что хеш-таблица не гарантирует сохранения порядка элементов, поэтому хеширование не применяется к отсортированным таблицам.

Рассмотрим пример, который демонстрирует использование Hashtable коллекции:

```
using System;
using System.Collections;
namespace MyProgram
{
   class Program
   {
      static void printTab(Hashtable tab)
      {
         Console.WriteLine("Количество элементов: {0}", tab.Count);
         Console.WriteLine("Содержимое таблицы: ");
         ICollection key = tab.Keys;   //прочитали все ключи
         //используем ключи для получения значения
         foreach (string i in key)
         {
            Console.WriteLine("{0} {1}", i, tab[i]);
         }
      }
      static void Main()
      {
         Hashtable tab = new Hashtable();
         printTab(tab);
         Console.WriteLine("\nДобавили данные в таблицу");
         tab.Add("001","ПЕРВЫЙ");
         tab.Add("002","ВТОРОЙ");
         tab.Add("003","ТРЕТИЙ");
         tab.Add("004", "ЧЕТВЕРТЫЙ");
         tab.Add("005", "ПЯТЫЙ");
         printTab(tab);
         Console.WriteLine("\nИзменили данные в таблице");
         tab["005"] = "НОВЫЙ ПЯТЫЙ";
         tab["001"] = "НОВЫЙ ПЕРВЫЙ";
         printTab(tab);
      }
   }
}
```

Результат работы программы:

```
Количество элементов: 0
Содержимое таблицы:

Добавили данные в таблицу
Количество элементов: 5
Содержимое таблицы:
002  ВТОРОЙ
003  ТРЕТИЙ
001  ПЕРВЫЙ
004  ЧЕТВЕРТЫЙ
005  ПЯТЫЙ

Изменили данные в таблицу
Количество элементов: 5
```

EPAM Systems

```
Содержимое таблицы:
002  ВТОРОЙ
003  ТРЕТИЙ
001  НОВЫЙ ПЕРВЫЙ
004  ЧЕТВЕРТЫЙ
005  НОВЫЙ ПЯТЫЙ
```

Задание

Дан текстовый файл. Используя Hashtable, выведите на экран частоту встречаемости каждого символа в тексте.

Как мы уже говорили, коллекции общего назначения могут использоваться для хранения значения любого типа, в том числе ссылочного, например, экземпляра класса. Рассмотрим пример, в котором моделируется простейшая база клиетнов, совершающих покупки. Проект содержит три класса (три файла):

1) класс Client – обеспечивает хранение и обработку данных о каждом клиенте;
2) класс BaseClients – обеспечивает хранение и обработку данных в базе клиентов;
3) класс Program – обеспечивает работу с базой клиентов.

Класс Client

```
using System;
using System.Collections;

namespace Example
{
   public class Client
   {
      public string name;
      public ArrayList buying; //список покупок
      public struct Buying  //информация о каждой покупке
      {
         DateTime data;      //дата покупки
         double sum;       //сумма покупки
         //конструктор структуры
         public Buying (DateTime data, double sum)
         {
            this.data = data;
            this.sum = sum;
         }
         //вывод информации о покупке на экран
         public void Show()
         {
            Console.WriteLine("{0} {1}",
                              data.ToShortDateString(),sum);
         }
      }
      public Client(string name) //конструктор класса
      {
         this.name = name;
         buying = new ArrayList();
      }
```

EDAM Systems

```
        //добавление данных о покупке
        public void AddBuying(DateTime data, double sum)
        {
           Buying item = new Buying(data, sum);
           buying.Add(item);
        }
        public void Show() //вывод информации о клиенте на экран
        {
           Console.WriteLine("Имя: {0}", name);
           Console.WriteLine("Сведенья о покупках: ");
           foreach (Buying item in buying)
           {
              item.Show();
           }
           Console.WriteLine();
        }
     }
  }
```

Класс BaseClients:

```
  using System;
  using System.Collections;
  namespace Example
  {
     public class BaseClients
     {
        uint index;     //номер клиента, генерируется автоматически
        Hashtable clients; //список клиентов
        public BaseClients()  //конструктор класса
        {
           index = 0;
           clients = new Hashtable();
        }
        //добавление нового клиента в хеш-таблицу:
        public void AddClient(string name)
        {  //ключ – index, значение – экземпляр класса Client
           index++;
           clients.Add(index, new Client(name));
        }
        //добавление информации о покупке по номеру клиента
        public void AddBuying(uint index, DateTime data, double sum)
        {
           Client item = (Client)clients[index];
           item.AddBuying(data, sum);
        }
        //добавление информации о покупке по фамилии клиента
        public void AddBuying(string name, DateTime data, double sum)
        {
           ICollection key = clients.Keys;    //прочитали все ключи
           foreach (uint index in key)
```

ЕРАМ Systems

```
            {
               //использовуем  ключ для получения значения хеш-таблицы
               Client item = (Client)clients[index];
               //если фамилия соответсвует фамиили клиента, то мы нашли
               //нужного клиента
               if (string.Compare(name, item.name) == 0)
               {
                  AddBuying(index, data,sum);//и добавляет новую
                                             //покупку по текущему ключу
                  break;
               }
            }
         }
         //удаляем клиента по номеру
         public void DeleteClient(uint index)
         {
            clients.Remove(index);
         }
         //удаляем клиента по фамилии
         public void DeleteClient(string name)
         {
            ICollection key = clients.Keys;
            foreach (uint index in key
            {
               Client item = (Client)clients[index];
               if (string.Compare(name, item.name) == 0)
               {
                  DeleteClient(index);
                  break;
               }
            }
         }
         //выводим данные о базе клиентов на экран
         public void Show()
         {
            ICollection key = clients.Keys; //прочитали все ключи
            foreach (uint index in key)
            {
               //используем ключ для получения информации о клиентах
               Client item = (Client)clients[index];
               Console.WriteLine("№{0}",index); //выводим номер клиета
               item.Show();  //выводим информацию о клиенте на экран
            }
         }
      }
   }
```

Класс Program

```
using System;
namespace Example
{
```

```
class Program
{
   static void Main()
   {
      //инициализируем базу клиентов
      BaseClients bs = new BaseClients();
      //создаем клиента; т.к. он добавляется первым, то его
      //номер - 1
      bs.AddClient("Иванов");
      //вносим данные о покупках первого клиента
      bs.AddBuying(1, new DateTime(2009, 2, 1), 1000);
      bs.AddBuying(1, new DateTime(2009, 2, 1), 2050);
      //создаем нового клиента, его номер автоматически равен 2
      bs.AddClient("Петров");
      //создаем нового клиента, его номер автоматически равен 3
      bs.AddClient("Сидоров");
      //вносим данные о покупках третьего клиента
      bs.AddBuying(3, new DateTime(2009, 2, 3), 1500);
      //вносим данные о покупках Петрова
      bs.AddBuying("Петров", new DateTime(2009, 2, 4), 1700);
      //создаем нового клиента, его номер автоматически равен 3
      bs.AddClient("Пирогов");
      //выводим информацию из базы клиентов
      Console.WriteLine("Исходная база клиентов");
      bs.Show();
      bs.DeleteClient(2);//удаляем клиента с номером 2
      bs.DeleteClient("Пирогов");   //удаляем клиента с фамилией
                                    //Пирогов
      //выводим информацию из базы клиентов
      Console.WriteLine("\nИзмененная база клиентов");
      bs.Show();
   }
}
}
```

Результат работы программы:

```
Исходная база клиентов
№4
Имя: Пирогов
Сведенья о покупках:
№3
Имя: Сидоров
Сведенья о покупках:
03.02.2009 1500
№2
Имя: Петров
Сведенья о покупках:
04.02.2009 1700
№1
Имя: Иванов
Сведенья о покупках:
01.02.2009 1000
```

EPAM Systems

```
01.02.2009 2050

Измененная база клиентов
№3
Имя: Сидоров
Сведенья о покупках:
03.02.2009 1500

№1
Имя: Иванов
Сведенья о покупках:
01.02.2009 1000
01.02.2009 2050
```

Задания

Добавьте в базу клиентов возможность:

1) *получить информацию о клиенте, совершившем покупку на максимальную сумму;*
2) *получить информацию о покупках, совершенных в определенный день.*

Класс SortedList

Класс SortedList предназначен для создания коллекции, которая хранит пары ключ/значение в упорядоченном виде, а именно отсортированы по ключу. Класс SortedList реализует интерфейсы IDictionary, ICollection, IEnumerable и ICloneable.

В классе SortedList определено несколько конструкторов, включая следующие:

```
public SortedList()   //создает пустую коллекцию с начальной
                      //емкостью, равной 16 элементам
public SortedList(IDictionary c) //создает коллекцию, которая
                                 //инициализируется элементами и
                                 //емкостью коллекции с
public SortedList(int capacity)  //создает пустую коллекцию, емкость
                                 //которой равна capacity
public SortedList(IComparer comp)//создает пустую коллекцию с
                                 //начальной емкостью, равной 16
                                 //элементам, и позволяет задать
                                 //метод сравнения для
                                 //упорядочивания элементов списка.
```

Рассмотрим методы, реализованные в классе SortedList:

Метод	**Описание**
public virtual bool ContainsKey(object k)	Возвращает значение true, если в коллекции содержится ключ, заданный параметром k. В противном случае возвращает значение false.
public virtual bool ContainsValue(object v)	Возвращает значение true, если в коллекции содержится значение, заданное параметром v. В противном случае возвращает значение false.
public virtual object GetByIndex(int idx)	Возвращает значение, индекс которого задан параметром idx.

EPAM Systems

Метод	Описание
public virtual IDictionaryEnumerator GetEnumerator()	Возвращает нумератор типа IDictionaryEnumerator для коллекции.
public virtual object GetKey(int idx)	Возвращает ключ, индекс которого задан параметром idx.
public virtual IList GetKeyList()	Возвращает IList-коллекцию ключей, хранимых в коллекции.
public virtual IList GetValueList()	Возвращает IList-коллекцию значений, хранимых в коллекции.
public virtual int IndexOfKey(object k)	Возвращает индекс ключа, заданного параметром k. Если в списке нет заданного ключа, то возвращает -1.
public virtual int IndexOfValue(object v)	Возвращает индекс первого вхождения значения, заданного параметром v. Если в списке нет такого значения, то возвращает -1.
public virtual void SetByIndex(int idx, object v)	Устанавливает значение по индексу, заданному параметром idx, равным значению, переданному в параметре v.
public virtual void TrimToSize()	Устанавливает свойство capacity равным значению свойства Count.

В классе SortedList определены два собственных свойства, позволяющих получить предназначенные только для чтения коллекции ключей и значений, хранимых в SortedList-коллекции:

```
public virtual ICollection Keys { get; }
public virtual ICollection Values { get; }
```

При этом порядок следования ключей и значений в полученных коллекциях отражает порядок SortedList-коллекции.

Рассмотрим пример использования SortedList-коллекции:

```
using System;
using System.Collections;
namespace Expample
{
   class Program
   {
      static void Main()
      {
         //создаем коллекцию и добавляем в нее элементы
         SortedList list = new SortedList();
         list.Add(1, "Петров");
         list.Add(4, "Сидоров");
         list.Add(3, "Иванов");
         list.Add(2, "Грачев");
         ICollection keys = list.Keys; //получаем коллекцию ключей
```

```
            //выводим на экран пары ключ/значение
            foreach (int key in keys)
            {
               Console.WriteLine("{0} {1}", key, list[key]);
            }
         }
      }
   }
```

Результат работы программы:

```
1 Петров
2 Грачев
3 Иванов
4 Сидоров
```

Вернемся к примеру базы клиентов, рассмотренной в разделе Hashtable, и внесем в нее изменения:

1) в классе Client список покупок клиента реализуем через SortedList, сортирующий покупки по увеличению их стоимости;
2) в класс BaseClients добавим номер текущей покупки, который генерируется автоматически.

Класс Client

```
using System;
using System.Collections;
namespace Example
{
   public class Client
   {
      public string name;//имя клиента
      public SortedList buying;  //список покупок
      public struct Buying  //структура «покупка»
      {
         public DateTime data;
         public double sum;
         public Buying (DateTime data, double sum)
         {
            this.data = data;
            this.sum = sum;
         }
         public void Show(uint key) //key- номер текущей покупки
         {
            Console.WriteLine("{0} {1} {2}",key,
                              data.ToShortDateString(),sum);
         }
      }
      //реализация стандартного интерфейса
      public class SortBySum : IComparer
      {
```

ERAM Systems

```
            //переопределяем метод Compare так, чтобы данные
            //сортировались по увеличению стоимости покупки
            int IComparer.Compare(object x, object y)
            {
               Buying t1 = (Buying)x;
               Buying t2 = (Buying)y;
               if (t1.sum > t2.sum) return 1;
               if (t1.sum < t2.sum) return -1;
               return 0;
            }
         }
         public Client(string name) //конструктор класса
         {
            this.name = name;
            //инициализируя список покупок, определяем способ
            //сортировки элементов в экземпляре класса SortedList
            buying = new SortedList(new Client.SortBySum());
         }
         //добавляем покупку в список покупок, при этом в ассоциации
         //ключ/значение ключом будет текущая покупка, а значением - ее
         //номер
         public void AddBuying(uint indexBuying,
                               DateTime data, double sum)
         {
            Buying item = new Buying(data, sum);
            buying.Add(item, indexBuying);
         }

         //вывод информации о клиенте
         public void Show()
         {
            Console.WriteLine("Имя: {0}", name);
            Console.WriteLine("Сведения о покупках: ");
            //получаем коллекцию ключей
            ICollection keys = buying.Keys;
            Console.WriteLine("{0} {1} {2}", "№покупки",
                                    "дата", "сумма");
            foreach (Buying key in keys)  //по ключам
            {
               //извлекаем данные из списка покупок и
               uint index = (uint)buying[key];
               key.Show(index);           //выводим данные о покупке
            }
            Console.WriteLine();
         }
      }
   }
```

Класс BaseClients

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

```
namespace Example
{
   public class BaseClients
   {
      uint indexClient;  //номер клиента, генерируется автоматически
      uint indexBuying;  //номер покупки, генерируется автоматически
      Hashtable clients; //список клиентов
      public BaseClients()  //конструктор класса
      {
         indexClient = 0;
         indexBuying = 0;
         clients = new Hashtable();
      }
      //добавление нового клиента в хеш-таблицу:
      public void AddClient(string name)
      {     //ключ - index, значение - экземпляр класса Client
         indexClient++;
         clients.Add(indexClient, new Client(name));
      }
      //добавление информации о покупке по номеру клиента
      public void AddBuying(uint indexClient, DateTime data,
                                 double sum)
      {
         indexBuying++;
         Client item = (Client)clients[indexClient];
         item.AddBuying(indexBuying, data, sum);
      }
      //добавление информации о покупке по имени клиента
      public void AddBuying(string name, DateTime data, double sum)
      {
         ICollection key = clients.Keys;
         foreach (uint index in key)
         {
            Client item = (Client)clients[index];
            if (string.Compare(name, item.name) == 0)
            {
               AddBuying(index, data,sum);
               break;
            }
         }
      }
      //удаляем клиента по номеру
      public void DeleteClient(uint indexClient)
      {
         clients.Remove(indexClient);
      }
      //удаляем клиента по имени
      public void DeleteClient(string name)
      {
         ICollection key = clients.Keys;
```

LEAN Systems

```
            foreach (uint index in key)
            {
               Client item = (Client)clients[index];
               if (string.Compare(name, item.name) == 0)
               {
                  DeleteClient(index);
                  break;
               }
            }
         }
         //выводим данные о базе клиентов на экран
         public void Show()
         {
            ICollection key = clients.Keys;
            foreach (uint index in key)
            {
               Client item = (Client)clients[index];
               Console.WriteLine("№{0}",index);
               item.Show();
            }
         }
      }
   }
```

Теперь рассмотрим основную программу:

```
using System;
namespace Example
{
   class Program
   {
      static void Main()
      {
         BaseClients bs = new BaseClients();
         bs.AddClient("Иванов");
         bs.AddBuying(1, new DateTime(2009, 2, 1), 1000);
         bs.AddBuying(1, new DateTime(2009, 2, 2), 2050);
         bs.AddBuying(1, new DateTime(2009, 2, 3), 1780);
         bs.AddBuying(1, new DateTime(2009, 2, 4), 340);
         bs.AddClient("Петров");
         bs.AddClient("Сидоров");
         bs.AddBuying(3, new DateTime(2009, 2, 3), 1500);
         bs.AddBuying("Петров", new DateTime(2009, 2, 4), 1700);
         bs.AddBuying("Петров", new DateTime(2009, 2, 5), 1680);
         bs.AddBuying("Петров", new DateTime(2009, 2, 10), 475);
         Console.WriteLine("Исходная база клиентов");
         bs.Show();
      }
   }
}
```

ARAM Systems

Результат работы программы:

```
Исходная база клиентов
№3
Имя: Сидоров
Сведения о покупках:
№покупки дата сумма
5  03.02.2009   1500
№2
Имя: Петров
Сведения о покупках:
№покупки дата сумма
8   10.02.2009  475
7   05.02.2009  1680
6   04.02.2009 1700
№1
Имя: Иванов
Сведения о покупках:
№покупки дата сумма
4    04.02.2009  340
1    01.02.2009 1000
3    03.02.2009  1780
2    02.02.2009 2050
```

Задания

Добавьте в базу клиентов возможность:

1) *получить информацию о клиенте по номеру покупки;*
2) *удалить информацию о покупке по ее номеру.*

Следует отметить, что SortedList, как и Hashtable, является реализацией абстракции «словарь» (IDictionary), но, в отличие от хэш-таблицы, SortedList поддерживает упорядоченность данных. Достигается это за счет хранения ключей и данных в двух отсортированных массивах. Вставка элемента в SortedList производится таким образом, чтобы массив ключей оставался упорядоченным. Для этого осуществляется поиск места вставки, а затем сдвиг элементов массивов, начиная от вставляемого и до конца массива. Так как сдвиг при больших размерах массива (десятки тысяч элементов) становится все медленнее, стоит стараться избегать использования этой коллекции, если требуются частные вставки-удаления. Совершенно не оправдано использование этой коллекции там, где нужен только поиск по ключу. В таких ситуациях намного лучше выбрать хэш-таблицу.

Обзор специализированных коллекций

В среде NET Framework предусмотрена возможность создания специализированных коллекций, которые оптимизированы для работы с конкретными типами данных, или для особого вида обработки. Эти классы коллекций определены в пространстве имен System. Collections.Specialized и перечислены в следующей таблице:

Коллекция	**Описание**
CollectionsUtil	Вспомогательный класс, позволяющий упростить создание коллекций, нечувствительных к регистру при хранении строк.
HybridDictionary	Реализация IDictionary, использующая ListDictionary, если

Коллекция	Описание
	количество элементов в коллекции мало (меньше девяти), и Hashtable, если количество элементов в коллекции превышает 9.
ListDictionary	Реализация IDictionary, использующая однонаправленный связанный список. Рекомендуется для коллекций, в которых заведомо не будет более 10 элементов.
NameObjectCollectionBase	Абстрактный базовый класс для коллекций, позволяющих использовать в качестве ключей строковые значения. В основном методы и свойства этого класса помечены модификатором protected и имеют префикс Base. Это позволяет не выставлять наружу нетипизированные методы и свойства базового класса. Сопоставление (ассоциация) ключей и значений осуществляется с помощью скрытой хэш-таблицы. Слово Object в названии подчеркивает, что в качестве значений в коллекции хранятся ссылки на Object.
NameObjectCollectionBase. KeysCollection	Реализует интерфейс ICollection и индексатор, возвращающий тип данных string. Используется как реализация коллекции ключей в классе NameObjectCollectionBase.
NameValueCollection	Позволяет хранить отсортированный список строк и ассоциированные с ним строковые значения. Коллекцию можно индексировать как с помощью строковых ключей, так и с помощью порядкового индекса. Интересной особенностью коллекции является то, что она позволяет хранить в одном ключе несколько строковых значений, которые конкатенируются перед возвратом пользователю.
StringCollection	Простая реализация коллекции строк. Для хранения списка используется ArrayList. Индексация производится последовательным целочисленным индексом. Коллекция позволяет хранить повторяющиеся значения и null.
StringDictionary	Строго типизированная реализация IDictionary, позволяющая хранить ассоциации строк со строками. В качестве хранилища используется Hashtable.
StringEnumerator	Итератор для класса StringCollection. Также типизированный.
BitVector32	Структура, позволяющая манипулировать с отдельными битами 32-битного числа. Она менее гибка, чем BitArray, но, в отличие от него, позволяет манипулировать отдельными диапазонами битов как миниатюрными целочисленными значениями.
BitVector32.Section	Используется совместно с BitVector32 для определения в последнем секций (наборов битов).

Большинство коллекций из пространства имен System.Collections.Specialized специфичны, и редко используются на практике. Исключение составляют NameObjectCollectionBase, StringCollection и StringDictionary. NameObjectCollectionBase удобна для создания собственных типизированных словарей, а StringCollection и StringDictionary – готовые строковые коллекции.

Задание

Изучите специализированные коллекции самостоятельно.

Обобщенные типы (generics)

Термин *обобщенные типы* (generics) по существу означает параметризованные типы (parameterized types). Параметризованные типы важны потому, что позволяют создавать классы, интерфейсы и методы, в которых тип данных задан как параметр. Используя обобщенные типы можно создать, например, один класс, который автоматически работает с различными типами данных. Класс, интерфейс или метод, который работает на основе параметризированного типа, называется обобщенным, т.е., это может быть обобщенный класс, обобщенный интерфейс или обобщенный метод.

Нужно отметить, что язык C# позволяет создавать классы, интерфейсы и методы, которые могут обрабатывать различные типы данных за счет использования ссылок на тип object. Поскольку object является базовым классом для всех других, то ссылка типа object может ссылаться на объект любого другого типа. Такой прием мы использовали при реализации однонаправленного списка общего вида и дерева двоичного поиска. Однако проблема заключается в том, что использование типа object не обеспечивает типовую безопасность. Например, осуществляя преобразование типов при получении элемента коллекции мы *предполагаем*, что там хранится объект нужного нам типа. Но так это, или нет, мы узнаем только во время исполнения программы.

Generics добавляют недостающую поддержку типовой безопасности, которая позволяет обнаруживать попытки присвоения ссылкам объектов значений некорректных типов уже в момент компиляции, а не во время работы программы. Обобщенные типы также упрощают сам процесс создания обобщенного кода, т.к. их использование устраняет необходимость применения приведения типов для преобразования между object и типом данных, который в действительности используется. Таким образом, обобщенные типы расширяют возможности по многократному использованию кода и позволяют делать это легко и безопасно.

Обобщенный метод

Простой пример использования обобщенных типов мы рассмотрели в разделе «Перегрузка методов». Вернемся к этому примеру еще раз:

```
using System;
// для работы с Generic мы подключаем специальное пространство имен
using System.Collections.Generic;
using System.Text;

namespace Example
{
   class Program
   {
      static void Swap<T>(ref T a, ref T b)
      {
```

```
            Console.WriteLine("Передаем в Swap() метод {0}",
                                        typeof(T));
            T temp;
            temp = a;
            a = b;
            b = temp;
         }
         static void Main(string[] args)
         {
            int a = 1, b = 2;
            Console.WriteLine("Перед Swap: {0} {1}", a, b);
            Swap<int>(ref a, ref b);           //первый вызов метода
            Console.WriteLine("После Swap: {0} {1}", a, b);
            Console.WriteLine();
            double x = 3.2, y = -123.27;
            Console.WriteLine("Перед Swap: {0} {1}", x, y);
            Swap<double>(ref x, ref y);   //второй вызов метода
            Console.WriteLine("После Swap: {0} {1}", x, y);
            Console.ReadLine();
         }
      }
   }
```

Результат работы программы:

```
Перед Swap: 1  2
Передаем в Swap() метод System.Int32
После Swap: 2  1

Перед Swap: 3.2  -123,27
Передаем в Swap() метод System.Double
После Swap: -123,27  3,2
```

Обратите внимание на то, как объявлен обобщенный метод Swap:

```
static void Swap<T>(ref T a, ref T b)
```

Здесь T – это имя параметра типа, используемого в качестве заменителя реального типа, который будет передан в метод Swap при вызове. Имя параметра T в угловых скобках указывает на то, что объявляется обобщенный метод.

Далее параметр T используется внутри Swap тогда, когда это необходимо. В нашем случае мы используем параметр T для описания формальных параметров метода a и b, а также локальной переменной temp.

Замечание

При объявлении метода использование именно имени T необязательно. Вместо него можно использовать любой другой допустимый идентификатор языка, но традиционно для этих целей используется имя T. В качестве имен параметра типа также часто используются имена V и E. Также можно использовать содержательные имена типов, но их принято начинать с буквы T, например, TValue или TKey.

Рассмотрим вызов обобщенного метода:

```
Swap<int>(ref a, ref b);
```

EPAM Systems

Обратите внимание на то, что при вызове метода после указания его имени внутри угловых скобок указан тип int. В данном случае int – это аргумент типа, который передан параметру типа, т.е. T, в метод Swap. Это создает версию Swap, в которой параметр типа T во всех случаях его использования будет заменен типом int. Таким образом, формальные параметры a и b, а также локальная переменная temp будут объявлены типом int.

При втором вызове метода Swap в качестве аргумента типа будет передан тип double и, следовательно, формальные параметры a и b, а также локальная переменная temp будут объявлены типом double. Таким образом, мы можем обращаться к методу Swap с любым типом, в том числе и типом некоторого класса.

Рассмотрим следующий фрагмент программы:

```
double x = 3.2, y = -123.27;
Swap<string>(ref x, ref y);//ошибка
```

В данном случае, в качестве аргумента типа методу Swap будет передан тип string, а в качестве фактических параметров переданы переменные x и y, тип которых double. При попытке скомпилировать приложение сработает защита типовой безопасности и нам будет выдано соответствующее сообщение об ошибке. Таким образом, типовая безопасность позволяет обнаруживать попытки присвоения ссылкам объектов значения некорректных типов уже в момент компиляции, а не во время работы программы.

Обобщенный класс

В разделе «Стек» данного пособия был разработан класс Stack, который основывался на использовании тип object. Это позволяло хранить в экземпляре класса Stack данные любого типа. Модифицируем данный класс, используя обобщенные типы данных.

```
using System;
using System.Collections.Generic;
namespace Example
{
   public class Stack<T>
   {
      //вложенный класс, реализующий элемент стека
      private class Node
      {
         private T inf;
         private Node next;
         public Node(T nodeInfo)
         {
            inf = nodeInfo;
            next = null;
         }
         public Node Next
         {
            get { return next; }
            set { next = value; }
         }
         public T Inf
         {
            get { return inf; }
            set { inf = value; }
         }
```

```
        }       //конец класса Node
        private Node head; //ссылка на вершину стека
        public Stack()  //конструктор класса, создает пустой стек
        {
           head = null;
        }
        // добавляет элемент в вершину стека
        public void Push(T nodeInfo)
        {
           Node r = new Node(nodeInfo);
           r.Next = head;
           head = r;
        }
        //извлекает элемент из вершины стека, если он не пуст
        public T Pop()
        {
           if (head == null)
           {
              throw new Exception("Стек пуст");
           }
           else
           {
              Node r = head;
              head = r.Next;
              return r.Inf;
           }
        }
        public bool IsEmpty//определяет пуст или нет стек
        {
           get
           {
              if (head == null)
              {
                 return true;
              }
              else
              {
                 return false;
              }
           }
        }
     }
  }
```

Обратите внимание на то, как объявлен класс:

```
public class Stack<T>
```

Здесь параметр T играет туже роль, что и при объявлении обобщенного метода. Он является заменителем реального типа, который будет задан при создании объекта класса Stack, и используется для описания идентификаторов внутри класса.

Создать экземпляр обобщенного класса можно следующим образом:

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

Здесь, и при объявлении ссылки на обобщенный класс (Stack <int> list), и при вызове конструктора (new Stack<int>()) в угловых скобках задается аргумент типа, который передается в класс Stack<T>.

Следующие способы создания экземпляра обобщенного класса ***ошибочны***:

```
Stack <int> list = new Stack();
Stack <int> list = new Stack<double>();
```

В первом случае при вызове конструктора не уточнен аргумент типа, во втором случае – аргумент типа не соответствует типу ссылки на экземпляр класса Stack <int>.

Одним из ключевых моментов понимания концепции обобщенных типов является то, что ссылки на объекты обобщенных типов, параметризованных разными типами, несовместимы между собой. Например:

```
Stack<int> iList = new Stack<int>();
Stack<double> dList = new Stack<double>();
iList = dList;  //ошибка!!!
```

Несмотря на то, что iList и dList основаны на Stack<T>, они являются ссылками на различные типы, т.к. их аргументы типов различны.

Теперь рассмотрим следующий фрагмент программы:

```
string line = "Hello!!!";
Stack<char> list = new Stack<char>();
foreach (char i in line)
{
   list.Push(i);
}
while (!list.IsEmpty)
{
   char i = list.Pop();  // извлечение данных из стека
   line += i;
}
```

Когда класс Stack основывался на использовании типа object, то при извлечении данных из стека необходимо было выполнять преобразование типа, а именно:

```
char i = (char)list.Pop();
```

Если вы удалите приведение типа, то программа не скомпилируется, т.к. метод Pop() возвращает значение типа object. Для обобщенного класса Stack<T> такое преобразование выполнять не нужно, т.к. метод Pop() возвращает значение, тип которого соответствует параметру T.

Замечание

Мы рассмотрели класс, в котором задан один аргумент типа. В общем случае синтаксис объявления обобщенного класса может выглядеть следующим образом:

```
class имя_класса <список_параметров_типа>
{…}
```

Соответственно объявление ссылки на обобщенный класс производится следующим образом:

ERAM Systems

```
имя_класса <список_аргументов_типа> имя_переменной =
      new имя_класса <список_аргументов_типа> (…);
```

Создание default-объекта параметра типа

Иногда при использовании параметризованного кода различие между типами значениями и параметрами типа очень важно, например, в ситуации, когда объекту параметра нужно присвоить значение по умолчанию. Для ссылочных типов значение по умолчанию равно null, для размерных типов, за исключением структур, значение по умолчанию равно 0. Значением по умолчанию для структуры является ее объект, все поля которого имеют значения, установленные по умолчанию. Таким образом, если нужно, чтобы переменная параметра типа имела значение по умолчанию, возникает вопрос, какое значение нужно использовать – null, 0, или что-то иное.

Чтобы решить эту проблему следует использовать ключевое слово default. Выражение default(тип) возвращает значение по умолчанию для заданного типа, независимо от того, какой тип используется. Например:

```
class Demo <T>
{
   T ob = default(T);
   …
}
```

Ограничения типов

Для обобщенного класса Stack<T> в качестве параметра T можно было задать любой тип. Хотя для множества случаев какие-либо запреты на определения аргумента типа не требуются, иногда полезно ввести ограничение на типы, которые можно передавать параметру T.

В таких случаях в языке C# используется ограниченные типы (constrained type). При задании параметра типа можно указать ограничение, которому должен удовлетворять данный параметр типа. Для этого используется выражение where, как это показано ниже:

```
class имя_класса <список_параметров_типа>
        where параметр_типа: ограничение
{…}
```

Существует пять видов ограничений:

1) ограничение типа значения (:struct) требует, чтобы аргумент был типом значением (value type constrain);
2) ограничение ссылочного типа (:class) требует, чтобы аргумент был ссылочным типом (reference type constrain);
3) ограничение конструктора (:new()) требует, чтобы аргумент типа поддерживал конструктор без параметров;
4) ограничение базового класса (:имя_базового_класса) требует, чтобы в аргументе типа присутствовал определенный базовый класс (base class constrain);
5) ограничение интерфейса (:имя_интерфейса) требует, чтобы аргумент типа являлся реализацией того или иного интерфейса (interface constrain).

Например, если в реализации класса Stack<T> добавить ограничение ссылочного типа:

ERAM Systems

```
class Stack<T> where T: class
{
   …
}
```

то использовать обобщенный класс Stack<T> для хранения значений размерного типа, например, int, станет невозможным:

```
Stack <int> charList = new Stack<int>(); //ошибка!!!
```

В общем случае с одним параметром может быть связано несколько ограничений. В этом случае ограничения указываются через запятую в виде списка. Например:

```
class Stack<T> where T: class, new(), IComparable
{
   …
}
```

В данном случае параметр T должен быть ссылочного типа, содержать конструктор без параметров и реализовывать интерфейс IComparable.

При использовании двух, или более параметров типа можно задавать ограничение для каждого параметра. Например:

```
class Demo <T, V>
      where T: class
      where V: struct
{ … }
```

В данном случае параметр T должен быть ссылочного типа, а параметр V – размерного типа.

Сравнение экземпляров параметра типа

Иногда бывает нужно сравнить два экземпляра параметров неизвестного типа.

```
using System;
using System.Collections.Generic;
using System.Text;
namespace Example
{
   class Program
   {
      public static bool Equals<T>(T a, T b)
      {
         if (a == b)   //ошибка!!!
         {
            return true;
         }
         else
         {
            return false;
         }
      }
      static void Main()
      {
         int a = 10;
         int b = 10;
```

EPAM Systems

```
            Console.WriteLine(Equals<int>(a, b));
         }
      }
   }
```

К сожалению, этот код не будет работать. Так как T – это обобщенный тип, то компилятор не может точно знать, как нужно проверять равенство заданных объектов. В этом случае, также как и при использовании типа object для ссылки на значение любого типа, нужно использовать метод CompareTo(), определенный в интерфейсе IComparable. При этом необходимо, чтобы тип T реализовывал данный интерфейс.

Модифицируем метод Equals следующим образом:

```
public static bool Equals<T>(T a, T b) where T:IComparable
{
   if (a.CompareTo(b) == 0)
   {
      return true;
   }
   else
   {
      return false;
   }
}
```

Ограничение, наложенное на параметр типа T, гарантирует, что в методе Equals будут использоваться только типы данных, реализующие интерфейс IComparable. Поэтому можно использовать метод CompareTo и корректно сравнить значения переменных a и b.

Задания

Модифицировать реализацию класс List из раздела «Однонаправленные списки общего вида» и реализацию класс BinaryTree из раздела «Деревья бинарного поиска», так чтобы они основывавшись на использовании обобщенных типов данных.

Практикум №17

Замечание

При решении задачи самостоятельно выбрать необходимую коллекцию. Свой выбор обосновать.

1. В текстовом файле записана без ошибок формула вида:

   <формула>=<цифра>|M(<формула>, <формула>)|m(<формула>, <формула>)

   где:

   - | означает логическую операцию или
   - <цифра>=0|1|2|3|4|5|6|7|8|9
   - M обозначает вычисление максимума, m – минимума

   Вычислить значение этой формулы. Например, M(m(3,5),M(1,2))=3
2. В текстовом файле записана без ошибок формула вида:

   <формула>=<цифра>|p(<формула>, <формула>)|m(<формула>, <формула>)

   где:

   - | означает логическую операцию или

- <цифра>=0|1|2|3|4|5|6|7|8|9
- m(a,b)=(a-b) % 10,
- p(a, b) = (a+b) % 10.

Вычислить значение этой формулы. Например, m(9, p(p(3, 5), m(3, 8))) = 6.

3. Пусть символ # определен в текстовом редакторе как стирающий символ Backspace, т.е. строка abc#d##c в действительности является строкой ac.

   Дан текстовый файл, в котором встречается символ #. Преобразовать его с учетом действия этого символа.

4. Дан текстовый файл. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все символы, отличные от цифр, затем все цифры, сохраняя исходный порядок в каждой группе символов.
5. Дан файл, содержащий информацию о сотрудниках фирмы: фамилия, имя, отчество, пол, возраст, размер зарплаты. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о мужчинах, потом все данные о женщинах, сохраняя исходный порядок в каждой группе сотрудников.
6. Дан файл, содержащий информацию о сотрудниках фирмы: фамилия, имя, отчество, пол, возраст, размер зарплаты. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о сотрудниках младше 30 лет, потом данные об остальных сотрудниках, отсортировав данные о сотрудниках в каждой группе по размеру зарплаты.
7. Дан файл, содержащий информацию о студентах: фамилия, имя, отчество, номер группы, оценки по трем предметам текущей сессии. За один просмотр файла напечатать элементы файла в следующем порядке: сначала все данные о студентах, обучающихся на 4 и 5, потом данные об остальных студентах, отсортировав данные о студентах в каждой группе по алфавиту.
8. Реализовать базу вакансий организаций, которая позволяет: добавлять и удалять информацию об огранизациях, добавлять и удалять вакансии в организации; просматривать полную информацию о вакансиях, или информацию о вакансиях в конкретной организации, осуществлять поиск вакансий по заданной специальности.
9. Реализовать каталог музыкальных компакт-дисков, который позволяет: добавлять и удалять диски, добавлять и удалять песни на диске, просматривать содержимое целого каталога и каждого диска в отдельности, осуществлять поиск всех песен заданного исполнителя по всему каталогу.
10. Реализовать библиотечный каталог, который позволяет: добавлять и удалять информацию о библиотечных фондах (периодика, читальный зал, абонемент, редкой литературы), добавлять и удалять записи в каждый фонд, просматривать содержимое целого каталога, или каждого фонда в отдельности, осуществлять поиск всех публикаций заданного автора по всему каталогу.

ERM Systems