Программирование C#

13. Коллекции, перечислители, итераторы

Карбаев Д.С., 2015

Коллекции

- В C# **коллекция** представляет собой совокупность объектов. Коллекции упрощают решение многих задач программирования благодаря тому, что предлагают готовые решения для создания целого ряда типичных, но порой трудоемких для разработки структур данных.
- Например, в среду .NET Framework встроены коллекции, предназначенные для поддержки динамических массивов, связных списков, стеков, очередей и хеш-таблиц.
- Все коллекции разработаны на основе набора четко определенных интерфейсов.
- Поддерживаются пять типов коллекций: необобщенные, специальные, с поразрядной организацией, обобщенные и параллельные.

5 типов коллекций

- **Необобщенные коллекции** (в пространстве имен System. Collections) реализуют ряд основных структур данных, включая динамический массив, стек, очередь.
- **Специальные коллекции** (System. Collections.Specialized) оперируют данными конкретного типа или же делают это каким-то особым образом.
- В прикладном интерфейсе Collections API определена одна **коллекция с поразрядной организацией** — это BitArray (System. Collections).
- **Параллельные коллекции** (System. Collections.Concurrent) поддерживают многопоточный доступ к коллекции.
- В пространстве имен System.Collections .ObjectModel находится также ряд классов, поддерживающих создание пользователями **собственных обобщенных коллекций**.

Перечислители и итераторы

- **Перечислитель** обеспечивает стандартный способ поочередного доступа к элементам коллекции, он перечисляет содержимое коллекции. В каждой коллекции должна быть реализована обобщенная или необобщенная форма интерфейса **IEnumerable**, поэтому элементы любого класса коллекции должны быть доступны посредством методов, определенных в интерфейсе **IEnumerator** или **IEnumerator<T>**.
- **Итератор** упрощает процесс создания классов коллекций, например специальных, поочередное обращение к которым организуется в цикле **foreach**.

Необобщенные коллекции

- **Необобщенные коллекции** представляют собой структуры данных общего назначения, оперирующие ссылками на объекты. Благодаря тому что необобщенные коллекции оперируют ссылками на объекты, в них можно хранить разнотипные данные.

Интерфейсы необобщенных коллекций

Интерфейс	Описание
`ICollection`	Определяет элементы, которые должны иметь все необобщенные коллекции
`IComparer`	Определяет метод `Compare()` для сравнения объектов, хранящихся в коллекции
`IDictionary`	Определяет коллекцию, состоящую из пар "ключ-значение"
`IDictionaryEnumerator`	Определяет перечислитель для коллекции, реализующей интерфейс `IDictionary`
`IEnumerable`	Определяет метод `GetEnumerator()`, предоставляющий перечислитель для любого класса коллекции
`IEnumerator`	Предоставляет методы, позволяющие получать содержимое коллекции по очереди
`IEqualityComparer`	Сравнивает два объекта на предмет равенства
`IHashCodeProvider`	Считается устаревшим. Вместо него следует использовать интерфейс `IEqualityComparer`
`IList`	Определяет коллекцию, доступ к которой можно получить с помощью индексатора
`IStructuralComparable`	Определяет метод `CompareTo()`, применяемый для структурного сравнения
`IStructuralEquatable`	Определяет метод `Equals()`, применяемый для выяснения структурного, а не ссылочного равенства. Кроме того, определяет метод `GetHashCode()`

Интерфейс ICollection

- Интерфейс **ICollection** служит основанием, на котором построены все необобщенные коллекции. В нем объявляются основные методы и свойства для всех необобщенных коллекций. Он также наследует от интерфейса **IEnumerable** метод **GetEnumerator()**.
- В интерфейсе **ICollection** определяется следующий метод

```
void CopyTo(Array target, int startIdx);
```

- копирует содержимое коллекции в массив **target**, начиная с элемента, указываемого по индексу **startIdx**

Свойство	Назначение
`int Count { get; }`	Содержит количество элементов в коллекции на данный момент
`bool IsSynchronized { get; }`	Принимает логическое значение `true`, если коллекция синхронизирована, а иначе — логическое значение `false`. По умолчанию коллекции не синхронизированы. Но для большинства коллекций можно получить синхронизированный вариант
`object SyncRoot { get; }`	Содержит объект, для которого коллекция может быть синхронизирована

Интерфейс IList

- В интерфейсе **IList** объявляется такое поведение необобщенной коллекции, которое позволяет осуществлять доступ к ее элементам по индексу с отсчетом от нуля. Этот интерфейс наследует от интерфейсов **ICollection** и **IEnumerable**.
- В некоторых методах предусматривается модификация коллекции. Если же коллекция доступна только для чтения или имеет фиксированный размер, то в этих методах генерируется исключение **NotSupportedException**.

Методы интерфейса IList

Метод	Описание
`int Add(object value)`	Добавляет объект *value* в вызывающую коллекцию. Возвращает индекс, по которому этот объект сохраняется
`void Clear()`	Удаляет все элементы из вызывающей коллекции
`bool Contains(object value)`	Возвращает логическое значение `true`, если вызывающая коллекция содержит объект *value*, а иначе — логическое значение `false`
`int IndexOf(object value)`	Возвращает индекс объекта *value*, если этот объект содержится в вызывающей коллекции. Если же объект *value* не обнаружен, то метод возвращает значение -1
`void Insert(int index, object value)`	Вставляет в вызывающую коллекцию объект *value* по индексу *index*. Элементы, находившиеся до этого по индексу *index* и дальше, смещаются вперед, чтобы освободить место для вставляемого объекта *value*
`void Remove(object value)`	Удаляет первое вхождение объекта *value* в вызывающей коллекции. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы устранить образовавшийся "пробел"
`void RemoveAt(int index)`	Удаляет из вызывающей коллекции объект, расположенный по указанному индексу *index*. Элементы, находившиеся до этого за удаленным элементом, смещаются назад, чтобы устранить образовавшийся "пробел"

Интерфейс IList

- В интерфейсе IList определяются следующие свойства:

```
bool IsFixedSize { get; }
bool IsReadOnly { get; }
```

- Если коллекция имеет фиксированный размер, то свойство **IsFixedSize** содержит логическое значение true. Это означает, что в такую коллекцию нельзя ни вставлять элементы, ни удалять их из нее. Если же коллекция доступна только для чтения, то свойство **IsReadOnly** содержит логическое значение true. Это означает, что содержимое такой коллекции не подлежит изменению.
- Кроме того, в интерфейсе **IList** определяется следующий индексатор.

```
object this[int index] { get; set; }
```

- Этот индексатор служит для получения и установки значения элемента коллекции. Но его нельзя использовать для добавления в коллекцию нового элемента. С этой целью обычно вызывается метод **Add ()**. Как только элемент будет добавлен в коллекцию, он станет доступным посредством индексатора.

Интерфейс IDictionary

- В интерфейсе **IDictionary** определяется такое поведение необобщенной коллекции, которое позволяет преобразовать уникальные ключи в соответствующие значения. Ключ представляет собой объект, с помощью которого значение извлекается впоследствии.
- В коллекции, реализующей интерфейс **IDictionary**, хранятся пары "ключ-значение". Как только подобная пара будет сохранена, ее можно извлечь с помощью ключа. Интерфейс **IDictionary** наследует от интерфейсов **ICollection** и IEnumerable.
- Некоторые из методов генерируют исключение **ArgumentNullException** при попытке указать пустой ключ, поскольку пустые ключи не допускаются.

Методы и свойства интерфейса IDictionary

Метод	Описание
`void Add(object key, object value)`	Добавляет в вызывающую коллекцию пару "ключ-значение", определяемую параметрами *key* и *value*
`void Clear()`	Удаляет все пары "ключ-значение" из вызывающей коллекции
`bool Contains(object key)`	Возвращает логическое значение `true`, если вызывающая коллекция содержит объект *key* в качестве ключа, в противном случае — логическое значение `false`
`IDictionaryEnumerator GetEnumerator()`	Возвращает перечислитель для вызывающей коллекции
`void Remove(object key)`	Удаляет из коллекции элемент, ключ которого равен значению параметра *key*

Свойство	Назначение
`bool IsFixedSize { get; }`	Принимает логическое значение `true`, если словарь имеет фиксированный размер
`bool IsReadOnly { get; }`	Принимает логическое значение `true`, если словарь доступен только для чтения
`ICollection Keys { get; }`	Получает коллекцию ключей
`ICollection Values { get; }`	Получает коллекцию значений

Кроме того, в интерфейсе `IDictionary` определяется следующий индексатор.

```
object this[object key] { get; set; }
```

Интерфейсы IEnumerable, IEnumerator и IDictionaryEnumerator

- Интерфейс **IEnumerable** является необобщенным, и поэтому он должен быть реализован в классе для поддержки перечислителей.
- Единственный метод **GetEnumerator ()**, определяемый в интерфейсе **IEnumerable**.

```
IEnumerator GetEnumerator()
```

- Он возвращает коллекцию. Благодаря реализации интерфейса **IEnumerable** можно также получать содержимое коллекции в цикле **foreach**.
- В интерфейсе **IEnumerator** определяются функции перечислителя. С помощью методов этого интерфейса можно циклически обращаться к содержимому коллекции. Если в коллекции содержатся пары "ключ-значение" (словари), то метод **GetEnumerator ()** возвращает объект типа **IDictionaryEnumerator**, а не типа **IEnumerator**. Интерфейс **IDictionaryEnumerator** наследует от интерфейса **IEnumerator** и вводит дополнительные функции, упрощающие перечисление словарей.
- В интерфейсе IEnumerator определяются также методы **MoveNext ()** и **Reset ()** и свойство **Current**. Свойство **Current** содержит элемент, получаемый в текущий момент. Метод **MoveNext ()** осуществляет переход к следующему элементу коллекции, а метод **Reset ()** возобновляет перечисление с самого начала.

Интерфейсы IComparer и IEqualityComparer

- В интерфейсе **IComparer** определяется метод **Compare ()** для сравнения двух объектов.

```
int Compare(object x, object y);
```

- Он возвращает положительное значение, если значение объекта x больше, чем у объекта y; отрицательное — если значение объекта x меньше, чем у объекта y; и нулевое — если сравниваемые значения равны.
- Данный интерфейс можно использовать для указания способа сортировки элементов коллекции.
- В интерфейсе **IEqualityComparer** определяются два метода.

```
bool Equals(object x, object y);
int GetHashCode(object obj);
```

- Метод **Equals ()** возвращает логическое значение **true**, если значения объектов x и y равны. А метод **GetHashCode ()** возвращает хеш-код для объекта obj.

Интерфейсы IStructuralComparable и IStructuralEquatable

- В интерфейсе **IStructuralComparable** определяется метод **CompareTo ()**, который задает способ структурного сравнения двух объектов для целей сортировки. (Иными словами, метод **CompareTo()** сравнивает содержимое объектов, а не ссылки на них.) Ниже приведена форма объявления данного метода.

```
int CompareTo(object other, IComparer comparer);
```

- Он должен возвращать -1, если вызывающий объект предшествует другому объекту **other**; 1, если вызывающий объект следует после объекта **other**; и 0, если значения обоих объектов одинаковы для целей сортировки. А само сравнение обеспечивает объект, передаваемый через параметр **comparer**.

Интерфейсы IStructuralComparable и IStructuralEquatable

- Интерфейс **IStructuralEquatable** служит для выяснения структурного равенства путем сравнения содержимого двух объектов. В этом интерфейсе определены следующие методы.

```
bool Equals(object other, IEqualityComparer comparer);
int GetHashCode(IEqualityComparer comparer);
```

- Метод **Equals()** должен возвращать логическое значение **true**, если вызывающий объект и другой объект **other** равны. А метод **GetHashCode()** должен возвращать хеш-код для вызывающего объекта. Само сравнение обеспечивает объект, передаваемый через параметр **comparer**.

Структура DictionaryEntry

- В пространстве имен System. Collections определена структура DictionaryEntry.
- Необобщенные коллекции пар "ключ-значение" сохраняют эти пары в объекте типа DictionaryEntry. В данной структуре определяются два следующих свойства.

```
public object Key { get; set; }
public object Value { get; set; }
```

- Эти свойства служат для доступа к ключу или значению, связанному с элементом коллекции. Объект типа DictionaryEntry может быть сконструирован с помощью конструктора:

```
public DictionaryEntry(object key, object value);
```

где **key** обозначает ключ, а **value** — значение.

Классы необобщенных коллекций

Класс	Описание
`ArrayList`	Определяет динамический массив, т.е. такой массив, который может при необходимости увеличивать свой размер
`Hashtable`	Определяет хеш-таблицу для пар "ключ-значение"
`Queue`	Определяет очередь, или список, действующий по принципу "первым пришел — первым обслужен"
`SortedList`	Определяет отсортированный список пар "ключ-значение"
`Stack`	Определяет стек, или список, действующий по принципу "первым пришел — последним обслужен"

Класс Array List

- В классе **ArrayList** поддерживаются динамические массивы, расширяющиеся и сокращающиеся по мере необходимости.
- В классе **ArrayList** определяется массив переменной длины, который состоит из ссылок на объекты и может динамически увеличивать и уменьшать свой размер. Массив типа **ArrayList** создается с первоначальным размером.
- В классе ArrayList реализуются интерфейсы ICollection, IList, IEnumerable и ICloneable. Ниже приведены конструкторы класса ArrayList.

```
public ArrayList();
public ArrayList(ICollection c);
public ArrayList(int capacity);
```

Методы класса Array List

Метод	Описание
`public virtual void AddRange(Icollection c)`	Добавляет элементы из коллекции *c* в конец вызывающей коллекции типа `ArrayList`
`public virtual int BinarySearch(object value)`	Выполняет поиск в вызывающей коллекции значения *value*. Возвращает индекс найденного элемента. Если искомое значение не найдено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован
`public virtual int BinarySearch(object value, Icomparer comparer)`	Выполняет поиск в вызывающей коллекции значения *value*, используя для сравнения способ, определяемый параметром *comparer*. Возвращает индекс совпавшего элемента. Если искомое значение не найдено, возвращает отрицательное значение. Вызывающий список должен быть отсортирован
`public virtual int BinarySearch(int index, int count, object value, IComparer comparer)`	Выполняет поиск в вызывающей коллекции значения *value*, используя для сравнения способ, определяемый параметром *comparer*. Поиск начинается с элемента, указываемого по индексу *index*, и включает количество элементов, определяемых параметром *count*. Метод возвращает индекс совпавшего элемента. Если искомое значение не найдено, метод возвращает отрицательное значение. Вызывающий список должен быть отсортирован
`public virtual void CopyTo(Array array)`	Копирует содержимое вызывающей коллекции в массив *array*, который должен быть одномерным и совместимым по типу с элементами коллекции
`public virtual void CopyTo(Array array, int arrayIndex)`	Копирует содержимое вызывающей коллекции в массив *array*, начиная с элемента, указываемого по индексу *arrayIndex*. Целевой массив должен быть одномерным и совместимым по типу с элементами коллекции

Методы класса Array List

`public virtual void CopyTo(int index, Array array, int arrayIndex, int count)`	Копирует часть вызывающей коллекции, начиная с элемента, указываемого по индексу *index*, и включая количество элементов, определяемых параметром *count*, в массив *array*, начиная с элемента, указываемого по индексу *arrayIndex*. Целевой массив должен быть одномерным и совместимым по типу с элементами коллекции
`public static ArrayList FixedSize(ArrayList list)`	Заключает коллекцию *list* в оболочку типа ArrayList с фиксированным размером и возвращает результат
`public virtual ArrayList GetRange(int index, int count)`	Возвращает часть вызывающей коллекции типа ArrayList. Часть возвращаемой коллекции начинается с элемента, указываемого по индексу *index*, и включает количество элементов, определяемое параметром *count*. Возвращаемый объект ссылается на те же элементы, что и вызывающий объект
`public virtual int IndexOf(object value)`	Возвращает индекс первого вхождения объекта *value* в вызывающей коллекции. Если искомый объект не обнаружен, возвращает значение –1
`public virtual void InsertRange(int index, ICollection c)`	Вставляет элементы коллекции *c* в вызывающую коллекцию, начиная с элемента, указываемого по индексу *index*
`public virtual int LastIndexOf(object value)`	Возвращает индекс последнего вхождения объекта *value* в вызывающей коллекции. Если искомый объект не обнаружен, метод возвращает значение –1

Методы класса Array List

Метод	Описание
`public static ArrayList ReadOnly(ArrayList list)`	Заключает коллекцию *list* в оболочку типа ArrayList, доступную только для чтения, и возвращает результат
`public virtual void RemoveRange(int index, int count)`	Удаляет часть вызывающей коллекции, начиная с элемента, указываемого по индексу *index*, и включая количество элементов, определяемое параметром *count*
`public virtual void Reverse()`	Располагает элементы вызывающей коллекции в обратном порядке
`public virtual void Reverse(int index, int count)`	Располагает в обратном порядке часть вызывающей коллекции, начиная с элемента, указываемого по индексу *index*, и включая количество элементов, определяемое параметром *count*
`public virtual void SetRange(int index, ICollection c)`	Заменяет часть вызывающей коллекции, начиная с элемента, указываемого по индексу index, элементами коллекции c
`public virtual void Sort()`	Сортирует вызывающую коллекцию по нарастающей
`public virtual void Sort(Icomparer comparer)`	Сортирует вызывающую коллекцию, используя для сравнения способ, определяемый параметром *comparer*. Если параметр *comparer* имеет пустое значение, то для сравнения используется способ, выбираемый по умолчанию

Методы класса Array List

`public virtual void Sort(int index, int count, Icomparer comparer)`	Сортирует вызывающую коллекцию, используя для сравнения способ, определяемый параметром *comparer*. Сортировка начинается с элемента, указываемого по индексу *index*, и включает количество элементов, определяемых параметром *count*. Если параметр comparer имеет пустое значение, то для сравнения используется способ, выбираемый по умолчанию
`public static ArrayList Synchronized(ArrayList list)`	Возвращает синхронизированный вариант коллекции типа ArrayList, передаваемой в качестве параметра *list*
`public virtual object[] ToArray()`	Возвращает массив, содержащий копии элементов вызывающего объекта
`public virtual Array ToArray(Type type)`	Возвращает массив, содержащий копии элементов вызывающего объекта. Тип элементов этого массива определяется параметром *type*
`public virtual void TrimToSize()`	Устанавливает значение свойства Capacity равным значению свойства Count

Коллекция Array List

- По умолчанию коллекция типа **ArrayList** не синхронизирована. Для получения синхронизированной оболочки, в которую заключается коллекция, вызывается метод `Synchronized()`.
- В классе **ArrayList** имеется свойство **Capacity**, помимо свойств, определенных в интерфейсах, которые в нем реализуются.

```
public virtual int Capacity { get; set; }
```

- Емкость обозначает количество элементов, которые может содержать коллекция типа **ArrayList** до ее вынужденного расширения.
- Всякая попытка установить значение свойства **Capacity** меньше значения свойства **Count** приводит к генерированию исключения **ArgumentOutOfRangeException**. Поэтому для получения такого количества элементов коллекции типа **ArrayList**, которое содержится в ней на данный момент, следует установить значение свойства **Capacity** равным значению свойства **Count**. Для этой цели можно также вызвать метод **TrimToSize()**.

```
//применение класса ArrayList.
using System; using System.Collections;
class ArrayListDemo {
    static void Main(){
        ArrayList a = new ArrayList();//коллекция в виде динамического массива
        Console.WriteLine("Исходное количество элементов: " + a.Count);
        Console.WriteLine("Добавить 6 элементов");
        // Добавить элементы в динамический массив
        a.Add('C'); a.Add('A'); a.Add('E'); a.Add('B'); a.Add('D'); a.Add('F');
        Console.WriteLine("Количество элементов: " + a.Count);
        // Отобразить содержимое динамического массива, используя индексирование
        Console.Write("Текущее содержимое: ");
        for (int i = 0; i < a.Count; i++) Console.Write(a[i] + " ");
        Console.WriteLine("Удалить 2 элемента"); a.Remove('F'); a.Remove('A');
        Console.WriteLine("Количество элементов: " + a.Count);
        Console.Write("Содержимое: ");
        foreach (char c in a) Console.Write(c + " ");
        Console.WriteLine("Добавить еще 20 элементов");
        // Добавить элементы для принудительного расширения массива
        for (int i = 0; i < 20; i++) a.Add((char)('a' + i));
        Console.WriteLine("Текущая емкость: " + a.Capacity);
        Console.WriteLine("Количество элементов после добавления 20 новых: " +
        a.Count);
        Console.Write("Содержимое: ");
        foreach (char c in a) Console.Write(c + " ");
        // Изменить содержимое динамического массива, используя индексирование
        Console.WriteLine("Изменить три первых элемента");
        a[0] = 'X'; a[1]= 'Y'; a[2] = 'Z';
        Console.Write("Содержимое: ");
        foreach (char c in a) Console.Write(c + " ");
    }}
```

Результат выполнения

```
Исходное количество элементов: 0
Добавить 6 элементов
Количество элементов: 6
Текущее содержимое: C A E B D F
Удалить 2 элемента
Количество элементов: 4
Содержимое: C E B D
Добавить еще 20 элементов
Текущая емкость: 32
Количество элементов после добавления 20 новых: 24
Содержимое: CEBDabcdefghij klmnopqrst
Изменить три первых элемента
Содержимое: XYZDabcdefghij klmnopqrst
```

```
// Отсортировать коллекцию типа ArrayList и осуществить в ней поиск
using System;
using System.Collections;
class SortSearchDemo {
static void Main() {
// Создать коллекцию в виде динамического массива
ArrayList a = new ArrayList();
// Добавить элементы в динамический массив
a.Add(55); a.Add(43); a.Add(-4);
a.Add(88); a.Add(3); a.Add(19);
Console.Write("Исходное содержимое: ");
foreach(int i in a) Console.Write(i + " ");
// Отсортировать динамический массив
a.Sort();
// Отобразить содержимое динамического массива, используя цикл foreach
Console.Write("Содержимое после сортировки: ");
foreach(int i in a) Console.Write(i + " ");
Console.WriteLine("Индекс элемента 43: " +
a.BinarySearch (43));
}
}
```

Исходное содержимое: 55 43 -4 88 3 19
Содержимое после сортировки: -4 3 19 43 55 88
Индекс элемента 43: 3

```
// Преобразовать коллекцию типа ArrayList в обычный массив
using System;
using System.Collections;
class ArrayListToArray{
    static void Main(){
        ArrayList a = new ArrayList();
        // Добавить элементы в динамический массив
        a.Add(1); a.Add(2); a.Add(3); a.Add(4);
        Console.Write("Содержимое: ");
        foreach (int i in a) Console.Write(i + " ");
        // Получить массив
        int[] ia = (int[])a.ToArray(typeof(int));
        int sum = 0;
        // Просуммировать элементы массива
        for (int i = 0; i < ia.Length; i++) sum += ia[i];
        Console.WriteLine("Сумма равна: " + sum);
    }}
```

Содержимое: 1 2 3 4
Сумма равна: 10

Класс Hashtable

- Класс **Hashtable** предназначен для создания коллекции, в которой для хранения ее элементов служит хеш-таблица. При хешировании для определения уникального значения, называемого хеш-кодом, используется информационное содержимое специального ключа. Полученный в итоге хеш-код служит в качестве индекса, по которому в таблице хранятся искомые данные, соответствующие заданному ключу.
- Преобразование ключа в хеш-код выполняется автоматически, и поэтому сам хеш-код вообще недоступен пользователю.
- В классе **Hashtable** реализуются интерфейсы **IDictionary**, **ICollection, IEnumerable, ISerializable, IDeserializationCallback** и **ICloneable**.
- В классе **Hashtable** определено немало конструкторов. Ниже приведены наиболее часто используемые конструкторы этого класса.

```
public Hashtable();
public Hashtable(IDictionary d);
public Hashtable(int capacity);
public Hashtable(int capacity, float loadFactor);
```

loadFactor - коэффициент заполнения, иногда еще называемый коэффициентом загрузки, должен находиться в пределах от 0,1 до 1,0.

Методы класса Hashtable

Метод	Описание
`public virtual bool ContainsKey(object key)`	Возвращает логическое значение `true`, если в вызывающей коллекции типа `Hashtable` содержится ключ *`key`*, а иначе — логическое значение `false`
`public virtual bool ContainsValue(object value)`	Возвращает логическое значение `true`, если в вызывающей коллекции типа `Hashtable` содержится значение *`value`*, а иначе — логическое значение `false`
`public virtual IDictionaryEnumerator GetEnumerator()`	Возвращает для вызывающей коллекции типа `Hashtable` перечислитель типа `IDictionaryEnumerator`
`public static Hashtable Synchronized(Hashtable table)`	Возвращает синхронизированный вариант коллекции типа `Hashtable`, передаваемой в качестве параметра *`table`*

Класс Hashtable

- В классе **Hashtable** доступны также открытые свойства, определенные в тех интерфейсах, которые в нем реализуются. Эти свойства определяются в интерфейсе **IDictionary** следующим образом.

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

- В классе **Hashtable** не поддерживаются упорядоченные коллекции, и поэтому ключи или значения получаются из коллекции в произвольном порядке. Кроме того, в классе **Hashtable** имеется защищенное свойство **EqualityComparer**.
- Пары "ключ-значение" сохраняются в коллекции типа **Hashtable** в форме структуры типа **DictionaryEntry**.
- Если, например, в коллекцию типа **Hashtable** добавляется элемент, то для этой цели вызывается метод **Add()**, принимающий два аргумента: ключ и значение.

Класс Hashtable

```
// Продемонстрировать применение класса Hashtable
using System;
using System.Collections;
class HashtableDemo {
static void Main() {
// Создать хеш-таблицу.
Hashtable ht = new Hashtable();
// Добавить элементы в таблицу
ht.Add ("здание", "жилое помещение");
ht.Add("автомашина", "транспортное средство");
ht.Add("книга", "набор печатных слов");
ht.Add("яблоко", "съедобный плод");
// Добавить элементы с помощью индексатора
ht["трактор"] = "сельскохозяйственная машина";
// Получить коллекцию ключей
ICollection c = ht.Keys;
// Использовать ключи для получения значений
foreach(string str in c)
Console.WriteLine(str + ": " + ht[str]);
} }
```

здание: жилое помещение
книга: набор печатных слов
трактор: сельскохозяйственная машина
автомашина: транспортное средство
яблоко: съедобный плод

Класс SortedList

- Класс **SortedList** предназначен для создания коллекции, в которой пары "ключ-значение" хранятся в порядке, отсортированном по значению ключей. В классе **SortedList** реализуются интерфейсы **IDictionary, ICollection, IEnumerable** и **ICloneable**.
- В классе **SortedList** определено несколько конструкторов, включая следующие.

```
public SortedList();
public SortedList(IDictionary d);
public SortedList(int initialCapacity);
public SortedList(IComparer comparer);
```

- В первом конструкторе создается пустая коллекция, первоначальная емкость которой равна нулю.
- Во втором конструкторе создается пустая коллекция типа **SortedList**, которая инициализируется элементами из коллекции d.
- В третьем конструкторе создается **SortedList**, первоначальный размер которой определяет емкость, задаваемая параметром **initialCapacity**.
- И в четвертой форме конструктора с помощью параметра **comparer** указывается способ, используемый для сравнения объектов по списку. В этой форме создается пустая коллекция, первоначальная емкость которой равна нулю.

Метод	Описание
`public virtual bool ContainsKey(object key)`	Возвращает логическое значение `true`, если в вызывающей коллекции типа `SortedList` содержится ключ *`key`*, а иначе — логическое значение `false`
`public virtual bool ContainsValue(object value)`	Возвращает логическое значение `true`, если в вызывающей коллекции типа `SortedList` содержится значение *`value`*, а иначе — логическое значение `false`
`public virtual object GetByIndex(int index)`	Возвращает значение, указываемое по индексу *`index`*
`public virtual IDictionaryEnumerator GetEnumerator()`	Возвращает для вызывающей коллекции типа `SortedList` перечислитель типа `IDictionaryEnumerator`
`public virtual object GetKey(int index)`	Возвращает значение ключа, указываемое по индексу *`index`*
`public virtual IList GetKeyList()`	Возвращает коллекцию типа `SortedList` с ключами, хранящимися в вызывающей коллекции типа `SortedList`
`public virtual IList GetValueList()`	Возвращает коллекцию типа `SortedList` со значениями, хранящимися в вызывающей коллекции типа `SortedList`
`public virtual int IndexOfKey(object key)`	Возвращает индекс ключа *`key`*. Если искомый ключ не обнаружен, возвращается значение `-1`
`public virtual int IndexOfValue(object value)`	Возвращает индекс первого вхождения значения *`value`* в вызывающей коллекции. Если искомое значение не обнаружено, возвращается значение `-1`
`public virtual void SetByIndex(int index, object value)`	Устанавливает значение по индексу *`index`* равным значению *`value`*
`public static SortedList Synchronized(SortedList list)`	Возвращает синхронизированный вариант коллекции типа `SortedList`, передаваемой в качестве параметра *`list`*
`public virtual void TrimToSize()`	Устанавливает значение свойства `Capacity` равным значению свойства `Count`

Класс SortedList

- В классе **SortedList** доступны также открытые свойства, определенные в тех интерфейсах, которые в нем реализуются. Эти свойства определяются в интерфейсе **IDictionary** следующим образом.

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

- Порядок следования ключей и значений отражает порядок их расположения в коллекции типа **SortedList**. Аналогично коллекции типа **Hashtable**, пары "ключ-значение" сохраняются в коллекции типа **SortedList** в форме структуры типа **DictionaryEntry**, но, как правило, доступ к ключам и значениям осуществляется по отдельности с помощью методов и свойств, определенных в классе **SortedList**.

```
   //применение SortedList
   static void Main() {
   // Создать отсортированный список
   SortedList s = new SortedList();
   // Добавить элементы в список
   s.Add("здание", "жилое помещение");
   s.Add("автомашина", "транспортное средство");
   s.Add("книга", "набор печатных слов");
   s.Add("яблоко", "съедобный плод");
   // Добавить элементы с помощью индексатора
   s["трактор"] = "сельскохозяйственная машина";
   // Получить коллекцию ключей
   ICollection c = s.Keys;
   // Использовать ключи для получения значений
   Console.WriteLine("Содержимое списка по
индексатору.");
   foreach(string str in c)
    Console.WriteLine(str + ": " + s[str]);
   // Отобразить список
   Console.WriteLine("Содержимое списка по
целочисленным индексам.");
   for(int i=0; i < s.Count; i++)
    Console.WriteLine(s.GetByIndex(i));
   // Показать целочисленные индексы элементов списка
   Console.WriteLine("Целочисленные индексы элементов
списка.");
   foreach (string str in c)
       Console.WriteLine(str+": "+s.IndexOfKey(str));
   }}
```

Содержимое списка по индексатору.
автомашина: транспортное средство
здание: жилое помещение
книга: набор печатных слов
трактор: сельскохозяйственная
Машина
яблоко: съедобный плод

Содержимое списка по
целочисленным индексам.
транспортное средство
жилое помещение
набор печатных слов
сельскохозяйственная машина
съедобный плод

Целочисленные индексы элементов
списка.
автомашина: О
здание: 1
книга: 2
трактор: 3
яблоко: 4

Класс Stack

- Класс коллекции, поддерживающий стек, носит название **Stack**. В нем реализуются интерфейсы **ICollection, IEnumerable и ICloneable**. Этот класс создает динамическую коллекцию, которая расширяется по мере потребности хранить в ней вводимые элементы. Всякий раз, когда требуется расширить такую коллекцию, ее емкость увеличивается вдвое.
- В классе **Stack** определяются следующие конструкторы.

```
public Stack();
public Stack(int initialCapacity);
public Stack(ICollection col);
```

- В первой форме конструктора создается пустой стек, во второй форме — пустой стек, первоначальный размер которого определяет первоначальная емкость, задаваемая параметром **initialCapacity**, и в третьей форме — стек, содержащий элементы указываемой коллекции **col**. Его первоначальная емкость равна количеству указанных элементов.

Класс Stack

Метод	Описание
`public virtual void Clear()`	Устанавливает свойство `Count` равным нулю, очищая, по существу, стек
`public virtual bool Contains(object obj)`	Возвращает логическое значение `true`, если объект *`obj`* содержится в вызывающем стеке, а иначе — логическое значение `false`
`public virtual object Peek()`	Возвращает элемент, находящийся на вершине стека, но не удаляет его
`public virtual object Pop()`	Возвращает элемент, находящийся на вершине стека, удаляя его по ходу дела
`public virtual void Push(object obj)`	Помещает объект *`obj`* в стек
`public static Stack Synchronized(Stack stack)`	Возвращает синхронизированный вариант коллекции типа `Stack`, передаваемой в качестве параметра *`stack`*
`public virtual object[] ToArray()`	Возвращает массив, содержащий копии элементов вызывающего стека

```
//применение класса Stack
static void ShowPush(Stack st, int a) {
 st.Push(a);
 Console.WriteLine("Поместить в стек: Push(" + a + ")");
 Console.Write("Содержимое стека: ");
 foreach(int i in st) Console.Write(i + " ");
}
static void ShowPop(Stack st) {
 Console.Write("Извлечь из стека: Pop -> ");
 int a = (int) st.Pop();
 Console.WriteLine(a) ;
 Console.Write("Содержимое стека: ");
 foreach(int i in st) Console.Write(i + " ");
}
static void Main() {
 Stack st = new Stack();
 foreach(int i in st) Console.Write(i + " ");
 Console.WriteLine();
 ShowPush(st, 22);
 ShowPush(st, 65);
 ShowPush(st, 91);
 ShowPop(st) ;
 ShowPop(st);
 ShowPop(st);
try {
 ShowPop(st);
} catch (InvalidOperationException){
  Console.WriteLine("Стек пуст.");
}}}
```

```
Поместить в стек: Push(22)
Содержимое стека: 22
Поместить в стек: Push(65)
Содержимое стека: 65 22
Поместить в стек: Push (91)
Содержимое стека: 91 65 22
Извлечь из стека: Pop -> 91
Содержимое стека: 65 22
Извлечь из стека: Pop -> 65
Содержимое стека: 22
Извлечь из стека: Pop -> 22
Содержимое стека:
Извлечь из стека: Pop -> Стек пуст.
```

Класс Queue

- Класс коллекции, поддерживающий очередь, носит название **Queue**. В нем реализуются интерфейсы **ICollection, IEnumerable** и **ICloneable**. Если в очереди требуется свободное место, ее размер увеличивается на коэффициент роста, который по умолчанию равен 2,0.
- В классе **Queue** определяются приведенные ниже конструкторы.

```
public Queue();
public Queue (int capacity);
public Queue (int capacity, float growFactor);
public Queue (ICollection col);
```

- В первой форме конструктора создается пустая очередь.
- Во второй форме создается пустая очередь, первоначальный размер которой определяет емкость, задаваемая параметром **capacity**.
- В третьей форме допускается указывать не только емкость (параметр **capacity**), но и коэффициент роста создаваемой очереди (параметр **growFactor** в пределах от 1,0 до 10,0).
- В четвертой форме создается очередь, состоящая из элементов указываемой коллекции **col**. Ее первоначальная емкость равна количеству указанных элементов, а коэффициент роста по умолчанию выбирается для нее равным 2,0.

Класс Queue

Метод	Описание
`public virtual void Clear()`	Устанавливает свойство `Count` равным нулю, очищая, по существу, очередь
`public virtual bool Contains(object obj)`	Возвращает логическое значение `true`, если объект `obj` содержится в вызывающей очереди, а иначе — логическое значение `false`
`public virtual object Dequeue()`	Возвращает объект из начала вызывающей очереди. Возвращаемый объект удаляется из очереди
`public virtual void Enqueue(object obj)`	Добавляет объект `obj` в конец очереди
`public virtual object Peek()`	Возвращает объект из начала вызывающей очереди, но не удаляет его
`public static Queue Synchronized(Queue queue)`	Возвращает синхронизированный вариант коллекции типа `Queue`, передаваемой в качестве параметра *queue*
`public virtual object[] ToArray()`	Возвращает массив, который содержит копии элементов из вызывающей очереди
`public virtual void TrimToSize()`	Устанавливает значение свойства `Capacity` равным значению свойства `Count`

```
// применение класса Queue
static void ShowEnq(Queue q, int a) {
 q.Enqueue(a);
 Console.WriteLine("Поместить в очередь: Enqueue(" + a + ")");
 Console.Write("Содержимое очереди: ");
 foreach(int i in q) Console.Write(i + " ");
 Console.WriteLine ();
}
static void ShowDeq(Queue q) {
 Console.Write("Извлечь из очереди: Dequeue -> ");
 int a = (int) q.Dequeue();
 Console.WriteLine (a);
 Console.Write("Содержимое очереди: ");
 foreach(int i in q) Console.Write(i + " ");
 Console.WriteLine();
}
static void Main() {
 Queue q = new Queue();
 foreach(int i in q) Console.Write(i + " ");
 Console.WriteLine();
 ShowEnq(q, 22);
 ShowEnq(q, 65);
 ShowEnq(q, 91);
 ShowDeq(q);
 ShowDeq(q);
 ShowDeq(q);
try {
 ShowDeq(q);
} catch (InvalidOperationException) {
   Console.WriteLine("Очередь пуста.");
}}
```

```
Поместить в очередь: Enqueue(22)
Содержимое очереди: 22
Поместить в очередь: Enqueue(65)
Содержимое очереди: 22 65
Поместить в очередь: Enqueue(91)
Содержимое очереди: 22 65 91
Извлечь из очереди: Dequeue -> 22
Содержимое очереди: 65 91
Извлечь из очереди: Dequeue -> 65
Содержимое очереди: 91
Извлечь из очереди: Dequeue -> 91
Содержимое очереди:
Извлечь из очереди: Dequeue ->
Очередь пуста.
```

Коллекция BitArray

- Класс **BitArray** служит для хранения отдельных битов в коллекции. В классе **BitArray** реализуются интерфейсы **ICollection** и **IEnumerable** как основополагающие элементы поддержки всех типов коллекций. Кроме того, в классе **BitArray** реализуется интерфейс **ICloneable**.
- В классе **BitArray** определено несколько конструкторов. Можно сконструировать объект типа **BitArray** из массива логических значений.

```
public BitArray(bool[] values)
```

каждый элемент массива **values** становится отдельным битом в коллекции.

- Коллекцию типа **BitArray** можно также составить из массива байтов.

```
public BitArray(byte[] bytes)
```

- Здесь битами в коллекции становится уже целый их набор из массива **bytes**, причем элемент **bytes[0]** обозначает первые 8 битов, элемент **bytes [1]** — вторые 8 битов и т.д.

Коллекция BitArray

- Аналогично, коллекцию типа **BitArray** можно составить из массива целочисленных значений.

```
public BitArray(int[] values);
```

элемент **values [0]** обозначает первые 32 бита, элемент **values [1]** — вторые 32 бита и т.д.

- С помощью следующего конструктора можно составить коллекцию типа **BitArray**, указав ее конкретный размер:

```
public BitArray(int length);
```

где **length** обозначает количество битов в коллекции, которые инициализируются логическим значением **false**.

- Можно указать не только размер коллекции, но и первоначальное значение составляющих ее битов.

```
public BitArray(int length, bool defaultValue);
```

все биты в коллекции инициализируются значением **defaultValue**, передаваемым конструктору в качестве параметра.

- Новую коллекцию типа BitArray можно создать из уже существующей.

```
public BitArray(BitArray bits);
```

Вновь сконструированный объект будет содержать такое же количество битов, как и в указываемой коллекции **bits**, а в остальном это будут две совершенно разные коллекции.

Коллекция BitArray

Метод	Описание
`public BitArray And(BitArray value)`	Выполняет операцию логического умножения И битов вызывающего объекта и коллекции *value*. Возвращает коллекцию типа `BitArray`, содержащую результат
`public bool Get(int index)`	Возвращает значение бита, указываемого по индексу *index*
`public BitArray Not()`	Выполняет операцию поразрядного логического отрицания НЕ битов вызывающей коллекции и возвращает коллекцию типа `BitArray`, содержащую результат
`public BitArray Or(BitArray value)`	Выполняет операцию логического сложения ИЛИ битов вызывающего объекта и коллекции *value*. Возвращает коллекцию типа `BitArray`, содержащую результат
`public void Set(int index, bool value)`	Устанавливает бит, указываемый по индексу *index*, равным значению *value*
`public void SetAll(bool value)`	Устанавливает все биты равными значению *value*
`public BitArray Xor(BitArray value)`	Выполняет логическую операцию исключающее ИЛИ над битами вызывающего объекта и коллекции *value*. Возвращает коллекцию типа `BitArray`, содержащую результат

Коллекция BitArray

- В классе **BitArray** определяется также собственное свойство, помимо тех, что указаны в интерфейсах, которые в нем реализуются.

```
public int Length { get; set; }
```

- Свойство **Length** позволяет установить или получить количество битов в коллекции. В отличие от свойства **Count**, свойство **Length** доступно не только для чтения, но и для записи, а значит, с его помощью можно изменить размер коллекции типа **BitArray**. Так, при сокращении коллекции типа **BitArray** лишние биты усекаются, начиная со старшего разряда. А при расширении коллекции типа **BitArray** дополнительные биты, имеющие логическое значение **false**, вводятся в коллекцию, начиная с того же старшего разряда.
- Кроме того, в классе **BitArray** определяется следующий индексатор.

```
public bool this[int index] { get; set; }
```

- С помощью этого индексатора можно получать или устанавливать значение элемента.

```
 //применение класса BitArray
public static void ShowBits(string rem, BitArray bits) {
 Console.WriteLine(rem);
for(int i=0; i < bits.Count; i++) Console.Write("{0, -6} ", bits[i]);
}
static void Main() {
 BitArray ba = new BitArray(8); byte[] b = { 67 };
 BitArray ba2 = new BitArray(b);
 ShowBits("Исходное содержимое коллекции ba:", ba) ;
 ba = ba.Not() ;
 ShowBits("Содержимое коллекции ba после логической операции NOT:", ba);
 ShowBits("Содержимое коллекции ba2:", ba2);
 BitArray ba3 = ba.Xor(ba2);
 ShowBits("Результат логической операции ba XOR ba2:", ba3);
}
```

Исходное содержимое коллекции ba:
False False False False False False False False
Содержимое коллекции ba после логической операции NOT:
True True True True True True True True
Содержимое коллекции ba2:
True True False False False False True False
Результат логической операции ba XOR ba2:
False False True True True True False True

Специальные коллекции

Класс специальной коллекции	Описание
`CollectionsUtil`	Содержит фабричные методы для создания коллекций
`HybridDictionary`	Предназначен для коллекций, в которых для хранения небольшого количества пар "ключ-значение" используется класс `ListDictionary`. При превышении коллекцией определенного размера автоматически используется класс `Hashtable` для хранения ее элементов
`ListDictionary`	Предназначен для коллекций, в которых для хранения пар "ключ-значение" используется связный список. Такие коллекции рекомендуются только для хранения небольшого количества элементов
`NameValueCollection`	Предназначен для отсортированных коллекций, в которых хранятся пары "ключ-значение", причем и ключ, и значение относятся к типу `string`
`OrderedDictionary`	Предназначен для коллекций, в которых хранятся индексируемые пары "ключ-значение"
`StringCollection`	Предназначен для коллекций, оптимизированных для хранения символьных строк
`StringDictionary`	Предназначен для хеш-таблиц, в которых хранятся пары "ключ-значение", причем и ключ, и значение относятся к типу `string`

Обобщенные коллекции

- Обобщенные коллекции объявляются в пространстве имен **System.Collections. Generic;**
- Как правило, классы обобщенных коллекций являются не более чем обобщенными эквивалентами классов необобщенных коллекций, хотя это соответствие не является взаимно однозначным. Например, в классе обобщенной коллекции **LinkedList** реализуется двунаправленный список, тогда как в необобщенном эквиваленте его не существует.
- В некоторых случаях одни и те же функции существуют параллельно в классах обобщенных и необобщенных коллекций, хотя и под разными именами. Так, обобщенный вариант класса **ArrayList** называется **List**, а обобщенный вариант класса **HashTable** — **Dictionary**.
- Конкретное содержимое различных интерфейсов и классов реорганизуется с минимальными изменениями для переноса некоторых функций из одного интерфейса в другой.

Интерфейсы обобщенных коллекций

Интерфейс	Описание
`ICollection<T>`	Определяет основополагающие свойства обобщенных коллекций
`IComparer<T>`	Определяет обобщенный метод `Compare()` для сравнения объектов, хранящихся в коллекции
`IDictionary<Tkey, TValue>`	Определяет обобщенную коллекцию, состоящую из пар "ключ-значение"
`IEnumerable<T>`	Определяет обобщенный метод `GetEnumerator()`, предоставляющий перечислитель для любого класса коллекции
`Enumerator<T>`	Предоставляет методы, позволяющие получать содержимое коллекции по очереди
`IEqualityComparer<T>`	Сравнивает два объекта на предмет равенства
`IList<T>`	Определяет обобщенную коллекцию, доступ к которой можно получить с помощью индексатора

Доступ к коллекции с помощью перечислителя

- **Перечислитель** — это объект, который реализует необобщенный интерфейс **IEnumerator** или обобщенный интерфейс **IEnumerator<T>**.
- В интерфейсе IEnumerator определяется свойство.

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

- А в интерфейсе **IEnumerator<T>** объявляется следующая обобщенная форма свойства **Current**.

```
T Current { get; }
```

- В обеих формах свойства **Current** получается текущий перечисляемый элемент коллекции.
- В интерфейсе IEnumerator определяются два метода. Первым из них является метод

```
bool MoveNext();
```

- При каждом вызове метода **MoveNext ()** текущее положение перечислителя смещается к следующему элементу коллекции. Этот метод возвращает логическое значение **true**, если следующий элемент коллекции доступен, и логическое значение **false**, если достигнут конец коллекции.

Доступ к коллекции с помощью перечислителя

- Для установки перечислителя в исходное положение, соответствующее началу коллекции, вызывается приведенный ниже метод **Reset ()**.

```
void Reset();
```

- После вызова метода **Reset ()** перечисление вновь начинается с самого начала коллекции. Поэтому, прежде чем получить первый элемент коллекции, следует вызвать метод **MoveNext ()**.
- В каждом классе коллекции предоставляется метод **GetEnumerator ()**, возвращающий перечислитель в начало коллекции. Используя этот перечислитель, можно получить доступ к любому элементу коллекции по очереди.
- Для циклического обращения к содержимому коллекции с помощью перечислителя рекомендуется придерживаться приведенной ниже процедуры.

1. Получить перечислитель, устанавливаемый в начало коллекции, вызвав для этой коллекции метод **GetEnumerator ()**.
2. Организовать цикл, в котором вызывается метод **MoveNext ()**. Повторять цикл до тех пор, пока метод **MoveNext ()** возвращает логическое значение **true**.
3. Получить в цикле каждый элемент коллекции с помощью свойства **Current**.

Доступ к коллекции с помощью перечислителя

```
 //применение перечислителя
using System;
using System.Collections;
class EnumeratorDemo {
static void Main() {
  ArrayList list = new ArrayList(1);
  for(int i=0; i < 10; i++) list.Add(i);
  // Использовать перечислитель для доступа к списку
  IEnumerator etr = list.GetEnumerator();
  while(etr.MoveNext()) Console.Write(etr.Current + " ") ;
  Console.WriteLine();
  // Повторить перечисление списка
  etr.Reset() ;
  while(etr.MoveNext()) Console.Write(etr.Current + " ");
  Console.WriteLine();
}}
```

```
0123456789
0123456789
```

Применение перечислителя типа IDictionaryEnumerator

- Если для организации коллекции в виде словаря, например типа **Hashtable**, реализуется необобщенный интерфейс **IDictionary**, то для циклического обращения к элементам такой коллекции следует использовать перечислитель типа **IDictionaryEnumerator** вместо перечислителя типа **IEnumerator**.
- Интерфейс **IDictionaryEnumerator** наследует от интерфейса **IEnumerator** и имеет три дополнительных свойства.

```
DictionaryEntry Entry { get; }
```

Свойство **Entry** позволяет получить пару "ключ-значение" из перечислителя в форме структуры **DictionaryEntry**.

- Два других свойства, определяемых в интерфейсе **IDictionaryEnumerator**.

```
object Key { get; }
object Value { get; }
```

- С помощью этих свойств осуществляется непосредственный доступ к ключу или значению.

Применение перечислителя типа IDictionaryEnumerator

- Перечислитель типа **IDictionaryEnumerator** используется аналогично обычному перечислителю, за исключением того, что текущее значение в данном случае получается с помощью свойств **Entry**, **Key** или **Value**, а не свойства **Current**.
- Следовательно, приобретя перечислитель типа **IDictionaryEnumerator**, необходимо вызвать метод **MoveNext ()**, чтобы получить первый элемент коллекции.
- А для получения остальных ее элементов следует продолжить вызовы метода **MoveNext ()**. Этот метод возвращает логическое значение **false**, когда в коллекции больше нет ни одного элемента.

Применение перечислителя типа IDictionaryEnumerator

```
   // применение перечислителя IDictionaryEnumerator
   static void Main() {
   // Создать хеш-таблицу
   Hashtable ht = new Hashtable();
   // Добавить элементы в таблицу
   ht.Add("Кен", "555-7756");
   ht.Add("Мэри", "555-9876");
   ht.Add("Том", "555-3456") ;
   ht.Add("Тодд", "555-3452");
   // Продемонстрировать применение перечислителя
   IDictionaryEnumerator etr = ht.GetEnumerator();
   Console.WriteLine("Отобразить информацию с помощью
свойства Entry.");
   while(etF.MoveNext())
     Console.WriteLine(etr.Entry.Key + ": " +
          etr.Entry.Value);
   Console.WriteLine ();
   Console.WriteLine("Отобразить информацию " +
          "с помощью свойств Key и Value.");
   etr.Reset() ;
   while(etr.MoveNext())
     Console.WriteLine(etr.Key + ": " + etr.Value);
   }}
```

Отобразить информацию с помощью свойства Entry.
Мэри: 555-9876
Том: 555-3456
Тодд: 555-3452
Кен: 555-7756 .

Отобразить информацию с помощью свойств Key и Value.
Мэри: 555-9876
Том: 555-3456
Тодд: 555-3452
Кен: 555-7756

```
// Реализовать интерфейсы IEnumerable и IEnumerator
using System;
using System.Collections;
class MyClass : IEnumerator, IEnumerable {
char[] chrs = { 'A' , 'B', 'C' , 'D' };
int idx = -1;
// Реализовать интерфейс IEnumerable
public IEnumerator GetEnumerator() { return this; }
// В следующих методах реализуется интерфейс IEnumerator
// Возвратить текущий объект
public object Current { get { return chrs[idx]; }}
// Перейти к следующему объекту
public bool MoveNext() {
  if(idx == chrs.Length-1) {
    Reset(); //установить перечислитель в конец
    return false;
  }
  idx++;
  return true;
}
// Установить перечислитель в начало
public void Reset() { idx = -1; }}
class EnumeratorImplDemo {
static void Main() {
  MyClass mc = new MyClass ();
  // Отобразить содержимое объекта mc
  foreach(char ch in mc) Console.Write(ch + " ");
  // Вновь отобразить содержимое объекта mc
  foreach(char ch in mc) Console.Write(ch + " ");
}}
```

```
A B C D
A B C D
```

```
// Простой пример применения итератора
using System;
using System.Collections;
class MyClass {
  char[] chrs = { 'A', 'B', 'C', 'D' };
  // Этот итератор возвращает символы из массива chrs
  public IEnumerator GetEnumerator() {
    foreach(char ch in chrs) yield return ch;
} }

class ItrDemo {
  static void Main() {
    MyClass mc = new MyClass ();
     foreach(char ch in mc) Console.Write(ch + " ");
    Console.WriteLine();
} }
```

A B C D

```
// Пример динамического построения значений,
// возвращаемых по очереди с помощью итератора
using System;
using System.Collections;
class MyClass {
  char ch = 'A';
  // Этот итератор возвращает буквы английского
  // алфавита, набранные в верхнем регистре
  public IEnumerator GetEnumerator() {
    for (int i = 0; i < 26; i++)
      yield return (char)(ch + i);
  }
}

class ItrDemo2 {
  static void Main() {
    MyClass mc = new MyClass();
    foreach(char ch in mc) Console.Write(ch + " ") ;
    Console.WriteLine();
} }
```

ABCDEFGHIJKLMNOPQRSTUVWXYZ

```
// Пример прерывания итератора
using System;
using System.Collections;
class MyClass {
  char ch = 'A';
  // Этот итератор возвращает первые 10 букв английского алфавита
  public IEnumerator GetEnumerator() {
    for(int i=0; i < 26; i++) {
      if(i == 10) yield break; // прервать итератор преждевременно
      yield return (char) (ch + i);
} } }

class ItrDemo3 {
  static void Main() {
    MyClass mc = new MyClass();
    foreach(char ch in mc) Console.Write(ch + " ");
    Console.WriteLine();
} }
```

ABCDEFGHIJ

```
// Пример применения нескольких операторов yield
using System;
using System.Collections;
class MyClass {
  // Этот итератор возвращает буквы A, B, C, D и E
  public IEnumerator GetEnumerator() {
    yield return 'A';
    yield return 'B';
    yield return 'C' ;
    yield return 'D';
    yield return 'E';
  }
}

class ItrDemo5 {
  static void Main() {
    MyClass mc = new MyClass ();
    foreach(char ch in mc) Console.Write(ch + " ");
    Console.WriteLine();
} }
```

A B C D E

Создание именованного итератора

- Создается метод, оператор или аксессор, возвращающий ссылку на объект типа **IEnumerable**. Именно этот объект используется в коде для предоставления итератора. Именованный итератор представляет собой метод:

```
public IEnumerable имя_итератора(спмсок_параметров) {
//...
yield return obj;
}
```

где **имя_итератора** обозначает конкретное имя метода; **список_параметров** — от нуля до нескольких параметров, передаваемых методу итератора; obj — следующий объект, возвращаемый итератором.

- Как только именованный итератор будет создан, его можно использовать везде, где он требуется, например для управления циклом **foreach**.

Создание именованного итератора

```
// Использовать именованные итераторы
class MyClass {
  char ch = 'A';
// Этот итератор возвращает буквы английского алфавита,
// начиная с буквы A и кончая указанным конечным пределом
  public IEnumerable Myltr(int end) {
    for(int i=0; i < end; i++)
    yield return (char) (ch + i);
  }
  // Этот итератор возвращает буквы в заданных пределах
  public IEnumerable Myltr(int begin, int end) {
    for(int i=begin; i < end; i++)
      yield return (char) (ch + i);
  } }
class ItrDemo4 {
  static void Main() {
    MyClass mc = new MyClass ();
    Console.WriteLine("Возвратить по очереди первые 7 букв:");
    foreach(char ch in mc.MyItr(7)) Console.Write(ch + " ");
    Console.WriteLine("Возвратить по очереди буквы от F до L:");
    foreach(char ch in mc.MyItr(5, 12)) Console.Write(ch + " ");
} }
```

Возвратить по очереди первые 7 букв:
A B C D E F G
Возвратить по очереди буквы от F до L:
F G H I J K L

```
// Простой пример обобщенного итератора
using System;
using System.Collections.Generic;
class MyClass<T> {
  T[] array;
  public MyClass(T[] a) {
    array = a;
  }
  // Этот итератор возвращает символы из массива chrs
  public IEnumerator<T> GetEnumerator() {
    foreach(T obj in array) yield return obj;
} }

class GeneridtrDemo {
  static void Main() {
    int[] nums = { 4, 3, 6, 4, 7, 9 };
    MyClass<int> mc = new MyClass<int>(nums);
    foreach(int x in mc) Console.Write(x + " ");
} }
```

4 3 6 4 7 9
True True False True

Инициализаторы коллекций

- Вместо того чтобы явно вызывать метод **Add ()**, при создании коллекции можно указать список инициализаторов. После этого компилятор организует автоматические вызовы метода **Add ()**, используя значения из этого списка.

```
List<char> lst = new List<char>() { 'C, 'A', 'E', 'B', 'D', 'F' };
```

- После выполнения этого оператора значение свойства **lst.Count** будет равно 6, поскольку именно таково число инициализаторов.
- А после выполнения следующего цикла **foreach**:

```
foreach(ch in lst) Console.Write(ch + " ");
```

получится такой результат: C A E B D F

- Для инициализации коллекции типа **LinkedList<TKey, TValue>**, в которой хранятся пары "ключ-значение", инициализаторы приходится предоставлять парами.

```
SortedList<int, string> lst = new SortedList<int, string>()
{ {1, "один"}, {2, "два" }, {3, "три"} };
```