

# Платформа Microsoft .NET и язык программирования C#



# Урок №8

## Использование коллекций

## Содержание

1.	Понятие коллекции.....	4
2.	Обсуждение существующих коллекций .....	5
3.	Классы коллекций ArrayList, Stack, Queue, Hashtable, SortedList.....	7
4.	Интерфейсы коллекций ICollection, IList, IDictionary, IDictionaryEnumerator .....	25
5.	Примеры использования классов коллекций для хранения стандартных и пользовательских типов .....	26
6.	Generics.....	35
	Что такое generics?.....	35
	Необходимость использования generics.	
	Упаковка, распаковка (boxing, unboxing).....	35

Сравнительный анализ generic-коллекций и необобщенных коллекций . . . . .	40
Создание generic-классов . . . . .	51
Вложенные типы внутрь generic-класса . . . . .	57
Использование ограничений . . . . .	59
Создание generic-методов . . . . .	61
Generic-интерфейсы . . . . .	63
<b>7. Итераторы . . . . .</b>	<b>71</b>
Что такое итератор? . . . . .	71
Синтаксис и примеры использования итераторов . . . . .	71
<b>Домашнее задание . . . . .</b>	<b>77</b>

# 1. Понятие коллекции

---

Коллекцией называется набор объектов, сгруппированных по определенному признаку. В предыдущих уроках для хранения объектов Вы использовали массивы, которые также являются коллекциями, но статическими. Поэтому изменить размер массива в процессе работы программы фактически невозможно, что накладывает ограничения при работе с ними.

Помимо массивов в пространстве .NET содержится большое количество классов, представляющих динамические коллекции, приведем их краткую классификацию.

## 2. Обсуждение существующих коллекций

Основным пространством имён для всех классов коллекций является `System.Collections`. В этом пространстве имен находятся необобщенные коллекции, в которых хранятся данные типа `object`. В этом же пространстве имен содержится коллекция `BitArray`, которая содержит значения битов, представленных типом `bool`, и поддерживает поразрядные операции.

Также существует ряд коллекций во вложенных пространствах имен. В пространстве имен `System.Collections.Generic` находятся обобщенные коллекции, в которых допускается хранение данных, совместимых с заданным типом. Коллекции, поддерживающие многопоточный доступ, содержатся в пространстве имен `System.Collections.Concurrent`. В пространстве имен `System.Collections.Specialized` содержатся несколько классов специальных коллекций, работа которых происходит особенным образом. Приведем краткое описание каждого из них:

- `System.Collections.Specialized`.

`ListDictionary` — |

реализует словарь с помощью одностороннего списка, наилучшая производительность достигается, если количество элементов менее десяти;

- `System.Collections.Specialized`.

`HybridDictionary` — |

поддерживает переключение между двумя словарями в зависимости от количества элементов. Если количество элементов не превышает десяти, то используется словарь `ListDictionary`, при увеличении количества элементов происходит автоматическое переключение на `Hashtable`;

- `System.Collections.Specialized.CollectionsUtil` —  
при помощи этого класса можно создать коллекции, которые будут игнорировать регистр строк;
- `System.Collections.Specialized.NameValueCollection` —  
словарь, в котором и ключ, и значение представлены типом `string`, при этом одному ключу могут соответствовать несколько значений;
- `System.Collections.Specialized.StringCollection` —  
коллекция, содержащая элементы типа `string`, значения которых могут повторяться и иметь значение `null`;
- `System.Collections.Specialized.StringDictionary` —  
словарь, ключ и значение которого представлены типом `string`, ключ, в отличие от значения, не может быть `null`.

Теперь перейдем к более подробному рассмотрению классов необобщенных коллекций.

### 3. Классы коллекций ArrayList, Stack, Queue, Hashtable, SortedList

Как было сказано ранее, необщенные коллекции позволяют хранить данные типа `object`, то есть элементы данного типа коллекций могут быть различных типов.

Наиболее часто используемой необщенной коллекцией является `ArrayList`. Класс `ArrayList` способен динамически изменять свой размер при добавлении или удалении элементов. При создании этого типа коллекции можно воспользоваться одним из трех конструкторов, в одном из них существует возможность явно задать вместимость коллекции:

```
ArrayList arrayList1 = new ArrayList(); // вместимость по
                                         // умолчанию
ArrayList arrayList2 = new ArrayList(5); // начальное
                                         // значение
                                         // вместимости
// элементы копируются из указанной коллекции
ArrayList arrayList3 = new ArrayList(new int[] { 1, 5, 48 });
```

Задавая вместимость, Вы указываете возможное количество элементов коллекции. Вместимость по умолчанию у `ArrayList` равна нулю, при добавлении хотя бы одного элемента в коллекцию, вместимость становится равной четырем. Если при дальнейшем добавлении элементов их количество превысит текущее значение вместимости, то

последняя будет автоматически увеличена в два раза. Текущее значение вместимости можно задать или получить при помощи свойства Capacity. Вызов метода TrimToSize() уменьшает значение вместимости `ArrayList` до фактического количества его элементов, которое возвращает свойство Count. Добавление элементов в данную коллекцию осуществляется с помощью методов Add (`object`) и AddRange (`ICollection`) — первый применяется для добавления единичного объекта, а второй — для коллекции объектов. Доступ к элементам этой коллекции может осуществляться по индексу. Демонстрация вышеописанной функциональности коллекции `ArrayList` представлена ниже (Рисунок 3.1).

```
using System.Collections;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrayList1 = new ArrayList();
            WriteLine($"Вместимость по умолчанию:
{arrayList1.Capacity}");

            arrayList1.Add("one");
            WriteLine($"Вместимость после добавления одного
элемента: {arrayList1.Capacity}");

            arrayList1.AddRange(new int[] { 2, 5, 48, 14 });
            WriteLine($"Вместимость после добавления
коллекции: {arrayList1.Capacity}");
        }
    }
}
```

### 3. Классы коллекций ArrayList, Stack, Queue, Hashtable...

```
arrayList1.Capacity = 10;
WriteLine($"Вместимость задана через свойство:
          {arrayList1.Capacity}");
WriteLine($"Фактическое количество элементов:
          {arrayList1.Count}");
arrayList1.TrimToSize();
WriteLine($"Вместимость уменьшена до реального
          количества элементов:
          {arrayList1.Capacity}");
WriteLine($"Элемент с индексом 2:
          {arrayList1[2]}");
WriteLine("Все элементы коллекции:");
foreach (object item in arrayList1)
{
    WriteLine($"{item}");
}
}
}
```

```
C:\WINDOWS\system32\cmd.exe
Вместимость по умолчанию: 0
Вместимость после добавления одного элемента: 4
Вместимость после добавления коллекции: 8
Вместимость задана через свойство: 10
Фактическое количество элементов: 5
Вместимость уменьшена до реального количества элементов: 5
Элемент с индексом 2: 5
Все элементы коллекции:
    one
    2
    5
    48
    14
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3.1. Функционал коллекции ArrayList

При добавлении в коллекцию `ArrayList` элементов с помощью метода `Add (object)` они добавляются в конец коллекции. Однако можно добавлять элементы и в заданное место коллекции, для этого применяется метод `Insert (int, object)`, первый параметр которого — индекс, куда добавлять новый элемент, а второй параметр — значение этого элемента. Аналогичным образом можно добавить диапазон значений, вызвав для этого метод `InsertRange (int, ICollection)`, вторым параметром в который передается коллекция.

Для удаления элементов коллекции `ArrayList` используются подобные методы `RemoveAt (int)` и `RemoveRange (int, int)`.

Метод `GetRange (int, int)` коллекции `ArrayList` позволяет сформировать новую коллекцию со значениями элементов, взятых в указанном диапазоне из исходной коллекции. Первый параметр метода `GetRange (int, int)` указывает индекс начала диапазона, второй — количество элементов для копирования.

Для поиска первого вхождения объекта в коллекции используются методы `IndexOf (object)`, а для последнего `LastIndexOf (object)`. Метод `Sort ()` сортирует элементы коллекции. Использование данных методов представлено ниже (Рисунок 3.2).

```
using System.Collections;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
```

### 3. Классы коллекций ArrayList, Stack, Queue, Hashtable...

```
static void Main(string[] args)
{
    Write("Исходная коллекция: ");
    ArrayList arrayList = new ArrayList(new int[]
        { 1, 2, 3, 4 });
    foreach (int i in arrayList)
    {
        Write($"{i} ");
    }

    Write("\n\nВставка элемента: ");
    arrayList.Insert(2, "Hello");
    foreach (object item in arrayList)
    {
        Write($"{item} ");
    }

    Write("\n\nУдаление элемента: ");
    arrayList.RemoveAt(3);
    foreach (object item in arrayList)
    {
        Write($"{item} ");
    }

    WriteLine("\n\nИндекс элемента \"Hello\": " +
        arrayList.IndexOf("Hello"));

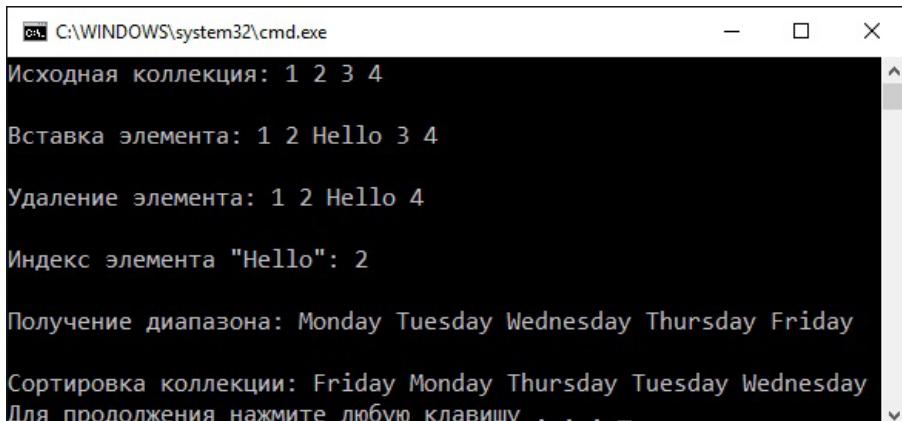
    Write("\nПолучение диапазона: ");

    ArrayList days = new ArrayList(new string[]
        { "Sunday", "Monday", "Tuesday",
        "Wednesday", "Thursday", "Friday",
        "Saturday" });

    ArrayList onlyWork = new ArrayList(days.
        GetRange(1, 5));
}
```

```
foreach (string s in onlyWork)
{
    Write(${s} );
}

Write("\n\nСортировка коллекции: ");
onlyWork.Sort();
foreach (string s in onlyWork)
{
    Write(${s} );
}
WriteLine();
}
}
```



The screenshot shows a command prompt window titled 'cmd' with the path 'C:\WINDOWS\system32\cmd.exe'. The window displays the output of a C# program. The output is as follows:

```
Исходная коллекция: 1 2 3 4
Вставка элемента: 1 2 Hello 3 4
Удаление элемента: 1 2 Hello 4
Индекс элемента "Hello": 2
Получение диапазона: Monday Tuesday Wednesday Thursday Friday
Сортировка коллекции: Friday Monday Thursday Tuesday Wednesday
Для продолжения нажмите любую клавишу . . .
```

**Рисунок 3.2.** Методы коллекции *ArrayList*

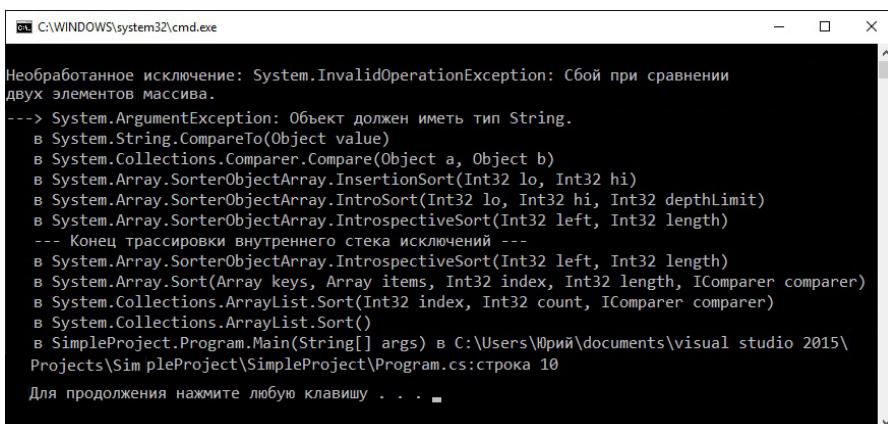
При вызове метода *Sort()* следует учитывать, что в коллекции *ArrayList* могут находиться элементы различных типов данных, в этом случае попытка отсортировать коллекцию приведет к ошибке на этапе выполнения (Рисунок 3.3).

### 3. Классы коллекций ArrayList, Stack, Queue, Hashtable...

```
using System.Collections;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrayList = new ArrayList(new object[]
                { 1, "Help", 3, 4 });
            arrayList.Sort();

        }
    }
}
```



**Рисунок 3.3.** Ошибка на этапе выполнения

Рассмотрим следующую коллекцию — **Stack**. Класс **Stack** — это коллекция, которая работает по принципу LIFO (*last-in-first-out*), то есть последовательность извлечения объектов противоположна последовательности их помещения в коллекцию.

Для создания данного типа коллекции существуют три конструктора:

```
Stack stack1 = new Stack(); // вместимость по умолчанию  
Stack stack2 = new Stack(7); // начальное значение емкости  
// элементы копируются из указанной коллекции  
Stack stack3 = new Stack(new ArrayList { 3, 5 });
```

Обратиться с помощью индекса к элементу коллекции `Stack` нельзя. В данную коллекцию элементы помещаются с помощью метода `Push (object)`, а извлекаются при помощи метода `Pop ()`. Если надо получить значение последнего элемента коллекции без его удаления, то необходимо применить метод `Peek ()`. Для копирования содержимого коллекции в массив применяется метод `CopyTo (Array, int)`. Проверка на наличие в коллекции `Stack` заданного значения, осуществляется при помощи свойства `Contains`, которое возвращает `true` в случае положительного результата и `false` в случае отрицательного. Для очистки содержимого коллекции применяется метод `Clear ()`. Результаты работы с коллекцией `Stack` показаны на рисунке 3.4.

```
using System.Collections;  
using static System.Console;  
  
namespace SimpleProject  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Stack stack = new Stack();
```

```
Write("Метод Push(): ");
stack.Push("one");
stack.Push("two");
stack.Push("three");

foreach (string item in stack)
{
    Write(${item} );
}

Write("\n\nМетод Pop(): ");
stack.Pop();
foreach (string item in stack)
{
    Write(${item} );
}

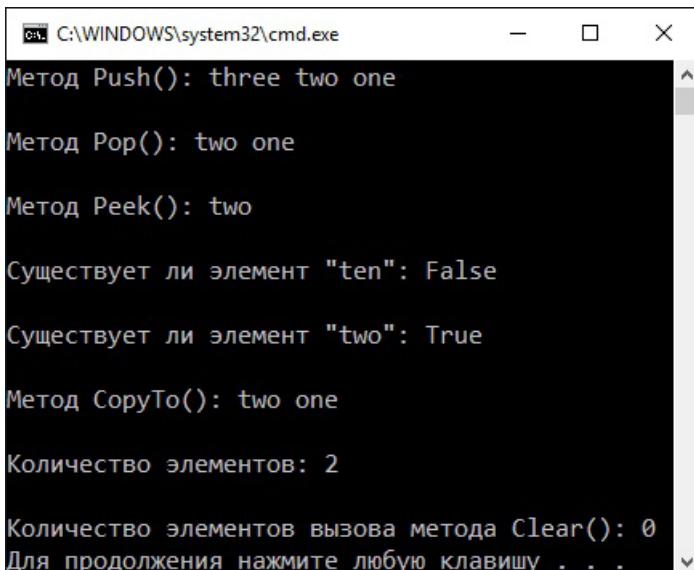
WriteLine(${"\n\nМетод Peek(): "
    ${{(string)stack.Peek()}}});

WriteLine("\nСуществует ли элемент \"ten\": " +
    stack.Contains("ten")); // false
WriteLine("\nСуществует ли элемент \"two\": " +
    stack.Contains("two")); // true

Write("\nМетод CopyTo(): ");
string[] strArr = new string[stack.Count];
stack.CopyTo(strArr, 0);
foreach (string item in strArr)
{
    Write(${item} );
}

WriteLine("\n\nКоличество элементов: " +
    stack.Count); // 3
stack.Clear();
```

```
        WriteLine("\nКоличество элементов вызова метода  
        Clear(): " + stack.Count); // 0  
    }  
}  
}
```



**Рисунок 3.4.** Работа с коллекцией Stack

Коллекция `Queue` работает по принципу FIFO (*first-in-first-out*), это значит, что элементы извлекаются из коллекции в той же последовательности, в которой они были в неё помещены. Индексация к элементам данной коллекции не применяется.

При создании коллекции задается вещественное значение — коэффициент роста, по умолчанию он равен 2.0, но существует конструктор, в котором данный коэффициент можно задать явно в диапазоне от 1.0 до 10.0:

```
Queue queue1 = new Queue(); // вместимость по умолчанию
Queue queue2 = new Queue(3); // начальное значение емкости
// элементы копируются из указанной коллекции
Queue queue3 = new Queue(new ArrayList { "one", 8.4 });
// начальное значение емкости и коэффициент роста
Queue queue4 = new Queue(7, 3.0f);
```

Основными методами очереди являются методы `Enqueue(object)` и `Dequeue()` для помещения элементов в коллекцию и извлечения соответственно. Метод `Peek()` возвращает значение элемента, который был первым помещён в коллекцию при этом не удаляя его. Демонстрация работы с коллекцией `Queue` представлена ниже (Рисунок 3.5).

```
using System.Collections;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Write("Метод Enqueue(): ");
            Queue queue = new Queue();
            for (int i = 1; i < 4; i++)
            {
                queue.Enqueue(i);
            }

            foreach (int item in queue)
                Write($"{item} ");
            WriteLine($"\\n\\nМетод Dequeue():
{queue.Dequeue()}\\n");
        }
    }
}
```

```
        WriteLine($"Существует ли элемент 1:  
        {queue.Contains(1)}\n");  
  
        WriteLine($"Метод Peek() {queue.Peek()}\n");  
  
        WriteLine($"Существует ли элемент 2:  
        {queue.Contains(2)}");  
    }  
}  
}
```

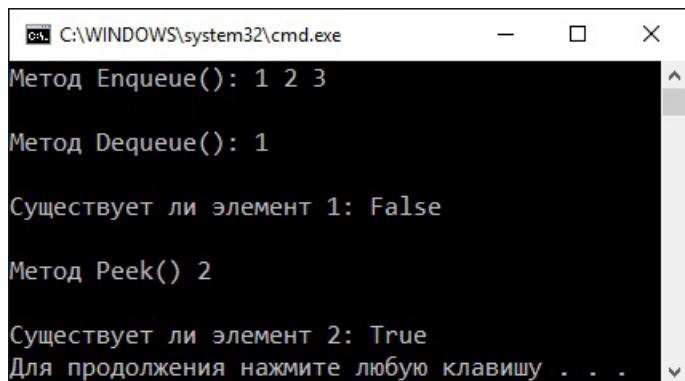


Рисунок 3.5. Методы коллекции Queue

Коллекция [Hashtable](#) представляет собой словарь, каждый элемент которого состоит из пары ключ-значение, при этом значение ключа должно быть уникальным. Для хранения элементов данной коллекции используется хеш-таблица, в которой хранятся значения по соответствующему ключу. Ключи хранятся в виде хеш-кода, который вычисляется при осуществлении записи в коллекцию. Хеш-код это некое уникальное целочисленное значение, которое вычисляется по определенному алгоритму, заключенному в реализации метода `GetHashCode()` для

конкретного типа. Так вот для чего надо переопределять метод `GetHashCode()` в своем классе!

Благодаря данному подходу продолжительность операций поиска, записи и удаления не зависит от количества данных в коллекции. Еще одним преимуществом хеширования является то, что ключи могут иметь разный тип данных, например одна пара ключ-значение может быть `int-string`, а другая `string-object` и так далее.

При создании словаря (существует 15 конструкторов) можно явно указать его вместимость, причём для этого рекомендуется использовать простые числа, что связано с алгоритмом работы словарей, так они работают эффективнее. Также при создании `Hashtable` можно указать коэффициент заполнения в пределах от 0.1 до 1.0, который влияет на процесс увеличения размеров хеш-таблицы — таблица расширяется если количество элементов становится больше емкости таблицы умноженной на этот коэффициент (по умолчанию равен 1.0).

Определить точное место в `Hashtable` куда будет помещена новая запись при добавление (метод `Add(object, object)`) нельзя, так как ее месторасположение вычисляется на основании хеш-кода ключа, да оно нам в принципе и не интересно, главное скорость получения данных из коллекции. Удаление записи осуществляется методом `Remove(object)` с указанием ключа.

Словарь характерен рядом свойств и методов, которых нет в обычной коллекции. Так, например, при помощи свойств `Keys` и `Values` можно получить коллекции ключей или значений текущего словаря, соответственно. А методы

ContainsKey(object) и ContainsValue(object) позволяют определить содержит ли текущий `Hashtable` указанный ключ или значение, соответственно. Пример работы с `Hashtable` представлен ниже (Рисунок 3.6).

```
using System.Collections;
using static System.Console;

namespace SimpleProject
{
    class Student
    {
        public string Name { get; set; }
        public override string ToString()
        {
            return $"Имя: {Name}";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Hashtable hash = new Hashtable();
            hash.Add(1, "John");
            hash.Add("two", new Student { Name = "Jack" });

            WriteLine("----- Вывод ключ-значение -----\\n");
            foreach (object item in hash.Keys)
            {
                WriteLine("Ключ: " + item + " Значение: "
                    + hash[item]);
            }

            hash.Add("Pi", 3.14159);
        }
    }
}
```

### 3. Классы коллекций ArrayList, Stack, Queue, Hashtable...

```
        WriteLine("\n----- Коллекция значений после  
        добавление элемента -----\"n");  
    foreach (object item in hash.Values)  
        WriteLine(item);  
  
    WriteLine("\n----- Удаление элемента по ключу  
    \"two\" -----\"n");  
    hash.Remove("two");  
  
    WriteLine($"Существует ли элемент \"two\":  
    {hash.ContainsKey("two")}); // false  
}  
}  
}
```

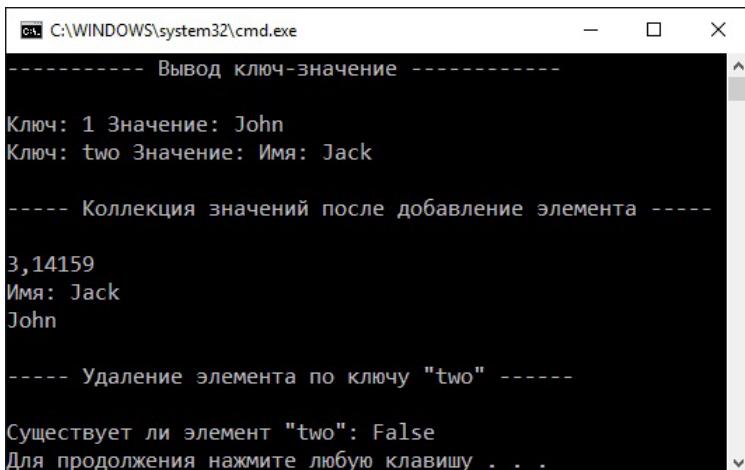


Рисунок 3.6. Методы словаря Hashtable

Коллекция `SortedList` является словарем, объекты которого сортируются по ключу автоматически при добавлении нового элемента, доступ к элементам можно получить как по ключу, так и по индексу.

Для создания `SortedList` можно воспользоваться одним из шести доступных конструкторов. Вместимость словаря по умолчанию равна нулю, после добавления первого элемента она становится равной 16. Если фактическое количество элементов превышает это значение, то вместимость автоматически удваивается.

Методы `SortedList` аналогичны методам класса `Hashtable`, но есть существенное отличие между этими коллекциями — ключи `SortedList` должны быть одного и того же типа данных, иначе словарь нельзя будет отсортировать. Возможный вариант работы с `SortedList` представлен на рисунке 3.7.

The screenshot shows a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. The output is as follows:

```
----- Вывод ключ-значение -----
Ключ: 1 Значение: one
Ключ: 2 Значение: Имя: Jack
Ключ: 3 Значение: 6,7

----- Вывод ключ-значение по индексу -----

Ключ: 1 Значение: one
Ключ: 2 Значение: Имя: Jack
Ключ: 3 Значение: 6,7

----- Коллекция значений -----

one
Имя: Jack
6,7

----- Удаление элемента по ключу 3 -----

Существует ли элемент 3: False
Для продолжения нажмите любую клавишу . . .
```

Рисунок 3.7. Использование словаря `SortedList`

```
using System.Collections;
using static System.Console;
namespace SimpleProject
{
    class Student
    {
        public string Name { get; set; }
        public override string ToString()
        {
            return $"Имя: {Name}";
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            SortedList sortedList = new SortedList();
            sortedList.Add(3, 6.7);
            sortedList.Add(2, new Student { Name = "Jack"
});

            sortedList.Add(1, "one");
            WriteLine("----- Вывод ключ-значение -----\\n");
            foreach (object item in sortedList.Keys)
            {
                WriteLine($"Ключ: {item}
Значение: {sortedList[item]}");
            }

            WriteLine("\n----- Вывод ключ-значение
по индексу -----\\n");

            for (int i = 0; i < sortedList.Count; i++)
            {
                WriteLine($"Ключ: {sortedList.GetKey(i)}
Значение: {sortedList.GetByIndex(i)}");
            }
        }
    }
}
```

```
WriteLine("\n---- Коллекция значений ----\n");
foreach (object item in sortedList.Values)
    WriteLine(item);

WriteLine("\n---- Удаление элемента
        по ключу 3 -----");
sortedList.Remove(3);

WriteLine($"Существует ли элемент 3:
{sortedList.ContainsKey(3)}"); // false
}
}
}
```

# 4. Интерфейсы коллекций ICollection, IList, IDictionary, IDictionaryEnumerator

В пространстве имён `System.Collections` содержатся интерфейсы, которые определяют функционал, связанный с управлением элементами коллекций. Некоторые из них уже были рассмотрены в уроке №6, в этом разделе мы дадим краткое описание еще ряду ключевых интерфейсов.

Интерфейс `ICollection` реализуется любой коллекцией и определяет набор свойств для: определения количества элементов в коллекции, копирования содержимого и обеспечения потокобезопасности.

Интерфейс `IList` обеспечивает основную функциональность для списочных коллекций (запись, вставку, удаление, поиск, очистку). Также позволяет реализовать индексатор и свойства, при помощи которых можно узнать доступна ли текущая коллекция только для чтения и фиксирован ли у нее размер.

Интерфейс `IDictionary` — базовый интерфейс для всех коллекций типа словарь и определяет весь основной функционал для работы с данным типом коллекций.

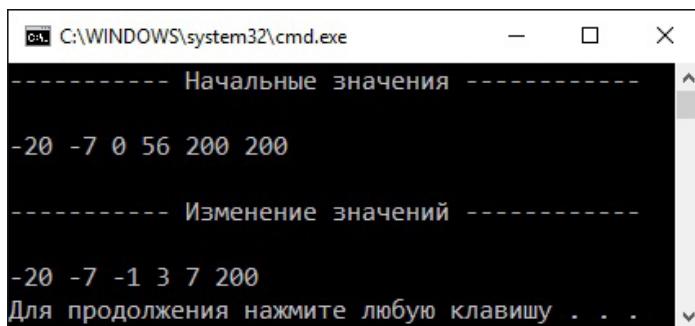
Интерфейс `IDictionaryEnumerator` — позволяет перечислять содержимое коллекции, реализующей интерфейс `IDictionary`, весь функционал только для чтения.

Для получения более подробной информации Вы можете воспользоваться окном ObjectBrowser или MSDN.

# 5. Примеры использования классов коллекций для хранения стандартных и пользовательских типов

Небольшие примеры использования коллекций Вы уже видели в разделе №3, в этом разделе мы покажем, как коллекции можно применять для решения более сложных задач.

В первом примере мы создадим класс [SortedArrayList](#), производный от [ArrayList](#), который будет представлять собой автоматически отсортированную коллекцию элементов. В этом классе мы определим метод для добавления элементов в коллекцию так, чтобы они сразу были отсортированы, а также метод для изменения значения элемента по заданному индексу, который также гарантирует расположение элементов в правильном



The screenshot shows a command-line interface window titled 'cmd' with the path 'C:\WINDOWS\system32\cmd.exe'. The window displays two sections of text:

```
----- Начальные значения -----
-20 -7 0 56 200 200

----- Изменение значений -----
-20 -7 -1 3 7 200
Для продолжения нажмите любую клавишу . . .
```

Рисунок 5.1. Использование класса SortedArrayList

порядке. Пример использования класса [SortedArrayList](#) представлен ниже (Рисунок 5.1).

```
using System.Collections;
using static System.Console;

namespace SimpleProject
{
    public class SortedArrayList : ArrayList
    {
        public void AddSorted(object item)
        {
            int position = BinarySearch(item);

            if (position < 0)
            {
                position = ~position;
            }

            Insert(position, item);
        }

        public void ModifySorted(object item, int index)
        {
            RemoveAt(index);

            int position = BinarySearch(item);

            if (position < 0)
            {
                position = ~position;
            }

            Insert(position, item);
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        SortedArrayList sortedAL = new SortedArrayList();

        WriteLine("----- Начальные значения -----\\n");
        sortedAL.AddSorted(200);
        sortedAL.AddSorted(-7);
        sortedAL.AddSorted(0);
        sortedAL.AddSorted(-20);
        sortedAL.AddSorted(56);
        sortedAL.AddSorted(200);

        foreach (int i in sortedAL)
        {
            Write(i + " ");
        }
        WriteLine();

        WriteLine("\\n----- Изменение значений -----\\n");
        sortedAL.ModifySorted(3, 5);
        sortedAL.ModifySorted(-1, 2);
        sortedAL.ModifySorted(2, 4);
        sortedAL.ModifySorted(7, 3);

        foreach (int i in sortedAL)
        {
            Write(i + " ");
        }
        WriteLine();
    }
}
```

В следующем примере, мы продемонстрируем использование коллекции `ArrayList` для хранения пользова-

тельского типа, в данном случае это будет класс `Student`, созданный нами в уроке №6. После внесения определенных корректив в существующий код, мы получили следующий результат (Рисунок 5.2).

```
using System;
using System.Collections;
using static System.Console;

namespace SimpleProject
{
    class DateComparer : IComparer
    {
        public int Compare(object x, object y)
        {
            if (x is Student && y is Student)
            {
                return DateTime.Compare((x as Student).BirthDate, (y as Student).BirthDate);
            }
            throw new NotImplementedException();
        }
    }

    class Student : IComparable
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime BirthDate { get; set; }

        public override string ToString()
        {
            return $"Фамилия: {LastName}, Имя: {FirstName},  
Родился: {BirthDate.ToString("yyyy-MM-dd")}";
        }
    }
}
```

```
public int CompareTo(object obj)
{
    if (obj is Student)
    {
        return LastName.CompareTo((obj as Student).
            LastName);
    }
    throw new NotImplementedException();
}

class Program
{
    static void Main(string[] args)
    {
        ArrayList auditory = new ArrayList {
            new Student {
                FirstName ="John",
                LastName ="Miller",
                BirthDate =new DateTime(1997,3,12)
            },
            new Student {
                FirstName ="Candice",
                LastName ="Leman",
                BirthDate =new DateTime(1998,7,22)
            }
        };

        WriteLine("++++ список студентов +++\n");
        foreach (Student student in auditory)
        {
            WriteLine(student);
        }
        WriteLine("\n++++ сортировка по фамилии +++\n");
        auditory.Sort();
        foreach (Student student in auditory)
        {
```

```
        WriteLine(student);
    }
    WriteLine("\n+++++ сортировка по дате рождения
+++++\n");
    auditory.Sort(new DateComparer());
    foreach (Student student in auditory)
    {
        WriteLine(student);
    }
}
}
```

```
C:\WINDOWS\system32\cmd.exe
++++++ список студентов ++++++
Фамилия: Miller, Имя: John, Родился: 12 марта 1997 г.
Фамилия: Leman, Имя: Candice, Родился: 22 июля 1998 г.

++++++ сортировка по фамилии ++++++
Фамилия: Leman, Имя: Candice, Родился: 22 июля 1998 г.
Фамилия: Miller, Имя: John, Родился: 12 марта 1997 г.

++++++ сортировка по дате рождения ++++++
Фамилия: Miller, Имя: John, Родился: 12 марта 1997 г.
Фамилия: Leman, Имя: Candice, Родился: 22 июля 1998 г.
Для продолжения нажмите любую клавишу . . .
```

**Рисунок 5.2.** Пример хранения пользовательского типа  
в коллекции ArrayList

Еще один пример демонстрирует возможность использования словаря `Hashtable` для хранения информации об оценках, полученных студентами. Ключом для каждой записи будет экземпляр класса `Student`, а соответствующим

ему значением будет коллекция `ArrayList` — список оценок. Для реализации данного примера необходимо переопределить метод `GetHashCode()` в классе `Student`, результат представлен на рисунке 5.3.

```
using System.Collections;
using static System.Console;

namespace SimpleProject
{
    class Student
    {
        public string FirstName { get; set; }
        public string LastName { get; set; }

        public override string ToString()
        {
            return $"{LastName} {FirstName}";
        }

        public override int GetHashCode()
        {
            return ToString().GetHashCode();
        }
    }

    class Program
    {
        static Hashtable group = new Hashtable {
            {
                new Student {
                    FirstName ="John",
                    LastName ="Miller"
                },new ArrayList{8,4,9}
            },
        }
    }
}
```

```

    {
        new Student {
            FirstName ="Candice",
            LastName ="Leman"
        },new ArrayList{12,9,10}
    }
};

static void RatingsList()
{
    WriteLine("++++++ Список студентов
    с оценками ++++++\n");

    foreach (Student student in group.Keys)
    {
        Write($"{student}: ");
        foreach (int item in (group[student] as
            ArrayList))
        {
            Write($"{item} ");
        }
        WriteLine();
    }
}

static void SetRating(string name, int mark)
{
    WriteLine($"\\n++++++ Студент {name}
    получил оценку {mark} +++++\n");

    foreach (Student item in group.Keys)
    {
        if (item.LastName == name)
        {
            (group[item] as ArrayList).Add(mark);
        }
    }
}

```

```
static void Main(string[] args)
{
    RatingsList();
    SetRating("Leman", 11);
    RatingsList();
}
}
```

```
C:\WINDOWS\system32\cmd.exe
++++++ Список студентов с оценками ++++++
Miller John: 8 4 9
Leman Candice: 12 9 10

++++++ Студент Leman получил оценку 11 +++++

++++++ Список студентов с оценками ++++++
Miller John: 8 4 9
Leman Candice: 12 9 10 11
Для продолжения нажмите любую клавишу . . .
```

**Рисунок 5.3.** Использование словаря Hashtable

Как Вы заметили класс `Student` значительно упростился, это сделано только целях экономии места, можете проверить самостоятельно, что с более сложной реализацией класса пример будет работать аналогично.

# 6. Generics

## Что такое generics?

В .Net Framework версии 2.0 была добавлена поддержка обобщенных типов (generics), которые предоставляют возможность создавать различные конструкции для конкретного типа данных, который и указывается в качестве параметра. Исходя из этого, Вы можете встретить еще два возможных названия — параметризованный тип и типизированный объект, например, типизированная коллекция.

Обобщенными могут быть:

- коллекции;
- классы;
- структуры;
- методы;
- интерфейсы;
- делегаты.

Обобщенные типы очень легко распознать, они отличаются наличием угловых скобок (<>) после названия типа, внутри которых указывается определенный тип или типы данных.

## Необходимость использования generics. Упаковка, распаковка (boxing, unboxing)

Введение обобщенных типов можно расценивать как этап эволюции языка C#, спровоцированный двумя важными проблемами, которые возникают при работе

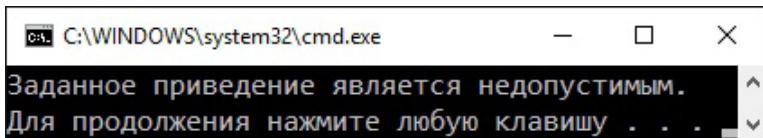
с необщенными коллекциями — низкая производительность и безопасность типов.

Проблема безопасности типов связана с универсальностью необщенных коллекций — хранение данных типа `object`. Такая гибкость в некоторых случаях удобна, когда в коллекцию необходимо поместить переменные любого типа. Однако при извлечении данных из коллекции необходимо осуществлять явное приведение типа, при этом можно легко допустить ошибку приведения, которая проявится только на этапе выполнения программы. Поэтому работа с необщенными коллекциями еще должна сопровождаться написанием дополнительного кода проверок и приведения — блок `try-catch` или операторы `is` или `as` (Рисунок 6.1).

```
using System;
using System.Collections;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrayList = new ArrayList();
            // помещаем в коллекцию элементы типа int
            arrayList.Add(10);
            try
            {
                // при извлечении выполняем приведение
                // к типу short из-за ошибочного указания
                // типа возникает исключение
                short a = (short)arrayList[0];
```

```
        }
    catch (InvalidOperationException e)
    {
        WriteLine(e.Message);
    }
}
```



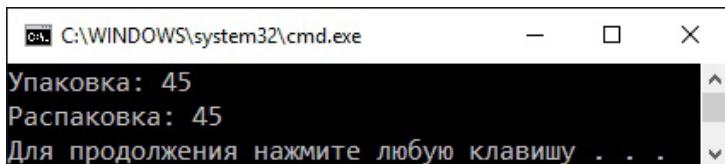
**Рисунок 6.1.** Неправильное приведение типов

Низкая производительность при работе с необобщенными коллекциями связана с механизмом языка C#, так называемой упаковкой-распаковкой (*boxing-unboxing*), который запускается средой CLR без участия программиста.

Как Вы знаете, в языке C# существуют два типа данных: ссылочные и значимые. Когда Вы инициируете ссылочный тип значимым значением, автоматически происходит процесс упаковки (*boxing*). Получение значимым типом значения из ссылочной переменной запускает обратный механизм — распаковку (*unboxing*). Продемонстрируем это на примере (Рисунок 6.2).

```
using static System.Console;
namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
```

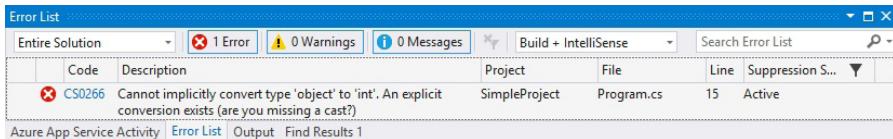
```
{  
    object obj = 45; // boxing  
    WriteLine($"Упаковка: {obj}");  
  
    int number = (int)obj; // unboxing  
    WriteLine($"Распаковка: {number}");  
}  
}  
}
```



**Рисунок 6.2.** Упаковка-распаковка типов

Следует обратить Ваше внимание на необходимость осуществления явного приведение типов, иначе сгенерируется ошибка на этапе компиляции (Рисунок 6.3).

```
using static System.Console;  
  
namespace SimpleProject  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            object obj = 45; // boxing  
            WriteLine($"Упаковка: {obj}");  
            int number = obj; // unboxing  
            WriteLine($"Распаковка: {number}");  
        }  
    }  
}
```



**Рисунок 6.3. Ошибка: нельзя конвертировать тип object в int**

В необщенных коллекциях, по определению, хранятся элементы типа `object`, и если данный вид коллекций используется для хранения значимых типов, то упаковка-распаковка типов сопровождает весь процесс работы с такими коллекциями (Рисунок 6.4).

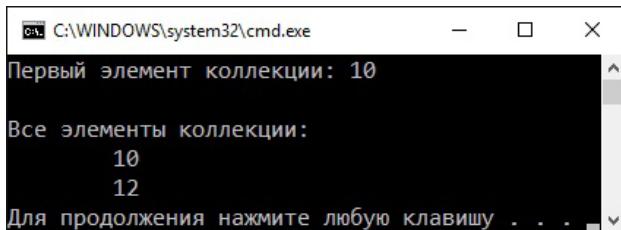
```
using System.Collections;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrayList = new ArrayList();
            // при записи элемента в коллекцию тип int
            // приводится в object - УПАКОВКА
            arrayList.Add(10);
            arrayList.Add(12);

            // при извлечении выполняем приведение типа -
            // РАСПАКОВКА
            int a = (int)arrayList[0];

            // вывод значения - int приводится к string -
            // УПАКОВКА
            Write($"Первый элемент коллекции: {a}"); //
            WriteLine("\n\nВсе элементы коллекции:");
        }
    }
}
```

```
        foreach (int item in arrayList)
        {
            WriteLine($"\\t{item}"); // УПАКОВКА
        }
    }
}
```



**Рисунок 6.4.** Упаковка-распаковка в необобщенных коллекциях

Использование обобщенных типов дает следующие преимущества:

- безопасность типов — в generic-коллекцию можно поместить только объекты определенного типа, указанного при раскрытии шаблона;
- более простой и понятный код, так как не нужно выполнять приведение типов;
- повышение производительности — при использовании generics, структурные типы передаются по значению, упаковка и распаковка не осуществляется.

### Сравнительный анализ generic-коллекций и необобщенных коллекций

С появлением концепции generics в FCL практически для всех классов необобщенных коллекций появились соответствующие им обобщенные коллекции. Эти

коллекции, а также обобщенные интерфейсы находятся в пространстве имен System.Collections.Generic.

В таблице 6.1 представлены обобщенные и соответствующие им необобщенные классы коллекций.

**Таблица 6.1.** Соответствие классов коллекций

Generic-коллекции	Необобщенные коллекции
Collection<T>	CollectionBase
List<T>	ArrayList
Dictionary< TKey, TValue >	Hashtable
SortedList< TKey, TValue >	SortedList
Stack<T>	Stack
Queue<T>	Queue
LinkedList<T>	Нет

Главное отличие generic-коллекций от необобщенных коллекций (помимо не всегда совпадающих названий) состоит в том, что вместо типа данных *object* в generic-коллекциях используется обобщенный параметр типа, который указывается в угловых скобках после названия коллекции. В результате те методы и свойства, которые в необобщенной коллекции использовали тип данных *object*, в generic-коллекции используют конкретный тип данных.

Продемонстрируем безопасность типов при работе с обобщенными коллекциями на примере коллекции *List<int>*. Попытка поместить в нее значение, тип которого отличается от типа обобщенного параметра, приведет к ошибке на этапе компиляции (Рисунок 6.5).

```
using System.Collections.Generic;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            List<int> listInt = new List<int> { 53, 12, 78
};

            listInt.Add(8.9);

        }
    }
}
```



**Рисунок 6.5.** Ошибка: нельзя конвертировать тип double в int

Для сравнения временных затрат при хранении структурных типов в необобщенной и обобщенной коллекции мы приведем пример, предложенный Jeffrey Richter в книге «The CLR via C#».

Для выполнения профилирования используется вспомогательный класс `OperationTimer`, назначение которого — точное измерение времени выполнения участка кода, а также подсчет количества сборок мусора. Этот класс является удачным паттерном и его можно использовать для профилирования в собственных проектах.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Diagnostics;
using static System.Console;

namespace SimpleProject
{
    /// <summary>
    /// Вспомогательный класс для профилирования участка кода
    /// выполняет измерения времени выполнения
    /// и подсчет количества сборок мусора
    /// </summary>
    internal sealed class OperationTimer : IDisposable
    {
        long _startTime;
        string _text;
        int _collectionCount;

        public OperationTimer(string text)
        {
            PrepareForOperation();

            _text = text;

            // сохраняется количество сборок мусора,
            // выполненных на текущий момент
            _collectionCount = GC.CollectionCount(0);

            // сохраняется начальное время
            _startTime = Stopwatch.GetTimestamp();
        }

        /// <summary>
        /// Вызывается при разрушении объекта
        /// Выводит:
        /// значение времени, прошедшего с момента
    }
}
```

```
    /// создания объекта до момента его удаления
    /// количество выполненных сборок мусора,
    /// выполненных за это время
    /// </summary>
public void Dispose()
{
    WriteLine($"{_text}\t{(_Stopwatch.GetTimestamp() - _startTime) / (double)_Stopwatch.Frequency:0.00}
    секунды (сборок мусора {GC.CollectionCount(0) - collectionCount})");
}

/// <summary>
/// Метод удаляются все неиспользуемые объекты
/// Это надо для "чистоты эксперимента",
/// т.е. чтобы сборка мусора происходила только
/// для объектов, которые будут создаваться
/// в профилируемом блоке кода
///</summary>

private static void PrepareForOperation()
{
    GC.Collect();
    GC.WaitForPendingFinalizers();
    GC.Collect();
}

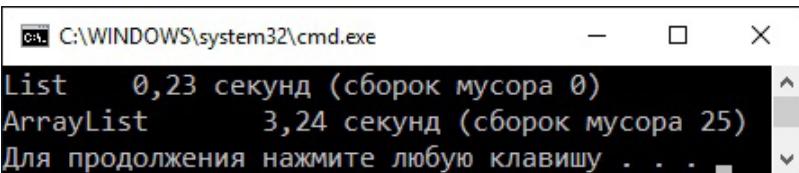
class Program
{
    /// <summary>
    /// метод для тестирования производительности
    /// обобщенного и необобщенного списка
    /// </summary>
private static void ValueTypePerfTest()
{
    const int COUNT = 10000000;
```

```

// объект OperationTimer создается
// перед началом использования коллекции
// после завершения ее использования
// выводит время, затраченное на работу
// с коллекцией
using (new OperationTimer("List"))
{
    //использование обобщенного списка
    List<int> list = new List<int>(COUNT);
    for (int n = 0; n < COUNT; n++)
    {
        list.Add(n);
        int x = list[n];
    }
    list = null; // для гарантированного
                  // выполнения сборки мусора
}
using (new OperationTimer("ArrayList"))
{
    // использование необобщенного списка
    ArrayList array = new ArrayList();
    for (int n = 0; n < COUNT; n++)
    {
        array.Add(n); // выполняется упаковка
        int x = (int)array[n]; // выполняется
                               // распаковка
    }
    array = null; // для гарантированного
                  // выполнения сборки мусора
}
static void Main(string[] args)
{
    ValueTypePerfTest();
}
}
}

```

Полученный Вами результат может отличаться от приведенного на рисунке 6.6, это зависит от параметров компьютера.



The screenshot shows a Windows Command Prompt window titled 'cmd C:\WINDOWS\system32\cmd.exe'. It displays two lines of output: 'List 0,23 секунд (сборок мусора 0)' and 'ArrayList 3,24 секунд (сборок мусора 25)'. Below the output, a message reads 'Для продолжения нажмите любую клавишу . . .'. The window has standard minimize, maximize, and close buttons at the top right.

**Рисунок 6.6.** Сравнения временных затрат в необобщенной и обобщенной коллекции

Из полученных результатов видно, что использование необобщенной коллекции `ArrayList` приводит к значительным временными затратам, а также к увеличению сборок мусора.

Так как generic-коллекции обеспечивают безопасность типов, и увеличение производительности при работе со структурными типами данных, корпорация Microsoft рекомендует везде, где это возможно, использовать обобщенные коллекции вместо необобщенных. Использование необобщенной коллекции может быть оправданным только если действительно существует необходимость хранить в одной коллекции объекты разных типов.

Работа с обобщенными коллекциями осуществляется также как и с необобщенными — используются подобные методы и свойства. Для демонстрации наших слов приведем примеры применения двух самых используемых обобщенных коллекций. Первый пример демонстрирует работу с generic-коллекцией `List<T>` (Рисунок 6.7).

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{

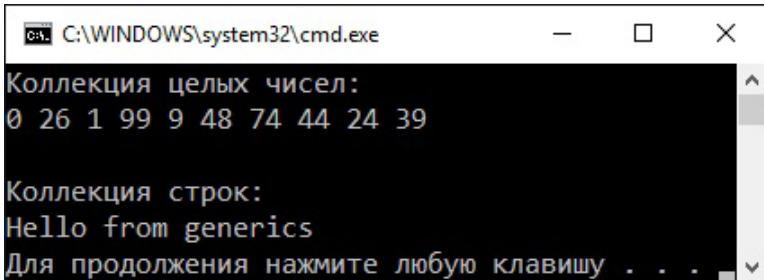
    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Коллекция целых чисел: ");
            List<int> listInt = new List<int>();
            Random rnd = new Random();
            for (int i = 0; i < 10; i++)
                listInt.Add(rnd.Next(100));

            foreach (int i in listInt)
            {
                Write(${i} );
            }

            WriteLine("\n\nКоллекция строк: ");
            List<string> listString = new List<string>();
            listString.Add("Hello");
            listString.Add("from");
            listString.Add("generics");

            foreach (string s in listString)
            {
                Write(${s} );
            }

            WriteLine();
        }
    }
}
```



**Рисунок 6.7.** Пример работы с generic-коллекцией List<T>

Обобщенная коллекция Dictionary является словарем, поэтому при ее создании следует указывать два параметра — ключ и значение, пример ее использования приведен на рисунке 6.8.

```
using System;  
using System.Collections.Generic;  
using static System.Console;  
  
namespace SimpleProject  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // Словарь для хранения пар: string-int  
            Dictionary<string, int> groups =  
                new Dictionary<string, int>();  
            // добавление значений в список  
            groups["GR1"] = 12;  
            groups.Add("GR2", 10);  
            groups.Add("GR3", 10);  
            groups.Add("GR4", 6);  
  
            // изменение значения  
            groups["GR1"] = 14;
```

```
// вывод всех элементов словаря
WriteLine("Содержимое словаря: ");

foreach (KeyValuePair<string, int> p in groups)
{
    WriteLine(${p.Key} {p.Value}");
}

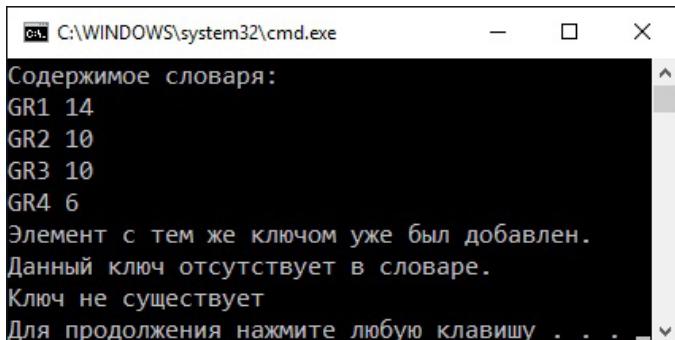
// удаление по значению ключа
groups.Remove("GR4");

// попытка добавления существующего ключа
try
{
    groups.Add("GR1", 15);
}
catch (ArgumentException e)
{
    WriteLine(e.Message);
}

// попытка обращения к несуществующему ключу
try
{
    WriteLine(groups["GR5"]);
}
catch (KeyNotFoundException e)
{
    WriteLine(e.Message);
}

// проверка существования ключа и получение
// значения
int key;
if (groups.TryGetValue("GR5", out key))
{
    WriteLine(key);
}
```

```
        else
        {
            WriteLine("Ключ не существует");
        }
    }
}
```



**Рисунок 6.8.** Пример работы с generic-коллекцией Dictionary<TKey, TValue>

В версии 6.0 языка C# упрощен процесс инициализации Dictionary начальными значениями, теперь значения можно присвоить при помощи ключа. Продемонстрируем это, взяв за основу предыдущий пример, результат при этом не изменится.

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
```

```

static void Main(string[] args)
{
    // Словарь для хранения пар: string-int
    Dictionary<string, int> groups =
        new Dictionary<string, int>
    {
        ["GR1"] = 12,
        ["GR2"] = 10,
        ["GR3"] = 10,
        ["GR4"] = 6
    };
    // остальной код остался прежним
}
}
}

```

## Создание generic-классов

При создании generic-класса параметр типа указывается в угловых скобках после имени класса, затем этот тип используется также как обычные типы. Обобщенных параметров типа может быть и несколько.

При использовании обобщенного класса вместо параметра типа подставляют реальный тип данных. Создание конкретного экземпляра generic-класса называется инстанцированием или специализацией.

Так как параметр типа заранее неизвестен, то нельзя определиться со значением по умолчанию для переменных обобщенного типа. Поэтому для задания значения по умолчанию параметру типа используется ключевое слово `default (T)`. При этом значениям ссылочного типа присваивается `null`, а структурного `0`.

С переменными обобщенного типа нельзя выполнять арифметические действия и операции сравнения. Это связано с тем, что при инстанцировании вместо параметра типа может быть использован тип данных, который не поддерживает эти операции.

В качестве примера создадим обобщенный класс `Point2D<T>`, описывающий точку в двухмерном пространстве (Рисунок 6.9).

```
using static System.Console;

namespace SimpleProject
{
    /// <summary>
    /// Обобщенный класс точки
    /// </summary>
    /// <typeparam name="T">
    /// координаты точки могут быть любого типа
    /// </typeparam>

    public class Point2D<T>
    {

        // параметр типа используется для задания типа свойства
        public T X { get; set; }
        public T Y { get; set; }

        // параметр типа используется для задания типов
        // параметров метода

        public Point2D(T x, T y)
        {
            X = x;
            Y = y;
        }
    }
}
```

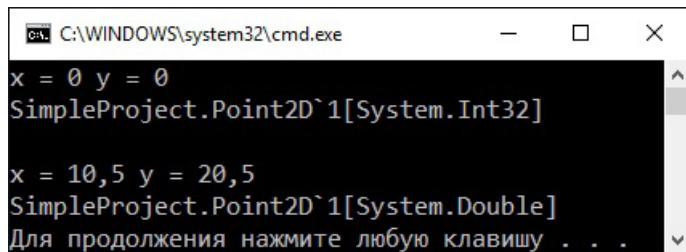
```

public Point2D()
{
    X = default(T);
    Y = default(T);
}
}

class Program
{
    static void Main(string[] args)
    {
        // тестирование обобщенного класса -
        // точки в 2D
        Point2D<int> p1 = new Point2D<int>();
        WriteLine($"x = {p1.X} y = {p1.Y}");
        WriteLine(typeof(Point2D<int>));

        Point2D<double> p2 =
            new Point2D<double>(10.5, 20.5);
        WriteLine($"x = {p2.X} y = {p2.Y}");
        WriteLine(typeof(Point2D<double>));
    }
}
}

```



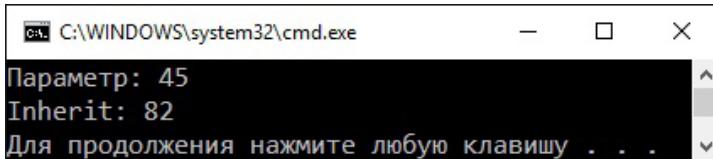
**Рисунок 6.9.** Пример работы с обобщенным классом Point2D<T>

Как видно из результатов выполнения программы, из generic-класса создаются классы для конкретных типов данных. Эти классы называются constructed type (сконструированный тип). Для объявления Point2D<int> создается класс с именем Point2D`1[System.Int32], для Point2D<double> — класс Point2D`1[System.Double]. Имея этих классов образуются по следующему правилу: Point2D — имя generic-класса, 1 — количество параметров типа, [имя типа] — тип, который использовался при специализации шаблона. Эти классы создаются из generic-класса Point2D<T> во время выполнения программы, когда JIT-компилятор впервые выполняет компиляцию метода, использующего переменную такого классы.

Generic-классы могут использоваться в качестве базовых классов. Следовательно, generic-классы могут иметь виртуальные и абстрактные методы. Однако необходимо соблюдать следующие правила наследования от generic-классов:

- если от generic-класса наследуется необобщенный класс — наследник должен конкретизировать параметр типа;
- при реализации generic виртуальных методов производный необобщенный класс должен конкретизировать параметр типа;
- если от generic-класса наследуется другой generic-класс, в нем необходимо учитывать ограничения типа, указанные в базовом классе.

Пример наследования от обобщенного класса представлен ниже (Рисунок 6.10).



**Рисунок 6.10.** Пример наследования от обобщенного класса

```
using static System.Console;

namespace SimpleProject
{
    class GenericClass<T>
    {
        public void M1(T obj)
        {
            WriteLine($"Параметр: {obj}");
        }

        public virtual void M2(T data)
        {
            WriteLine($"Generic: {data}");
        }
    }

    class InheritClass : GenericClass<int> // необходимо
                                              // явно указывать тип
    {

        public override void M2(int data)
        {
            WriteLine($"Inherit: {data}");
        }
    }

    class Program
    {
```

```
    static void Main()
    {
        InheritClass obj = new InheritClass();
        obj.M1(45);
        obj.M2(82);
    }
}
```

Наследование обобщенного класса от обычного класса не связано с какими-либо ограничениями (Рисунок 6.11).

```
using static System.Console;

namespace SimpleProject
{
    class BasicClass
    {
        protected int age;
    }

    class GenericClass<T> : BasicClass
    {

        public void M1(T obj)
        {
            age = 57;
            WriteLine($"Basic: {age}\nGeneric: {obj}");
        }
    }

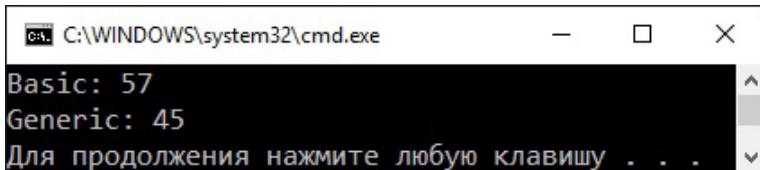
    class Program
    {

        static void Main()
    }
```

```

        GenericClass<int> obj =
            new GenericClass<int>();
        obj.M1(45);
    }
}
}

```



**Рисунок 6.11.** Пример наследования обобщенного класса

## Вложенные типы внутри generic-класса

Внутри generic-класса можно объявлять вложенные классы. Эти классы также будут являться обобщенными, даже если они не имеют собственных параметров типа. Во вложенных классах можно использовать параметр типа, указанный во внешнем классе. Во вложенном классе можно также объявлять собственный список параметров типа. В этом случае вложенный класс будет параметризирован двумя наборами типов: собственным и внешнего класса (Рисунок 6.12).

```

using static System.Console;

namespace SimpleProject
{
    class A<T>
    {

```

```
public class Inner
{
}

class B<T>
{
    // вложенный класс имеет собственный список
    // параметров типа
    public class Inner<U>
    {
    }
}

class Program
{
    static void Main(string[] args)
    {
        // для использования вложенного класса
        // необходимо указать реальный тип вместо
        // параметра типа внешнего класса
        A<int>.Inner a = new A<int>.Inner();
        WriteLine(a);
        A<double>.Inner a1 = new A<double>.Inner();
        WriteLine(a1);

        // для использования вложенного класса
        // необходимо указать реальный тип вместо
        // параметра типа вложенного класса
        B<int>.Inner<string> b =
            new B<int>.Inner<string>();
        WriteLine(b);
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
SimpleProject.A`1+Inner[System.Int32]
SimpleProject.A`1+Inner[System.Double]
SimpleProject.B`1+Inner`1[System.Int32, System.String]
Для продолжения нажмите любую клавишу . . .
```

**Рисунок 6.12.** Пример наследования  
от обобщенного класса

## Использование ограничений

Для параметра типа можно указать несколько ограничений, указывающие каким требованиям должен удовлетворять тип данных, используемый вместо этого параметра.

Список возможных ограничений представлен в таблице 6.2.

**Таблица 6.2.** Ограничения параметров типов

Ограничение обобщения	Описание
where T: struct	Параметр типа должен наследоваться от <code>System.ValueType</code> , т.е. быть структурным типом
Where T: class	Параметр типа <u>не должен</u> наследоваться от <code>System.ValueType</code> , т.е. быть ссылочным типом
where T: new()	Класс должен иметь конструктор по умолчанию ( <u>указывается последним</u> )
where T: BaseClass	Параметр типа должен быть производным классом от указанного базового класса
where T: Interface	Параметр типа должен реализовать указанный интерфейс

Синтаксис задания ограничения выглядит следующим образом:

```
class ИмяКласса<T> where T : ограничения
```

Например:

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

Эти ограничения требуют, чтобы тип, который используется вместо параметра Т, являлся ссылочным, реализовал интерфейс IComparable и имел конструктор по умолчанию.

Пропишем ограничения для класса Point2D<T>, созданного нами ранее.

```
public class Point2D<T> where T : struct
{
    // остальной код остался прежним
}
```

Хотя внесенные ограничения никак повлияли на итоговый результат (Рисунок 6.9), но за счет использования ограничений правильность и типобезопасность кода теперь проверяется на этапе компиляции. При попытке скомпилировать специализацию для конкретного типа, компилятор будет проверять, выполняются ли для этого типа указанные ограничения.

Если мы попытаемся при создании экземпляра класса Point2D<T> указать в качестве параметра типа ссылочный тип, например `string`, то возникнет ошибка на этапе компиляции (Рисунок 6.13).

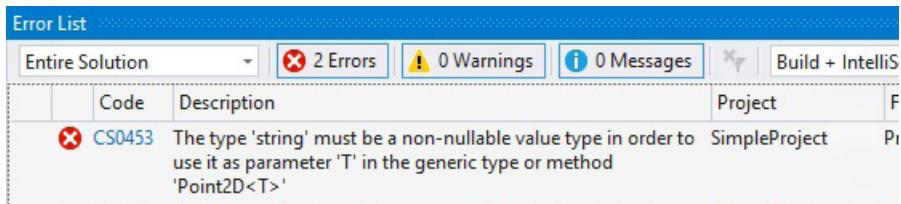
```

namespace SimpleProject
{
    public class Point2D<T> where T : struct
    {
        // остальной код остался прежним
    }

    class Program
    {

        static void Main(string[] args)
        {
            Point2D<string> point =
                new Point2D<string>("0", "0");
        }
    }
}

```



**Рисунок 6.13.** Ошибка: тип string  
не должен принимать null-значение

## Создание generic-методов

В некоторых случаях удобно иметь метод, параметризованный каким-то типом данных. Такой метод может находиться как в обобщенном, так и в необобщенном классе. При создании generic-метода параметр типа указывается в угловых скобках после имени метода.

В качестве примера приведем реализацию метода нахождения максимального элемента в одномерном массиве. Так как обобщенным методам, как и классам, можно задавать ограничения параметров типа, мы потребуем, чтобы тип элементов массива реализовал интерфейс IComparable. Потому что для поиска максимального элемента массива необходимо выполнять сравнение его элементов (Рисунок 6.14).

```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static T MaxElem<T>(T[] arr) where T : IComparable<T>
        {
            T max = arr[0];
            foreach (T item in arr)
            {
                if (item.CompareTo(max) > 0)
                    max = item;
            }
            return max;
        }

        static void Main(string[] args)
        {
            int[] arrInt = new int[] { 22, 63, 718, 14, 5 };

            // реальный тип для параметра типа
            // указывается явно
            WriteLine($"Максимальный элемент:
{MaxElem<int>(arrInt)}");
        }
}
```

```

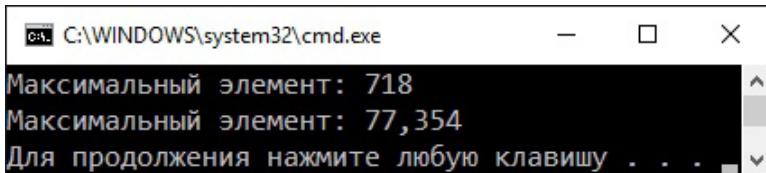
        double[] arrDouble = new double[] { 45.62,
                                            77.354, 18.48, 11.3 };

        // реальный тип определяется
        // по типу переданного массива
        WriteLine($"Максимальный элемент:
{MaxElem(arrDouble)}");

    }

}
}

```



**Рисунок 6.14.** Пример использования обобщенного метода

Как Вы могли заметить, если не указывать тип данных при вызове метода, то он определяется автоматически по типу параметров, переданных в метод. И хотя такой код не приводит к ошибке, рекомендуется явно указывать тип данных при вызове метода, чтобы облегчить читабельность Вашей программы.

## Generic-интерфейсы

Как уже говорилось, в пространстве имен `System.Collections.Generic` находятся обобщенные интерфейсы, которые соответствуют необобщенным интерфейсам, изученным Вами ранее (таблица 6.3).

**Таблица 6.3. Соответствие интерфейсов**

Generic-интерфейсы	Необобщенные интерфейсы
ILits<T>	ILits
IDictionary<T>	IDictionary
ICollection<T>	ICollection
IEnumerator<T>	IEnumerator
IComparer<T>	IComparer
IComparable<T>	IComparable

Теперь при необходимости реализации какого-либо интерфейса предпочтение следует отдавать обобщенным интерфейсам. Это не прихоть и не совет, а точный расчет. Чтобы доказать Вам это, мы возьмем пример из раздела №5 текущего урока (использование коллекции для хранения пользовательского типа), но теперь используем обобщенные интерфейсы и коллекцию для его реализации (Рисунок 6.15).

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    class DateComparer : IComparer<Student>
    {

        public int Compare(Student x, Student y)
        {
            return DateTime.Compare(x.BirthDate, y.BirthDate);
        }
    }
}
```

```

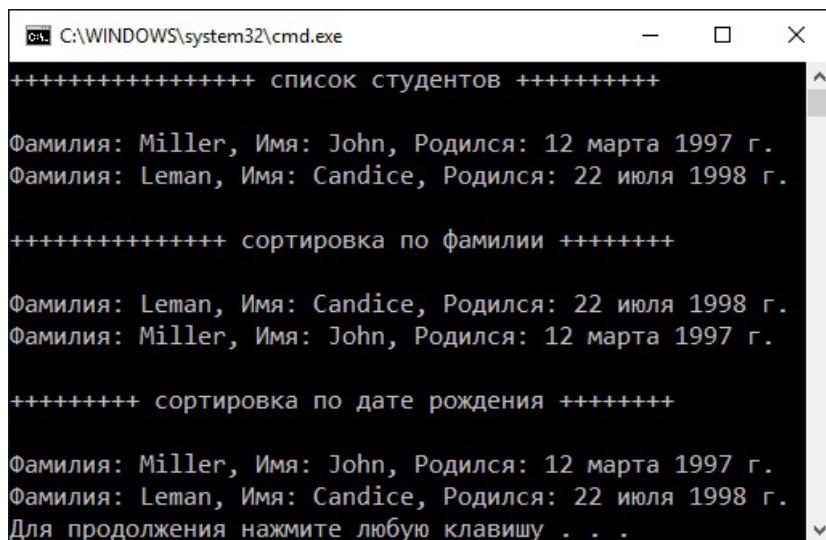
class Student : IComparable<Student>
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }
    public override string ToString()
    {
        return $"Фамилия: {LastName}, Имя: {FirstName},
               Родился: {BirthDate.ToString("yyyy-MM-dd")}";
    }
    public int CompareTo(Student other)
    {
        return LastName.CompareTo(other.LastName);
    }
}
class Program
{
    static void Main(string[] args)
    {
        List<Student> auditory = new List<Student> {
            new Student {
                FirstName ="John",
                LastName ="Miller",
                BirthDate =new DateTime(1997,3,12)
            },
            new Student {
                FirstName ="Candice",
                LastName ="Leman",
                BirthDate =new DateTime(1998,7,22)
            }
        };
        WriteLine("++++++ список студентов +++++\n");
        foreach (Student student in auditory)
        {
            WriteLine(student);
        }
    }
}

```

```
        WriteLine("\n+++++ сортировка по фамилии ++++\n");
        auditory.Sort();

        foreach (Student student in auditory)
        {
            WriteLine(student);
        }
        WriteLine("\n+++++ сортировка по дате рождения
                ++++++++\n");
        auditory.Sort(new DateComparer());

        foreach (Student student in auditory)
        {
            WriteLine(student);
        }
    }
}
```



**Рисунок 6.15.** Пример использования generics  
при работе с пользовательским типом

Как Вы видите, результат работы программы не отличается от полученных нами ранее (Рисунок 5.2), однако код стал более лаконичным, так как при реализации интерфейсов параметрами методов теперь является конкретный тип данных, а не `object`, что устраниет необходимость в дополнительных проверках.

При необходимости можно создавать собственные обобщенные интерфейсы. Обобщенные интерфейсы удобны тем, что их можно указать в качестве ограничений параметра типа при создании generic-класса, и при реализации этого класса использовать методы, объявленные в интерфейсе.

В качестве примера создадим generic-класс, в котором будет содержаться коллекция данных обобщенного типа, и в этом классе реализуем метод, который должен возвращать сумму элементов этой коллекции. Сумма должна иметь тот же тип, что и тип данных в коллекции.

Для того чтобы элементы коллекции можно было суммировать надо создать интерфейс, содержащий метод вычисления суммы, и указать этот интерфейс в качестве ограничения параметра типа.

```
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    /// <summary>
    /// Обобщенный интерфейс с методом вычисления суммы
    /// </summary>
    /// <typeparam name="T"></typeparam>
```

```
interface ICalc<T>
{
    T Sum(T b);
}

class Program
{

    /// <summary>
    /// Необобщенный класс, реализующий интерфейс ICalc
    /// </summary>
    class CalcInt : ICalc<CalcInt>
        //при наследовании указывается реальный тип данных
    {
        int _number = 0;
        public CalcInt(int n)
        {
            _number = n;
        }

        // при реализации методов вместо обобщенного типа
        // используется тип CalcInt
        public CalcInt Sum(CalcInt b)
        {
            return new CalcInt(_number + b._number);
        }

        public override string ToString()
        {
            return _number.ToString();
        }
    }

    /// <summary>
    /// Обобщенный класс, который содержит в себе
    /// коллекцию данных обобщенного типа
    /// и имеет метод вычисления суммы
    /// Для вычисления суммы задается ограничение:
}
```

```

/// параметр типа должен реализовать
/// интерфейс ICalc<T>
/// </summary>
/// <typeparam name="T"></typeparam>
class MyList<T> where T : ICalc<T>
{
    // коллекция данных обобщенного типа
    List<T> list = new List<T>();
    // метод добавления данных в коллекцию

    public void Add(T t)
    {
        list.Add(t);
    }
    // метод вычисления суммы

    public T Sum()
    {
        if (list.Count == 0)
        {
            return default(T);
        }
        T result = list[0];
        // для суммирования используется метод
        // интерфейса ICalc<T>
        for (int i = 1; i < list.Count; i++)
        {
            result = result.Sum(list[i]);
        }
        return result;
    }
}

static void Main(string[] args)
{
    MyList<CalcInt> myList = new MyList<CalcInt>();
    myList.Add(new CalcInt(10));
}

```

```
myList.Add(new CalcInt(20));
myList.Add(new CalcInt(23));
WriteLine($"Сумма элементов коллекции:
{myList.Sum()}");
}
}
}
```

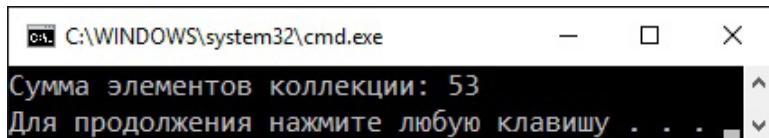


Рисунок 6.16. Использование generic-интерфейса

# 7. Итераторы

## Что такое итератор?

Как Вы уже знаете, для того чтобы класс-коллекция, мог перечислять свои элементы при помощи конструкции `foreach`, он должен реализовать интерфейс `IEnumerable`. Однако существует еще один способ создания типов, которые могут работать с конструкцией `foreach`, без реализации интерфейса `IEnumerable`. Для этого необходимо реализовать в конкретном классе-коллекции метод, который определяет, каким образом будут передаваться элементы в конструкцию `foreach`, такой метод и называется итератором.

## Синтаксис и примеры использования итераторов

Для реализации итератора используется ключевое слово `yield`, которое предназначено для возврата значений в конструкцию `foreach` вызывающего кода с использованием синтаксиса `yield return`. При этом текущее значение сохраняется, и последующий вызов итератора будет продолжаться уже с этого места. Пример использования итераторов приведен ниже (Рисунок 7.1).

```
using System;
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
```

```
class Student
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public DateTime BirthDate { get; set; }

    public override string ToString()
    {
        return $"Фамилия: {LastName}, Имя: {FirstName},
               Родился: {BirthDate.ToString("yyyy-MM-dd")}";
    }
}

class Auditory
{
    List<Student> _auditory = new List<Student> {
        new Student {
            FirstName ="John",
            LastName ="Miller",
            BirthDate =new DateTime(1997,3,12)
        },
        new Student {
            FirstName ="Candice",
            LastName ="Leman",
            BirthDate =new DateTime(1998,7,22)
        },
        new Student {
            FirstName ="Joey",
            LastName ="Finch",
            BirthDate =new DateTime(1996,11,30)
        },
        new Student {
            FirstName ="Nicole",
            LastName ="Taylor",
            BirthDate =new DateTime(1996,5,10)
        }
    };
}
```

```

public IEnumerator<Student> GetEnumerator()
{
    for (int i = 0; i < _auditory.Count; i++)
    {
        yield return _auditory[i];
    }
}

class Program
{
    static void Main(string[] args)
    {
        Auditory auditory = new Auditory();

        WriteLine("+++++ список студентов +++++\n");

        foreach (Student student in auditory)
        {
            WriteLine(student);
        }
    }
}
}

```

```

C:\WINDOWS\system32\cmd.exe
+++++ список студентов +++++

Фамилия: Miller, Имя: John, Родился: 12 марта 1997 г.
Фамилия: Leman, Имя: Candice, Родился: 22 июля 1998 г.
Фамилия: Finch, Имя: Joey, Родился: 30 ноября 1996 г.
Фамилия: Taylor, Имя: Nicole, Родился: 10 мая 1996 г.
Для продолжения нажмите любую клавишу . . .

```

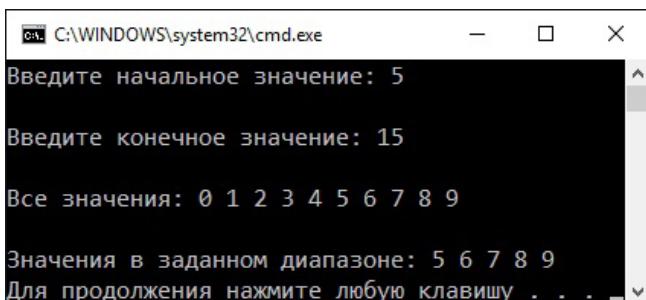
**Рисунок 7.1.** Пример использования итераторов  
в классе-коллекции

Если в классе-коллекции есть необходимость использования более одного итератора, тогда существует возможность создания, так называемых именованных итераторов. Это методы, типом возврата которых должен быть интерфейс `IEnumerable`. Именованный итератор имеет следующую общую форму записи:

```
public IEnumerable имя_итератора(список _параметров)
{
    // необходимый код
    yield return возвращаемое _значение;
}
```

Итераторы могут использоваться не только при работе с коллекциями, они также могут применяться в любом классе, чтобы возвращать в конструкцию `foreach` элементы полученные динамически.

В качестве примера создадим класс `NamedIterator`, в котором именованный итератор используется для вывода целочисленных значений в указанном диапазоне, а обычный итератор возвращает значения от нуля до указанного конечного значения. Мы искусственно



**Рисунок 7.2.** Пример использования итераторов в классе

ограничили количество выводимых значений. Для этой цели мы применили конструкцию `yield break`, которая используется, если существует необходимость в преждевременном прерывании итератора. Возможный результат работы программы представлен на рисунке 7.2.

```
using System.Collections.Generic;
using static System.Console;

namespace SimpleProject
{
    class NamedIterator
    {
        const int LIM = 10;
        int _limit;
        public NamedIterator(int limit)
        {
            _limit = limit;
        }
        public IEnumarator<int> GetEnumerator()
        {
            for (int i = 0; i < _limit; i++)
            {
                if (i == LIM)
                {
                    yield break; // прерывание итератора
                                  // по условию
                }
                yield return i;
            }
        }

        public IEnumarable<int> GetRange(int start)
        {
            for (int i = start; i <= _limit; i++)
            {
```

```
        if (i == LIM)
    {
        yield break;
    }
    yield return i;
}
}

class Program
{
    static void Main(string[] args)
    {
        Write("Введите начальное значение: ");
        int start = int.Parse(ReadLine());

        Write("\nВведите конечное значение: ");
        int end = int.Parse(ReadLine());
        NamedIterator namedIterator =
            new NamedIterator(end);

        Write("\nВсе значения: ");
        foreach (int item in namedIterator)
        {
            Write($"{item} ");
        }

        Write("\n\nЗначения в заданном диапазоне: ");
        foreach (int item in
            namedIterator.GetRange(start))
        {
            Write($"{item} ");
        }
        WriteLine();
    }
}
```

# Домашнее задание

Создать примитивный англо-русский и русско-английский словарь, содержащий пары слов — названий стран на русском и английском языках. Пользователь должен иметь возможность выбирать направление перевода и запрашивать перевод.

Создать необобщенный класс точки в трехмерном пространстве с целочисленными координатами (Point3D), который наследуется от generic-класса Point2D<T>, рассмотренного в уроке. Реализовать в классе:

- конструктор с параметрами, который принимает начальные значения для координат точки
- метод ToString()

Создать обобщенный класс прямой на плоскости. В классе предусмотреть 2 поля типа обобщенной точки — точки, через которые проходит прямая. Реализовать в классе:

- конструктор, который принимает 2 точки
- конструктор, которые принимает 4 координаты (x и y координаты для первой и второй точки)
- метод ToString()

Подсчитать, сколько раз каждое слово встречается в заданном тексте. Результат записать в коллекцию Dictionary< TKey, TValue >



## Урок №8

# Использование коллекций

© Юрий Задерей.

© Компьютерная Академия «Шаг»  
[www.itstep.org](http://www.itstep.org).

Все права на охраняемые авторским правом фото-, аудио- и видеопропизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.