Microsoft .Net Framework and C# Programming Language

Lesson 8

The use of collections

Contents

1. The concept of the collection

A collection is a set of objects grouped according to a certain feature. In previous lessons, you used arrays, which are also collections, but static, for storing objects. Therefore, it is virtually impossible to change the size of an array during the program operation, which imposes restrictions when working with them.

In addition to the arrays in .NET space, there are a large number of classes that represent dynamic collections, we will give their brief classification.

2. Discussion of existing collections

The main namespace for all collection classes is `System.Collections`. In this namespace there are non-generic collections, which store data of `object` type. In the same namespace, there is a `BitArray` collection that contains the values of bits represented by the `bool` type and supports bitwise operations.

There are also a number of collections in nested namespaces. The `System.Collections.Generic` namespace contains generic collections in which one can store data compatible with the specified type. Collections that support multithreaded access are contained in the `System.Collections.Concurrent` namespace. The `System.Collections.Specialized` namespace contains several classes of special collections, which work in a special way. Here is a brief description of each of them:

- `System.Collections.Specialized.ListDictionary` — implements a dictionary using a unidirectional list, the best performance is achieved if the number of elements is less than ten;
- `System.Collections.Specialized.HybridDictionary` — supports switching between two dictionaries depending on the number of elements. If the number of elements does not exceed ten, then the `ListDictionary` is used,

with an increase in the number of elements, an automatic switch to the `Hashtable` occurs;

- `System.Collections.Specialized.CollectionsUtil` — using this class, one can create collections that will ignore string case;
- `System.Collections.Specialized.NameValueCollection` — is a dictionary in which both the key and the value are represented by the `string` type, with several values corresponding to one key;
- `System.Collections.Specialized.StringCollection` — is a collection containing elements of `string` type, whose values can be repeated and have the `null` value;
- `System.Collections.Specialized.StringDictionary` — is a dictionary whose key and value are represented by the `string` type, the key, unlike the value, cannot be `null`.

Now let's move on to a more detailed examination of classes of non-generic collections.

3. Classes of the ArrayList, Stack, Queue, Hashtable, SortedList collections

As it was said earlier, non-generic collections allow storing data of `object` type, that is, elements of this type of collections can be of different types.

The most commonly used non-generic collection is the `ArrayList`. The `ArrayList` class is able to dynamically change its size when adding or removing elements. When you create this type of collection, you can use one of three constructors, in one of them you can explicitly set the collection capacity:

```
ArrayList arrayList1 = new ArrayList();  // default capacity
ArrayList arrayList2 = new ArrayList(5); // initial capacity
                                         // value elements
// are copied from the specified collection
ArrayList arrayList3 = new ArrayList(new int[] { 1, 5, 48 });
```

Specifying the capacity, you specify the possible number of elements in the collection. The default capacity of the `ArrayList` is zero, if you add at least one element to the collection, the capacity becomes equal to four. If the number of the elements added exceeds the current capacity value, the latter will automatically be doubled. The current capacity can be set or retrieved using the Capacity property. Calling the `TrimToSize ()` method reduces the `ArrayList` capacity to the actual number of its elements, which the Count

property returns. Adding elements to this collection is done using the `Add(object)` and `AddRange(ICollection)` methods — the first is used to add a single object, and the second is used for the collection of objects. Access to the elements of this collection can be carried out by index. Demonstration of the above described functionality of the `ArrayList` collection is presented below (Figure 3.1).

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

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

            arrayList1.Add("one");

            WriteLine($"The capacity after adding one element:
                    {arrayList1.Capacity}");

            arrayList1.AddRange(new int[] { 2, 5, 48, 14 });

            WriteLine($"Capacity after adding a collection:
                    {arrayList1.Capacity}");

            arrayList1.Capacity = 10;

            WriteLine($"The capacity is given through
                    the property: {arrayList1.Capacity}");

            WriteLine($"Actual number of elements:
                    {arrayList1.Count}");

            arrayList1.TrimToSize();
```

```
            WriteLine($"Capacity is reduced to the actual
                      number of elements:
                      {arrayList1.Capacity}");

            WriteLine($"Element with index 2:
                      {arrayList1[2]}");

            WriteLine("All elements of the collection:");
            foreach (object item in arrayList1)
            {
                WriteLine($"\t{item}");
            }
        }
    }
}
```

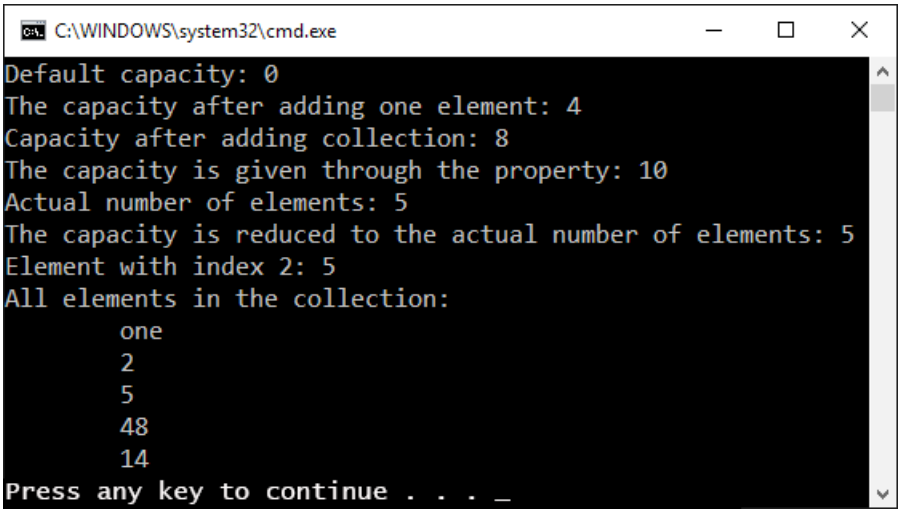

Figure 3.1. Functionality of the ArrayList collection

When you add elements to the `ArrayList` collection using the `Add(object)` method, they are added to the end of the collection. However, you can add items to a specified location in the collection by using the `Insert(int, object)`

method, the first parameter of which is the index where to add the new element, and the second parameter is the value of this element. Similarly, you can add a range of values by calling the `InsertRange(int, ICollection)` method, the second parameter of which is used to pass the collection.

To remove elements of the `ArrayList` collection, you use the same methods `RemoveAt(int)` and `RemoveRange(int, int)`.

The `GetRange(int, int)` method of the `ArrayList` collection allows you to create a new collection with the values of the elements taken in the specified range from the original collection. The first parameter of the `GetRange(int, int)` method specifies the index of the beginning of the range, the second — the number of elements to copy.

To find the first occurrence of an object in a collection, use `IndexOf(object)` methods, and for the last — `LastIndexOf (object)`. The `Sort()` method sorts the elements of the collection. The use of these methods is presented below (Figure 3.2).

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

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            Write("The initial collection: ");
            ArrayList arrayList = new ArrayList(new int[]
                                        { 1, 2, 3, 4 });
```

```
foreach (int i in arrayList)
{
    Write($"{i} ");
}

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

Write("\n\nDeleting an element: ");
arrayList.RemoveAt(3);
foreach (object item in arrayList)
{
    Write($"{item} ");
}

WriteLine("\n\nThe index of the \"Hello\"
      element: ":" + arrayList.IndexOf("Hello"));
Write("\nGetting a range: ");
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\nSorting a collection: ");
onlyWork.Sort();
```

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

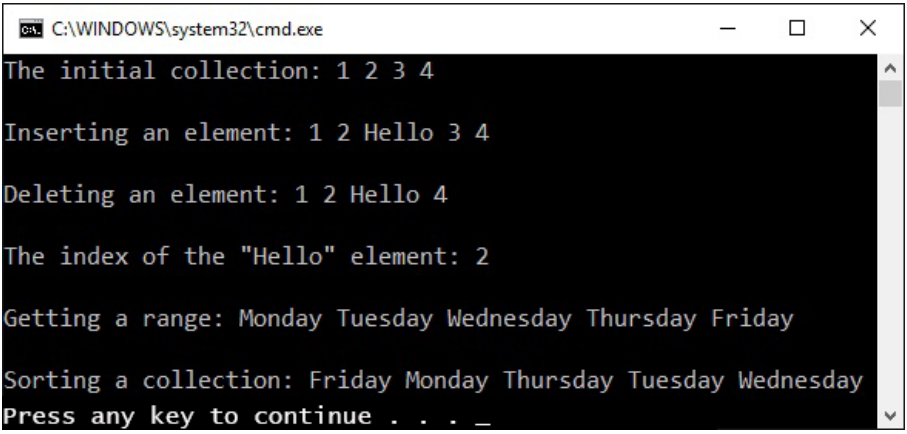

Figure 3.2. ArrayList collection methods

When calling the `Sort()` method, note that the `ArrayList` collection can contain elements of different data types, in this case an attempt to sort the collection will result in an error at the runtime (Figure 3.3).

```
using System.Collections;

namespace SimpleProject
{
    class Program
    {
```

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

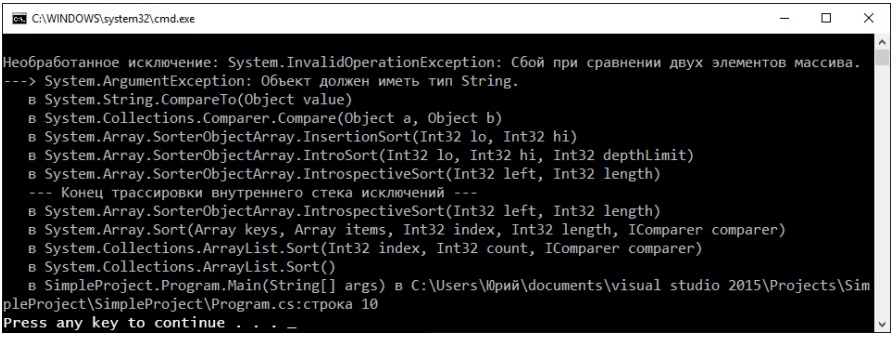

Figure 3.3. Runtime error

Consider the following collection — `Stack`. The `Stack` class is a collection that operates on the LIFO principle (last-in-first-out), that is, the sequence of object extraction is opposite to the sequence of their placement in the collection.

To create this collection type, there are three constructors:

```
Stack stack1 = new Stack(); // default capacity
Stack stack2 = new Stack(7); // initial capacity value
// elements are copied from the specified collection
Stack stack3 = new Stack(new ArrayList { 3, 5 });
```

You cannot use an index to refer to an element in the `Stack` collection. The elements are placed in this collection using

the `Push(object)` method, but are retrieved using the `Pop()` method. If you want to get the value of the last element in the collection without deleting it, you must use the `Peek()` method. To copy the contents of a collection to an array, use the `CopyTo(Array, int)` method. A check for a specified value in the `Stack` collection is performed using the Contains property, which returns `true` in the case of a positive result and `false` in the case of a negative result. To clear the contents of the collection, use the `Clear()` method. The results of working with the `Stack` collection are shown in Figure 3.4.

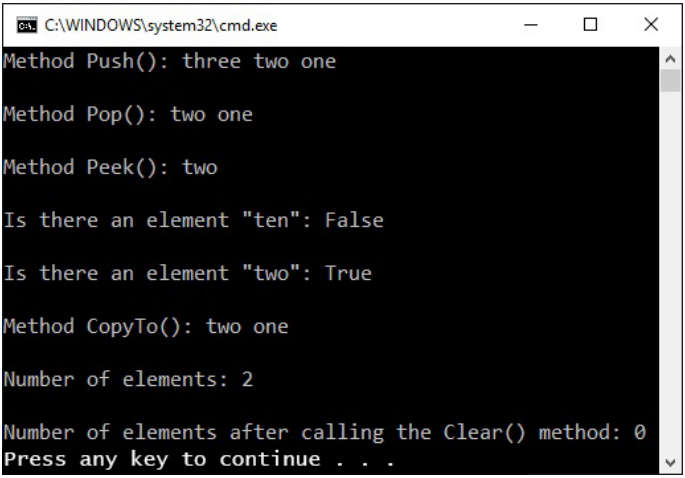

Figure 3.4. Working with the Stack Collection

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

namespace SimpleProject
{
    class Program
    {
```

```
static void Main(string[] args)
{
    Stack stack = new Stack();
    Write("Method Push(): ");
    stack.Push("one");
    stack.Push("two");
    stack.Push("three");

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

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

    WriteLine($"nMethod Peek():
              {(string)stack.Peek()}");
    WriteLine("\nIs there an element \"ten\" exist: "+
              stack.Contains("ten")); // false
    WriteLine("\nIs there an element \"two\" exist: " +
              stack.Contains("two")); // true
    Write("\nMethod CopyTo(): ");
    string[] strArr = new string[stack.Count];
    stack.CopyTo(strArr, 0);
    foreach (string item in strArr)
    {
        Write($"{item} ");
    }

    WriteLine("\n\nNumber of elements: " +
              stack.Count); // 3
```

```
            stack.Clear();

            WriteLine("\nNumber of the elements after calling
                      the Clear() method: " +
                      stack.Count); // 0
        }
    }
}
```

The `Queue` collection works on the FIFO principle (first-in-first-out), which means that the elements are extracted from the collection in the same sequence in which they were placed in it. Indexing does not apply to the elements of this collection.

When creating a collection, a real value is set — the growth factor, by default it is 2.0, but there is a constructor in which this factor can be set explicitly in the range from 1.0 to 10.0:

```
Queue queue1 = new Queue(); // default capacity
Queue queue2 = new Queue(3); // initial capacity value
// elements are copied from the specified collection
Queue queue3 = new Queue(new ArrayList { "one", 8.4 });
// initial capacity value and growth factor
Queue queue4 = new Queue(7, 3.0f);
```

The basic methods of the queue are the `Enqueue(object)` and `Dequeue()` methods to place the elements into the collection and retrieve them accordingly. The `Peek()` method returns the value of the element that was first placed into the collection without deleting it. Demonstration of work with the `Queue` collection is presented below (Figure 3.5).

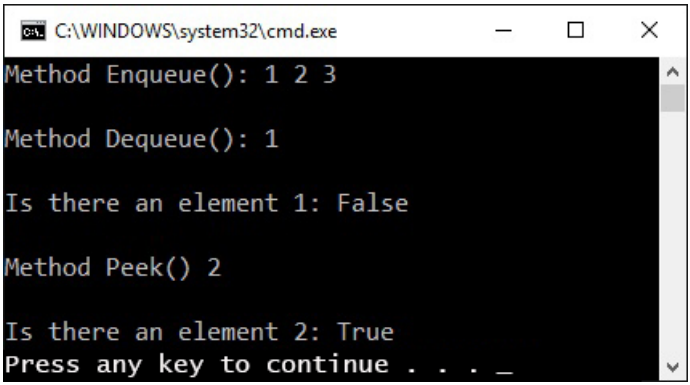

Figure 3.5. Queue collection methods

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

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

            foreach (int item in queue)
                Write($"{item} ");
            WriteLine($"\n\nMethod Dequeue():
                      {queue.Dequeue()}\n");
            WriteLine($"Is there an element 1:
                      {queue.Contains(1)}\n");
```

```
            WriteLine($"Method Peek() {queue.Peek()}\n");
            WriteLine($"Is there an element 2:
                    {queue.Contains(2)}");
        }
    }
}
```

The `Hashtable` collection is a dictionary, each element of which consists of a key-value pair, and the key value must be unique. A hash table is used to store the elements of this collection, and there the values are stored according to the corresponding key. The keys are stored in the form of a hash code, which is calculated when writing to the collection. A hash code is a kind of unique integer value that is calculated by a specific algorithm, encapsulated in the implementation of the `GetHashCode()` method for a particular type. So that's why you need to override the `GetHashCode()` method in your class!

Due to this approach, the duration of the search, record and delete operations does not depend on the amount of data in the collection. Another advantage of hashing is that keys can have different data types, for example one key-value pair can be an `int-string`, and another – a `string-object`, and so on.

When creating a dictionary (there are 15 constructors), you can explicitly specify its capacity, and it is recommended to use simple numbers for this, which is related to the algorithm of the dictionaries, so they work more efficiently. Also, when creating a `Hashtable`, you can specify a fill ratio between 0.1 and 1.0, which affects the process of increasing the size of the hash table — the table expands if the number of elements

becomes larger than the capacity of the table multiplied by this factor (the default is 1.0).

It is impossible to determine the exact place in the `Hashtable` where a new record will be placed when adding (the `Add(object, object)` method), since its location is calculated based on the hash code of the key, and it is not interesting to us in principle; the main is the speed of getting data from the collection. Deleting an entry is done using the `Remove(object)` method with the key specified.

The dictionary is characterized by a number of properties and methods that are absent in the usual collection. For example, using the Keys and Values properties, you can get collections of keys or values of the current dictionary, respectively. And the `ContainsKey(object)` and `ContainsValue(object)` methods let you determine whether the current `Hashtable` contains the specified key or value, respectively. An example of working with `Hashtable` is presented below (Figure 3.6).

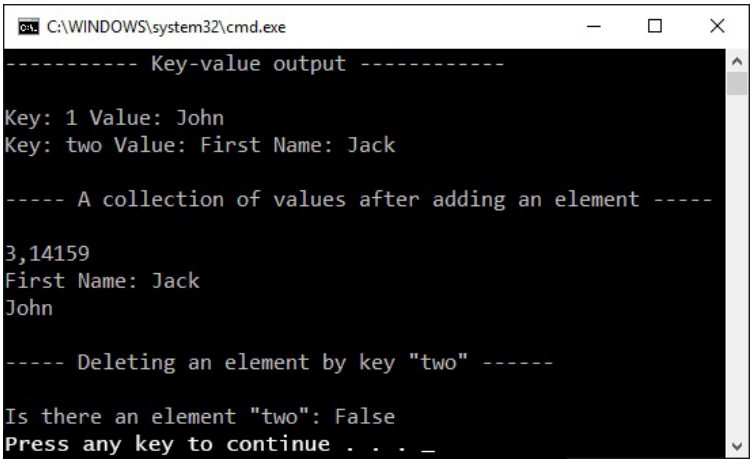

Figure 3.6. Hashtable dictionary methods

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

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

    class Program
    {
        static void Main(string[] args)
        {
            Hashtable hash = new Hashtable();
            hash.Add(1, "John");
            hash.Add("two", new Student { Name = "Jack" });
            WriteLine("----- Key-value output -------\n");
            foreach (object item in hash.Keys)
            {
                WriteLine("Key: " + item + " Value: "
                          + hash[item]);
            }
            hash.Add("Pi", 3.14159);
            WriteLine("\n----- A collection of values after
                       adding an element -----\n");
            foreach (object item in hash.Values)
                WriteLine(item);
            WriteLine("\n----- Deleting an element by the key
                       \"two\" ------\n");
            hash.Remove("two");
```

```
            WriteLine($"Is there an element \"two\":
                        {hash.ContainsKey("two")}"); // false
        }
    }
}
```

The `SortedList` collection is a dictionary whose objects are sorted by key automatically when a new element is added; access to the elements can be obtained either by key or by index.

To create `SortedList`, you can use one of the six available constructors. The default capacity of the dictionary is zero, after adding the first element it becomes 16. If the actual number of elements exceeds this value, the capacity automatically doubles.

The `SortedList` methods are similar to the methods of the `Hashtable` class, but there is a significant difference between these collections — the `SortedList` keys must be of the same data type, otherwise the dictionary cannot be sorted. The possible work with `SortedList` is shown in Figure 3.7.

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

namespace SimpleProject
{
    class Student
    {
        public string First Name { get; set; }
        public override string ToString()
        {
            return $"First Name: {First 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("----- Key-value output ------\n");
            foreach (object item in sortedList.Keys)
            {
                WriteLine($"Key: {item}
                           Value: {sortedList[item]}");
            }

            WriteLine("\n------- Key-value output by index
                       --------\n");

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

            WriteLine("\n--- Value collection ---\n");
            foreach (object item in sortedList.Values)
                WriteLine(item);

            WriteLine("\n----- Deleting an element
                       by the key 3 ------\n");
            sortedList.Remove(3);

            WriteLine($"Is there an element 3 exist:
                     {sortedList.ContainsKey(3)}"); // false
        }
    }
}
```

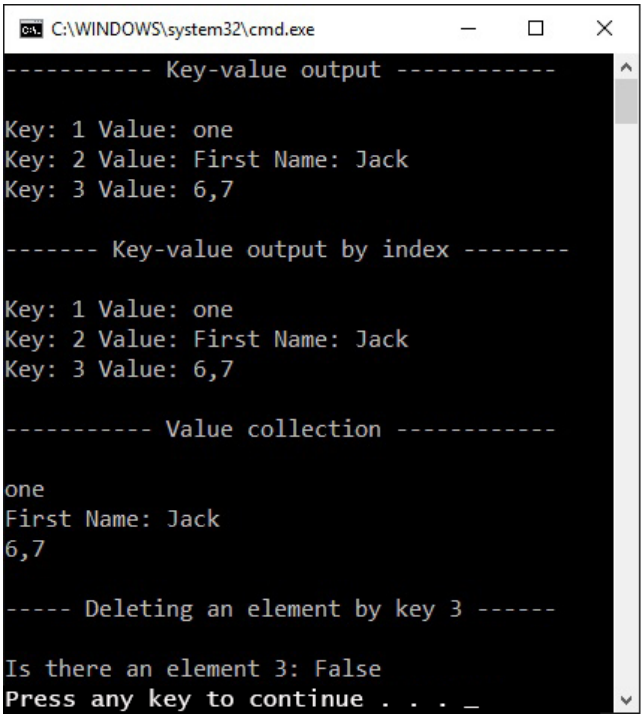

Figure 3.7. Using the SortedList dictionary

4. ICollection, IList, IDictionary, IDictionaryEnumerator Collection Interfaces

The `System.Collections` namespace contains interfaces that define the functionality associated with managing collection elements. Some of them have already been considered by us in the lesson 6, in this section we will give a brief description of a number of key interfaces.

The `ICollection` interface is implemented by any collection and defines a set of properties for: determining the number of elements in the collection, copying the content, and ensuring stream safety.

The `IList` interface provides the basic functionality for list collections (write, insert, delete, search, clear). It also allows you to implement the indexer and properties, with which you can find out whether the current collection is read-only and whether its size is fixed.

The `IDictionary` interface is the basic interface for all dictionary collections and defines all the basic functionality for working with this type of collection.

The `IDictionaryEnumerator` Interface allows you to enumerate the contents of a collection that implements the `IDictionary` interface, all functionality is read-only.

To learn more details, you can use the ObjectBrowser window or MSDN.

5. Examples of using collection classes for storing standard and custom types

You have already seen some examples of using collections in Section 3, in this section we show how collections can be used to solve more complex problems.

In the first example, we will create the `SortedArrayList` class, derived from the `ArrayList`, which will be an automatically sorted collection of elements. In this class, we define a method to add elements to the collection so that they are immediately sorted, and a method for changing the value of the element by a given index, which also guarantees the arrangement of the elements in the correct order. An example of using the `SortedArrayList` class is shown below (Figure 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("----- Initial values -------\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----- Changing the values -----\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();
        }
    }
}
```

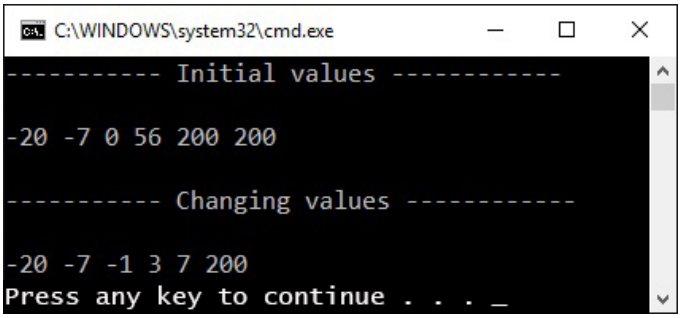

Figure 5.1. Using the SortedArrayList class

In the following example, we demonstrate the use of the `ArrayList` collection to store a custom type, in this case it will be the `Student` class that we created in the lesson 6. After making certain adjustments to the existing code, we got the following result (Figure 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 $"Last Name: {LastName},
                 First Name: {FirstName},
                 Born:{BirthDate.ToShortDateString()}";
    }

    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("++++ list of students ++++\n");
            foreach (Student student in auditory)
            {
                WriteLine(student);
            }
            WriteLine("\n++++ sorting by last name +++\n");
            auditory.Sort();
            foreach (Student student in auditory)
            {
                WriteLine(student);
            }
            WriteLine("\n++ sorting by date of birth +++\n");
            auditory.Sort(new DateComparer());
            foreach (Student student in auditory)
            {
                WriteLine(student);
            }
        }
    }
}
```

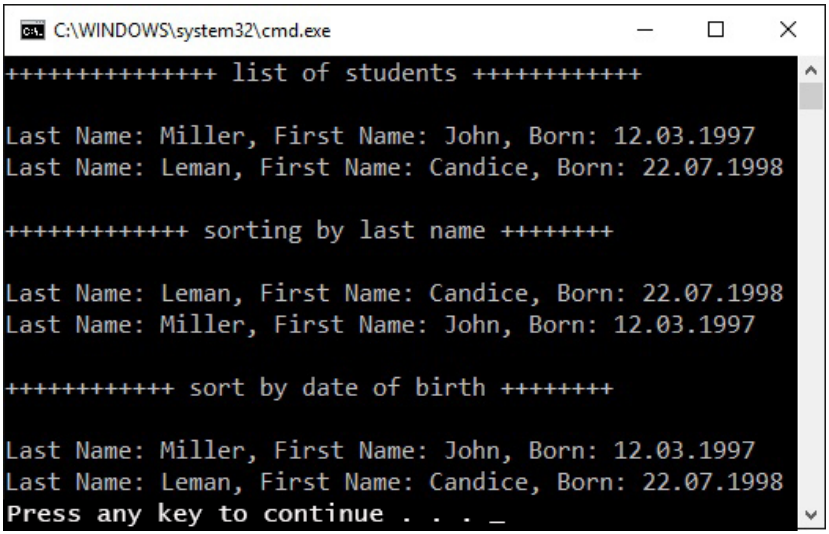

Figure 5.2. Example of storing a custom type in the ArrayList collection

Another example demonstrates the possibility of using the `Hashtable` dictionary to store information about the grades gained by students. The key for each record is an instance of the `Student` class, and the corresponding value is the `ArrayList` collection — a list of grades. To implement this example, you need to override the `GetHashCode()` method in the `Student` class, the result is shown in Figure 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("++++++++++++++ List of students
                       with grades ++++++++++\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++++++++++++++ Student {name}
                      got the grade {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();
        }
    }
}
```

As you noticed the `Student` class has become much simpler, this was done only to save space, you can check for yourself that with a more complex implementation of the class the example will work similarly.

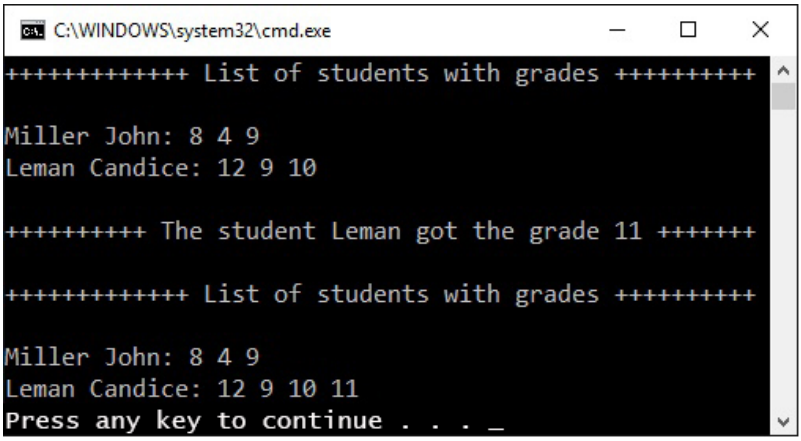

Figure 5.3. Using the Hashtable dictionary

6. Generics

What is generics?

Since the .Net Framework version 2.0, the platform supports generics, which provide the ability to create different constructs for a particular data type, which is specified as a parameter. Based on this, you can meet two more possible names — a parameterized type and a typed object, for example, a typed collection.

The following can be generic:

- collections;
- classes;
- structures;
- methods;
- interfaces;
- delegates.

Generic types are very easy to recognize, they are distinguished by the presence of angle brackets (<>) after the type name, within which a particular type or data types are specified.

The need to use generics. Boxing, unboxing

Introduction of generic types can be regarded as a stage of the evolution of the C# language, provoked by two important problems that arise when working with non-generic collections — low productivity and type security.

The problem of type safety is related to the universality of non-generic collections — storing data of `object` type. In some cases, this flexibility is convenient when you need to

put variables of any type in the collection. However, when extracting data from the collection, it is necessary to perform an explicit type cast, and you can easily commit the cast error, which will only appear at the runtime. Therefore, work with non-generic collections should still be accompanied by writing additional code of checks and casts — the `try-catch` block or the `is` or `as` operators (Figure 6.1).

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

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrayList = new ArrayList();
            // we put the elements of the type int
            // in the collection
            arrayList.Add(10);
            try
            {
                // when extracting, casting to the short
                // type throws an exception due
                // to an incorrect type indication
                short a = (short)arrayList[0];
            }
            catch (InvalidCastException e)
            {
                WriteLine(e.Message);
            }
        }
    }
}
```

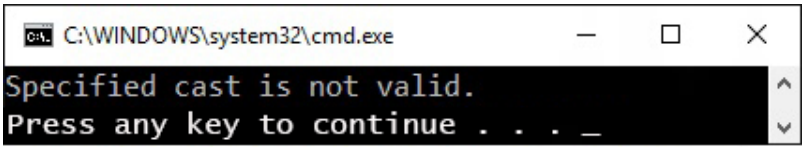

Figure 6.1. Wrong type cast

Low performance when working with non-generic collections is associated with the mechanism of the C# language, the so-called boxing-unboxing, which is started by the CLR without the participation of the programmer.

As you know, in C#, there are two types of data: reference and value. When you initiate a reference type with a value, the boxing process automatically takes place. Getting a value from a reference variable by a value type triggers a reverse mechanism, unboxing. Let's illustrate this with an example (Figure 6.2).

Figure 6.2. Type boxing-unboxing

```
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
```

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

You should pay attention to the need to implement explicit type cast, otherwise the compilation error will be generated (Figure 6.3).

```
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            object obj = 45;
            WriteLine($"Boxing: {obj}");
            int number = obj;
            WriteLine($"Unboxing: {number}");
        }
    }
}
```

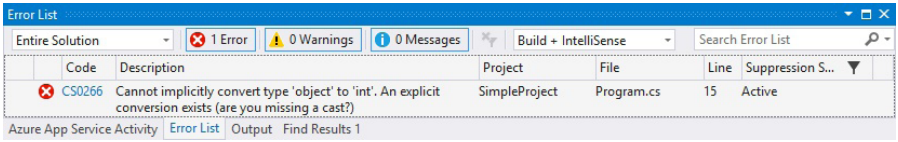

Figure 6.3. Error: cannot convert type 'object' to 'int'

In non-generic collections, by definition, `objects` type elements are stored, and if this kind of collections is used to store value types, then boxing-unboxing of types accompanies the whole process of working with such collections (Figure 6.4).

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

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            ArrayList arrayList = new ArrayList();
            // when an element is written to a collection,
            // the type int is cast to object — BOXING
            arrayList.Add(10);
            arrayList.Add(12);

            // when extracting, perform casting of type —
            // UNBOXING
            int a = (int)arrayList[0];

            // output of the value — int is cast
            // to string — BOXING
            Write($"The first element of the collection:
                  {a}"); //

            WriteLine("\n\nAll elements of the collection:");
            foreach (int item in arrayList)
            {
                WriteLine($"\t{item}"); // BOXING
            }
        }
    }
}
```

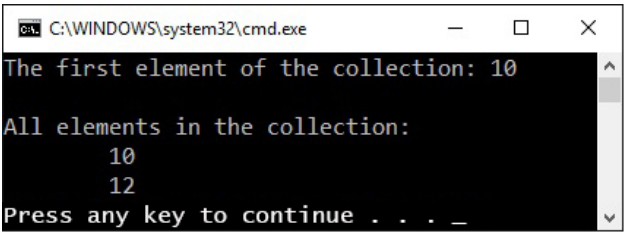

Figure 6.4. Boxing-Unboxing in non-generic collections

The use of generic types gives the following advantages:

- type safety — only objects of a certain type specified when the pattern is expanded can be placed in the generic collection;
- more simple and understandable code, since it is not necessary to perform type casting;
- improved performance — using generics, structured types are passed by value, boxing and unboxing are not performed.

Comparative analysis of generic collections and non-generic collections

Generic collections appeared with the advent of the concept of generics in FCL; they correspond to almost all classes of non-generic collections. These collections, as well as generic interfaces, are in the `System.Collections.Generic` namespace.

Table 6.1 presents generic and corresponding non-generic classes of collections.

The main difference between generic collections and non-generic collections (besides not always matching names) is that a generic parameter of the type, which is specified in angle brackets after the collection name, is used in generic

Таблица 6.1. Соответствие классов коллекций

Generic-collections	Non-generic collections
`Collection<T>`	CollectionBase
`List<T>`	ArrayList
`Dictionary<TKey, TValue>`	Hashtable
`SortedList<TKey, TValue>`	SortedList
`Stack<T>`	Stack
`Queue<T>`	Queue
`LinkedList<T>`	Нет

collections instead of the `object` data type. As a result, those methods and properties that used the `object` data type in the non-generic collection use a specific data type in the generic collection.

Let's demonstrate the type safety when working with generic collections on the example of the `List<int>` collection. Attempting to put a value in it, the type of which differs from the type of the generic parameter, will lead to a compilation error (Figure 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);
        }
    }
}
```

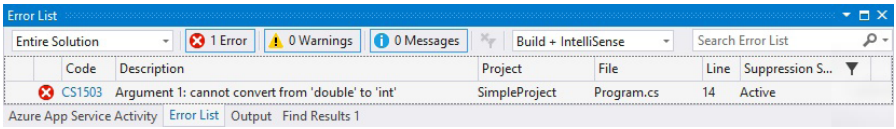

Figure 6.5. Error: cannot convert type 'double' to 'int'

To compare the time spent for storing structural types in a non-generic and generic collection, we give an example, proposed by Jeffrey Richter in the book 'The CLR via C#'.

To perform the profiling, we use the helper class `OperationTimer`, whose purpose is to accurately measure the execution time of the code section, and also to calculate the number of garbage collections. This class is a good pattern and can be used for profiling in your own projects.

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

namespace SimpleProject
{
    /// <summary>
    /// The auxiliary class for profiling the code section
    /// performs runtime measurements
    /// and counts the number of garbage collections
    /// </summary>
    internal sealed class OperationTimer : IDisposable
    {
        long _startTime;
        string _text;
        int _collectionCount;

        public OperationTimer(string text)
        {
```

```
            PrepareForOperation();
            _text = text;
            // the number of garbage collections that
            // have been completed so far is saved
            _collectionCount = GC.CollectionCount(0);
            // the initial time is preserved
            _startTime = Stopwatch.GetTimestamp();
        }

        /// <summary>
        /// called when the object is destroyed
        /// Outputs:
        /// The value of the time elapsed from the moment
        /// of creation of the object
        /// to the time it was deleted, the number
        /// of garbage collections performed during
        /// this time
        /// </summary>

        public void Dispose()
        {
            WriteLine($"{_text}\t{(Stopwatch.GetTimestamp() -
             _startTime) / (double)Stopwatch.Frequency:0.00}
             seconds (garbage cllections GC.CollectionCount(0) -
             _collectionCount})");
        }
        /// <summary>
        /// The method deletes all unused objects
        /// This is needed for maintaining
        /// the experimental integrity,
        /// i.e.So that garbage collection occurs only
        /// for objects that will be created in
        /// the profiled code block
        ///</summary>
    private static void PrepareForOperation()
        {
            GC.Collect();
```

```
            GC.WaitForPendingFinalizers();
            GC.Collect();
        }
    }

    class Program
    {
        /// <summary>
        /// method for testing the performance of generic
        /// and non-generic list
        /// </summary>

        private static void ValueTypePerfTest()
        {
            const int COUNT = 10000000;
            // The OperationTimer object is created before
            // the collection is used, after its use is
            // completed, displays the time spent for
            // working with the collection
            using (new OperationTimer("List"))
            {
                // use of a generic list
                List<int> list = new List<int>(COUNT);
                for (int n = 0; n < COUNT; n++)
                {
                    list.Add(n);
                    int x = list[n];
                }

                list = null; // for guaranteed execution
                             // of garbage collection
            }

            using (new OperationTimer("ArrayList"))
            {
                // use of a non-generic collection
                ArrayList array = new ArrayList();
```

```
                for (int n = 0; n < COUNT; n++)
                {
                    array.Add(n); // boxing
                    int x = (int)array[n]; // unboxing
                }

                array = null; // for guaranteed execution
                              // of garbage collection
            }
        }

        static void Main(string[] args)
        {
            ValueTypePerfTest();
        }
    }
}
```

The result you got may differ from the one shown in the Figure 6.6, it depends on the computer settings.

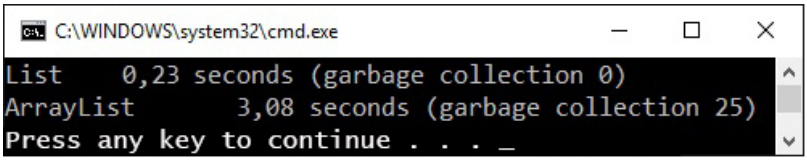

Figure 6.6. Comparison of time costs in a non-generic and generic collection

From the results obtained, it can be seen that the use of the non-generic `ArrayList` collection results in significant time costs, as well as an increase in garbage collections.

Since generic collections provide type security, and performance increases when working with structured data types, Microsoft recommends that whenever possible, generic

collections should be used instead of non-generic collections. The use of the non-generic collection can be justified only if there really is a need to store objects of different types in the same collection.

Work with generic collections is carried out in the same way as with non-generic collections — similar methods and properties are used. To demonstrate our words, we give examples of applying the two most commonly used generic collections. The first example demonstrates working with the `List<T>` generic collection (Figure 6.7).

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

namespace SimpleProject
{

    class Program
    {
        static void Main(string[] args)
        {
            WriteLine("Collection of integers: ");
            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\nCollection of strings: ");
            List<string> listString = new List<string>();
```

```
            listString.Add("Hello");
            listString.Add("from");
            listString.Add("generics");

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

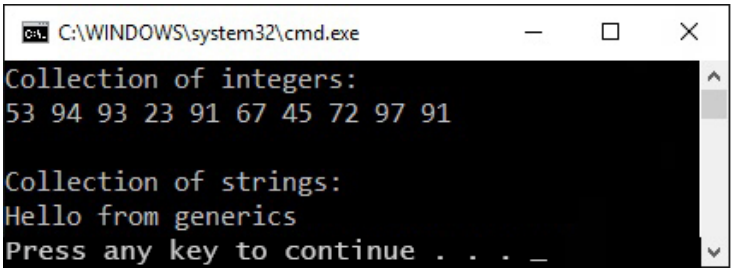

Figure 6.7. Example of working with the List<T> generic collection

When creating a `Dictionary` generic collection, you need to specify two parameters — a key and a value, an example of how to use it is shown in Figure 6.8.

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

namespace SimpleProject
{
    class Program
    {
```

```
static void Main(string[] args)
{
    // Dictionary for storing pairs: string-int
    Dictionary<string, int> groups =
              new Dictionary<string, int>();
    // adding values to the list
    groups["GR1"] = 12;
    groups.Add("GR2", 10);
    groups.Add("GR3", 10);
    groups.Add("GR4", 6);
   // changing the value
    groups["GR1"] = 14;
    // output of all dictionary elements
    WriteLine("Dictionary content: ");
    foreach (KeyValuePair<string, int> p in groups)
    {
        WriteLine($"{p.Key} {p.Value}");
    }

    // deletion by key value
    groups.Remove("GR4");
    // attempt to add an existing key
    try
    {
        groups.Add("GR1", 15);
    }
    catch (ArgumentException e)
    {
        WriteLine(e.Message);
    }
    // an attempt to access an element by
    // a nonexistent key
    try
    {
        WriteLine(groups["GR5"]);
    }
```

```
            catch (KeyNotFoundException e)
            {
                WriteLine(e.Message);
            }
            // checking the existence of the key
            // and getting the value
            int key;
            if (groups.TryGetValue("GR5", out key))
            {
                WriteLine(key);
            }
            else
            {
                WriteLine("The key does not exist");
            }
        }
    }
}
```

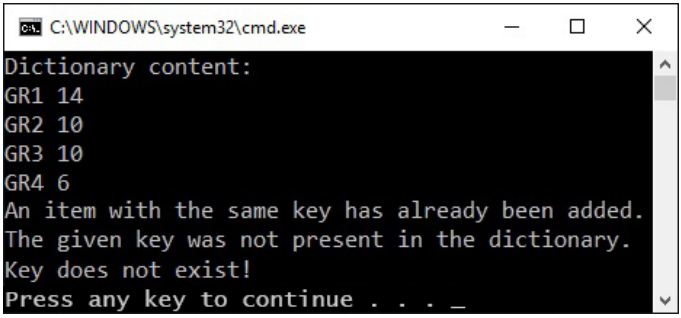

Figure 6.8. Example of working with the Dictionary<TKey, TValue> generic-collection

In the version 6.0 of the C# language, the initialization of `Dictionary` with initial values is simplified, now the values can be assigned with a key. Let's demonstrate this, taking the previous example as a basis, the result will not change.

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

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            // Dictionary for storing pairs: string-int
            Dictionary<string, int> groups =
                        new Dictionary<string, int>
            {
                ["GR1"] = 12,
                ["GR2"] = 10,
                ["GR3"] = 10,
                ["GR4"] = 6
            };
            // the rest of the code remains the same
        }
    }
}
```

Creating generic classes

When creating a generic class, the type parameter is specified in angle brackets after the class name, then this type is also used as normal types. There can be several type generic parameters.

When using a generic class, a real data type is substituted for the type parameter. Creating a specific instance of the generic class is called instantiation.

Since the type parameter is not known in advance, it is impossible to determine the default value for variables

of the generic type. Therefore, the `default(T)` keyword is used to set the default value for the type parameter. In this case, the values of the reference type are assigned null, and the structured — 0.

You cannot perform arithmetic and comparison operations with variables of a generic type. This is due to the fact that, when you instantiate, instead of a type parameter, you can use a data type that does not support these operations.

As an example, create a generic class `Point2D<T>`, which describes a point in two-dimensional space (Figure 6.9).

```
using static System.Console;

namespace SimpleProject
{
    /// <summary>
    /// The generic point class
    /// </summary>
    /// <typeparam name="T">
    /// The coordinates of a point can be of any type
    /// </typeparam>

    public class Point2D<T>
    {
       // type parameter is used to set the type
       // of the property
        public T X { get; set; }
        public T Y { get; set; }
        // type parameter is used to specify the types
        // of method parameters
        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)
        {
            // testing of the generic class - point in 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($"\nx = {p2.X} y = {p2.Y}");
            WriteLine(typeof(Point2D<double>));
        }
    }
}
```

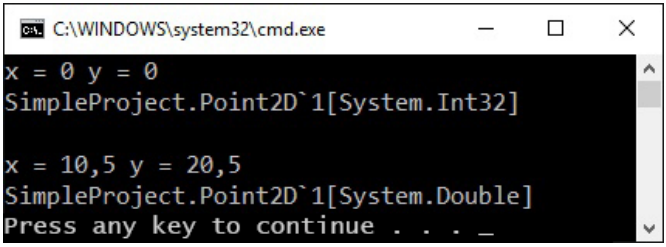

Figure 6.9. Example of working with the generic class Point2D<T>

As can be seen from the results of the program execution, classes for specific data types are created from the generic class.

These classes are called constructed type. To declare `Point-2D<int>`, a class called Point2D`1[System.Int32] is created, for `Point2D<double>` — Point2D`1[System.Double] class. The names of these classes are formed according to the following rule: Point2D is the name of the generic class, 1 is the number of type parameters, [type name] is the type used for pattern specialization. These classes are created from the `generic` class `Point2D<T>` at runtime, when the JIT compiler first compiles a method that uses a variable of such class.

Generic classes can be used as base classes. Therefore, generic classes can have virtual and abstract methods. However, you must follow the following rules of inheritance from generic classes:

- if a non-generic class is inherited from the generic class, the child must specify the type parameter;
- When implementing generic virtual methods, the derived non-generic class must specify the type parameter;
- If another generic class is inherited from the generic class, it must take into account the type constraints specified in the base class.

An example of inheritance from a generic class is shown below (Figure 6.10).

Figure 6.10. Example of inheritance from a generic class

```
using static System.Console;

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

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

    class InheritClass : GenericClass<int> // You must
                         // explicitly specify the type
    {

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

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

The inheritance of a generic class from a regular class is not associated with any restrictions (Figure 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);
        }
    }
}
```

Figure 6.11. Example of generic class inheritance

Nested types inside the generic class

Inside the generic class, you can declare nested classes. These classes will also be generic, even if they do not have their own type parameters. In nested classes, you can use the type parameter specified in the external class. In a nested class, you can also declare your own list of type parameters. In this case, the nested class will be parameterized by two sets of types: own and that of external classes (Figure 6.12).

```
using static System.Console;

namespace SimpleProject
{
    class A<T>
    {
        public class Inner
        {
        }
    }

    class B<T>
    {
        // a nested class has its own list
        // of parameters of type
        public class Inner<U>
        {
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            // to use a nested class you must specify
```

```
            // a real type instead of an external class
            // type parameter
            A<int>.Inner a = new A<int>.Inner();
            WriteLine(a);

            A<double>.Inner a1 = new A<double>.Inner();
            WriteLine(a1);

            // to use a nested class, you must specify
            // a real type instead of a nested class
            // type parameter
            B<int>.Inner<string> b =
                  new B<int>.Inner<string>();

            WriteLine(b);
        }
    }
}
```

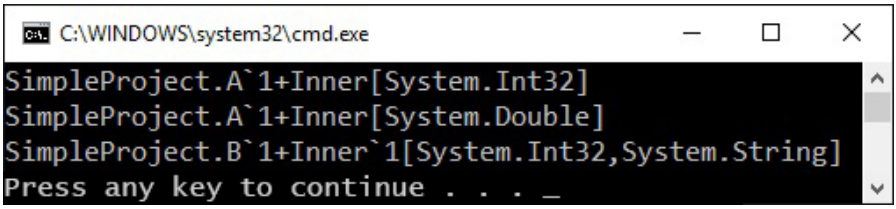

Figure 6.12. Example of inheritance from a generic class

Using constraints

For a type parameter, you can specify several constraints that specify which requirements the data type used instead of this parameter should satisfy.

The list of possible constraints is presented in Table 6.2.

Таблица 6.2. Ограничения параметров типов

Generic constraints	DEscription
`where T: struct`	The type parameter must be inherited from System.ValueType, i.e. be of structured type
`Where T: class`	The type parameter must not be inherited from System.ValueType, i.e. be of reference type
`where T: new()`	The class must have a default constructor (indicated last)
`where T: BaseClass`	The type parameter must be a derived class from the specified base class
`where T: Interface`	The type parameter must implement the specified interface

The syntax for setting the constraint is as follows:

```
class ClassName<T> where T : contraints
```

For example:

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

These constraints require that the type that is used instead of the T parameter be a reference, implement the `IComparable` interface, and have a default constructor.

Let's write down the restrictions for the class `Point2D <T>`, created by us earlier.

```
public class Point2D<T> where T : struct
{
    // the rest of the code remains the same
}
```

Although the introduced constraints did not affect the final result (Figure 6.9), but due to the use of the constraints, the correctness and type safety of the code is now checked at the compilation stage. When you try to compile a specialization for a particular type, the compiler will check whether the specified constraints are met for this type.

If we try to specify a reference type, for example `string`, as a type parameter when creating a `Point2D<T>` class instance, then an error occurs at the compilation stage (Figure 6.13).

```
namespace SimpleProject
{
    public class Point2D<T> where T : struct
    {
        // the rest of the code remains the same
    }
    class Program
    {
        static void Main(string[] args)
        {
            Point2D<string> point =
                    new Point2D<string>("0", "0");
        }
    }
}
```

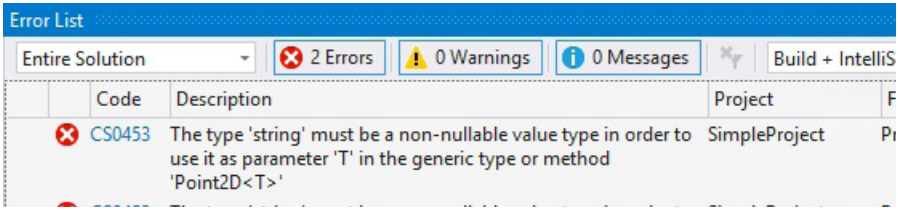

Figure 6.13. Error: The type 'string' must be a non-nullable value type

Creating generic methods

In some cases, it is convenient to have a method parametrized by some type of data. Such a method can be found both in a generic and in the non-generic class. When creating a generic method, the type parameter is specified in angle brackets after the method name.

As an example, we present the implementation of the method of finding the maximum element in a one-dimensional array. Since generic methods, like classes, can be constrained by type parameters, we require that the type of array elements implement the `IComparable` interface. Because to search for the maximum element of the array, you need to compare its elements (Figure 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 };
            // the actual type for the type parameter
            // is specified explicitly
            WriteLine($"Maximum element:
                      {MaxElem<int>(arrInt)}");
            double[] arrDouble = new double[] { 45.62,
                                  77.354, 18.48, 11.3 };
            // the actual type is determined by the type
            // of the transferred array
            WriteLine($"Maximum element:
                      {MaxElem(arrDouble)}");
        }
    }
}
```

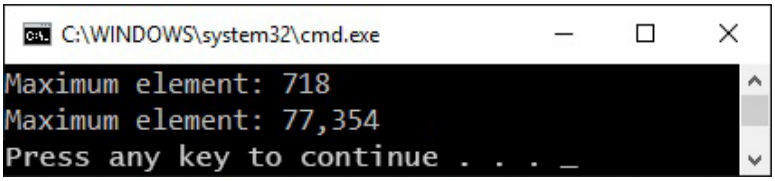

Figure 6.14. Example of using the generic method

As you could notice, if you do not specify the data type when the method is called, it is determined automatically by the type of parameters passed to the method. And although such code does not lead to an error, it is recommended to explicitly specify the data type when calling the method to make your program easier to read.

Generic interfaces

As already mentioned, in the `System.Collections.Generic` namespace there are generic interfaces that correspond to the non-generic interfaces you have learned (Table 6.3).

Table 6.3. Conformity of interfaces

Generic interfaces	Non-generic interfaces
ILits<T>	ILits
IDictionary<T>	IDictionary
ICollection<T>	ICollection
IEnumerator<T>	IEnumerator
IComparer<T>	IComparer
IComparable<T>	IComparable

Now, if you need to implement an interface, preference should be given to generic interfaces. This is not a whim or a tip, but an accurate calculation. To prove this to you, we'll take an example from the Section 5 of the current lesson (using a collection for storing a custom type), but now we use generic interfaces and collection to implement it (Figure 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 $"Last name: {LastName},
                   First name: {FirstName},
                   Born: {BirthDate.ToLongDateString()}";
        }

        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("++++++ list of students ++++++\n");
            foreach (Student student in auditory)
            {
                WriteLine(student);
            }
```

```
            WriteLine("\n++++ sorting by last name ++++\n");
            auditory.Sort();

            foreach (Student student in auditory)
            {
                WriteLine(student);
            }
            WriteLine("\n++++ sort by date of birth
                      ++++++++\n");
            auditory.Sort(new DateComparer());

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

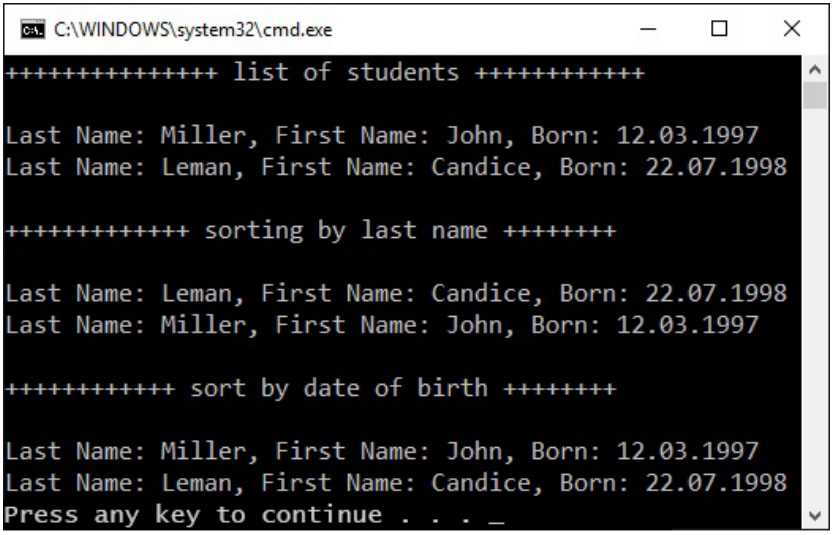

Figure 6.15. Example of using generics when working with a custom type

As you can see, the result of the program does not differ from the results we obtained earlier (Figure 5.2), however, the code became more concise, because when implementing interfaces, the method parameters are now a specific data type, not `object`, which eliminates the need for additional checks.

If necessary, you can create your own generic interfaces. Generic interfaces are convenient because they can be specified as type parameter constraints when creating a generic class, and when implementing this class, one can use methods declared in the interface.

As an example, create a generic class that will contain a collection of data of a generic type, and in this class we implement a method that returns the sum of the elements of this collection. The sum must be of the same type as the data type in the collection.

In order for the elements of the collection to be summed up, you must create an interface containing the method for calculating the sum, and specify this interface as a constraint of the type parameter.

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

namespace SimpleProject
{
    /// <summary>
    /// Generic interface with method for calculating the sum
    /// </summary>
    /// <typeparam name="T"></typeparam>
    interface ICalc<T>
    {
```

```
        T Sum(T b);
    }

    class Program
    {
        /// <summary>
        /// The non-generic class that implements
        /// the ICalc interface
        /// </summary>

        class CalcInt : ICalc<CalcInt>
        // when inheriting the actual data type is specified
        {
            int _number = 0;
            public CalcInt(int n)
            {
                _number = n;
            }
            // when implementing methods, the CalcInt type
            // is used instead of the generic type

            public CalcInt Sum(CalcInt b)
            {
                return new CalcInt(_number + b._number);
            }

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

        /// <summary>
        /// A generic class that contains a collection
        /// of data of a generic type and has a method for
        /// calculating the sum.
        /// To calculate the sum, you set the constraint:
```

```
        /// the type parameter must
        /// implement the interface ICalc<T>
        /// </summary>
        /// <typeparam name="T"></typeparam>
        class MyList<T> where T : ICalc<T>
        {
            // generic data collection
            List<T> list = new List<T>();
            // method of adding data to a collection

            public void Add(T t)
            {
                list.Add(t);
            }
            // method of calculating the sum

            public T Sum()
            {
                if (list.Count == 0)
                {
                    return default(T);
                }
                T result = list[0];
                // the ICalc interface method
                // is used for summation <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($"Sum of elements in the collection:
                      {myList.Sum()}");
        }
    }
}
```

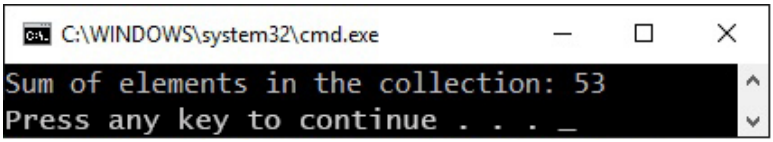

Figure 6.16. Using the generic interface

7. Iterators

What is an iterator?

As you already know, in order for a collection class to list its elements using the `foreach` construct, it must implement the `IEnumerable` interface. However, there is another way to create types that can work with the `foreach` construct, without implementing the `IEnumerable` interface. To do this, in a particular collection class you need to implement a method that determines how elements are transferred to the `foreach` construct, such a method is called an iterator.

Syntax and examples of using iterators

To implement the iterator, you use the `yield` keyword, which is designed to return values to the `foreach` constructor of the calling code using the `yield return` syntax. Herewith, the current value is saved, and the subsequent call of the iterator will continue from this point. An example of using iterators is shown below (Figure 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 $"Last name: {LastName},
                   First name: {FirstName},
                   Born: {BirthDate.ToLongDateString()}";
        }
    }

    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("+++++ list of students +++++\n");
            foreach (Student student in auditory)
            {
                WriteLine(student);
            }
        }
    }
}
```

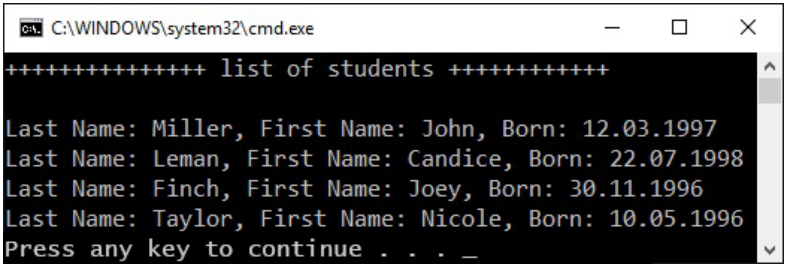

Figure 7.1. Example of using iterators in a collection class

If there is a need for more than one iterator in the collection class, then it is possible to create so-called named

iterators. These are methods whose return type should be the `IEnumerable` interface. The named iterator has the following general form for writing:

```
public IEnumerable interator _ name(list _ of _ parameters)
{
    // necessary code
    yield return returned _ value;
}
```

Iterators can be used not only when working with collections, they can also be used in any class to return the elements received dynamically to the `foreach` construct.

As an example, create a `NamedIterator` class in which the named iterator is used to output integer values in the specified range, and a regular iterator returns values from zero to the specified final value. We have artificially limited the number of output values. For this purpose, we used the `yield break` construct, which is used if there is a need for premature break of the iterator. The possible output of the program is shown in Figure 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 IEnumerator<int> GetEnumerator()
        {
            for (int i = 0; i < _limit; i++)
            {
                if (i == LIM)
                {
                    yield break; // interrupt of an iterator
                                 // by condition
                }
                yield return i;
            }
        }

        public IEnumerable<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("Enter the initial value: ");
            int start = int.Parse(ReadLine());
            Write("\nEnter the final value: ");
            int end = int.Parse(ReadLine());
```

```
            NamedIterator namedIterator =
                          new NamedIterator(end);
            Write("\nAll values: ");
            foreach (int item in namedIterator)
            {
                Write($"{item} ");
            }

            Write("\n\nValues within the specified range: ");
            foreach (int item in
                     namedIterator.GetRange(start))
            {
                Write($"{item} ");
            }
            WriteLine();
        }
    }
}
```

Homework assignment

1. Create a primitive English-Spanish and Spanish-English (or any other) dictionary containing pairs of words — names of countries in Spainish and English. The user should be able to choose the direction of the translation and request a translation.
2. Create a non-generic point class in a three-dimensional space with integer coordinates (`Point3D`), which is inherited from the generic class `Point2D<T>` discussed in the lesson. Implement the following in the class:

- a constructor with parameters that takes initial values for the point coordinates;
- the `ToString()` method.

3. Create a generic class of a line in the plane. In the class, there are 2 fields of the generic point type, the points which the straight line passes through. Implement the following in the class:

- a constructor that takes 2 points;

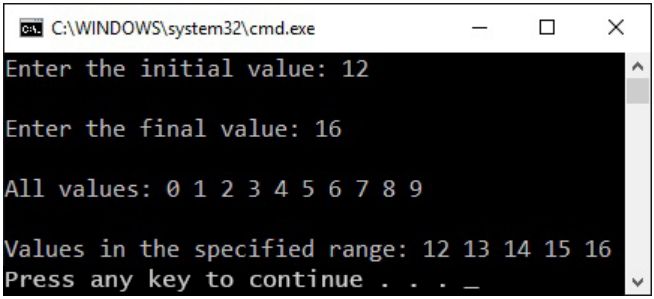

Figure 7.2. Example of using iterators in a class

- a constructor that takes 4 coordinates (x and y coordinates for the first and second points);
- the `ToString()` method.

4. Calculate how many times each word occurs in a given text. The result is written to the Dictionary<`TKey, TValue`> collection.

Lesson 8
The use of collections