Arrays
Strings
Collections

Arrays

- An array is a data series that contains a specified number of elements of the same type.

- The array elements are stored in one contiguous block of memory for quick access.

- The elements can be accessed diretly (ie any element can be read and written at any given moment, any number of times).

Array declaration syntax

type[] Identifier;

- *type*
  Data type of array elements (required);

- *Identifier*
  Array name (required).

An example :

int[]   Numbers;

Arrays cont.1

- Declaring an array variable does not instantiate an array . It is merely a reference variable that will be bound to a physical array only after the array has been instantiated.

- The base class for all arrays in C# is the *System.Array class* , so when you create an array, you can use the methods and properties of that class.

Declaration of multidimensional arrays

- Arrays can be single or multi-dimensional . When declaring one-dimensional arrays, empty square brackets are used:

```
int[] Numbers; // one-dimensional array declaration
```

- For multidimensional arrays, the square brackets contain comma characters that separate the individual dimensions. So a two-dimensional array is declared by putting a single comma inside the parentheses, a three-dimensional array is declared by putting two commas, etc:

```
int [,] NumbersTw; // declaration of a 2D array
int [,,] NumbersTh; // declaration of a 3D array
```

Create an instance of an array

- Declaring an array variable does not instantiate an array (because it's a reference type).

- To create an instance of an array, the operator *new should be used.*

- When creating an instance of an array, specify its size in each of the dimensions:

```
int[] Numbers = new int[20]; // one-dimensional 20-element array
int[,] Numbers2D = new int[2, 2]; // two-dimensional 2x2 array
```

Creating an instance of arrays cont.

- If we do not give any of the sizes in a given dimension, the compiler will report an error:

```
float[]   Numbers = new float[];    // ERROR
double[,]   NumbersD = new double[10,    ]; // ERROR
```

Initializing the elements of an array

- After an array is instantiated, all array elements are initialized with their default values :

  - for **integer type** elements are initialized with values **0** ,

  - for a **floating point type with** values of **0.0** ,

  - for a **logical type,** elements are initialized with the value ***false*** ,

  - for a **reference type,** elements are initialized with ***null values.***

Access to array elements

- To access a specific element of an array, use the indexing operator (square brackets) specifying the index in the given dimension.

- The index determines the position of an element in the array and can take values **from 0 to the number of elements** in a given dimension minus one (due to the numbering from zero).

Access to elements - examples

- In the case of a one-dimensional array, one index is given in parentheses:

```
int[] Numbers        = new int[10];
Numbers[0]         =  10; // assign a value to the first element
Numbers[9]         = Numbers[0] + 10; // assign a value to the last
```

- In the case of multidimensional arrays, the indices for each dimension are separated by commas:

```
int[,]   Numbers = new int[10, 10];
Numbers[1,2]       = 100;
Numbers[2,1]    = Numbers[1,2] * 2;
```

Access to elements - examples cont.

- If the value of the specified index is outside the range (0..size-1), an IndexOutOfRangeException exception is *thrown.*

```
int[] Numbers = new int[10];
Numbers [99] = 100; // ERROR: index out of range
```

Initialization of array elements

- When creating an instance of an array, it is possible to initialize the elements of that array with initial values with **initiation list** .

- Initial values are given in brackets {} separated by commas .

- Remember to enter all initial values .

Initialization of array elements Initiative List

```
int[] Numbers = new int [4] {1, 2, 3, 4};

int[,] Numbers = new int [2,2] {{1, 2}, {3, 4}};

int[]   Numbers = new int [4] {1, 2};   //
ERROR: all values must be there


int [,]Numbers = new int [2, 2] {{1,3}}; //
   ERROR: must be all values
```

Initialization of array elements - short notation

- If an instance of an array is created when an array variable is declared to be initialized with initial values, you can use the short notation in parentheses {}.

- The number of values given in these parentheses will then become the size of the array:

```
int[] Numbers = {1, 2, 3, 4}; // an instance of a four-element array
int[] Numbers = {1, 2}; // an instance of a two-element array
int[,] Numbers = {{1, 2}, {3, 4}}; // an instance of a 2x2 two-dimensional array
```

This notation can only be used when declaring a variable . If the variable is already declared in when creating a new reference to the instance, shorthand notation is not allowed.

Properties of arrays

- *System.Array* class contains many useful properties that you can use when working with arrays.

- The most important and most frequently used are:

- ***Length*** - read-only property that specifies the number of all array elements in all dimensions.

```
int[] Numbers = new int[10];
for (int i = 0; i <Numbers.Length; i ++)
    Numbers[i] = 9999;
```

- ***Rank*** - Read-only property that specifies the number of dimensions of the array.

```
int [,,] NumbersT = new int [10, 10, 10];
if (NumbersT.Rank == 3)
    Console.WriteLine ("Three-dimensional array.");
```

Methods operating on arrays

- The most important and frequently used methods of the *System.Array class* are:

  - ***Clear*** - (static method) assigns default values to array elements from the specified range (**0** for integer types, **0.0** for floating point types, ***false*** for ***bool*** , and ***null*** for reference types).

```
int[] Numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
System.Array.Clear (Numbers, 0, Numbers.Length); // fill
   with zeros in the range 0 to Number.Length
```

  - ***GetLength*** - Returns the length of the array in the specified dimension .

```
int[,] NumbersD = {{1, 2, 3, 4}, {5, 6, 7, 8}};
int size0 = NumbersD.GetLength (0); // 2
int size1 = NumbersD.GetLength (1); // 4
```

Array methods cont. 1

- ***Sort*** - (static method) Sorts the elements of an array . This method can be used to sort an array composed of the elements of a specific class or structure, provided that it implements an interface *IComparable.*

```
int[] Numbers = {4, 2, 7, 1, 5, 6, 3, 9, 8};
System.Array.Sort (Numbers); // 1, 2, 3, 4, 5, 6, 7, 8, 9
```

- ***Clone*** - creates a new instance of an array and copies the values of all elements from the cloned array . This method only creates a " shallow copy " of the elements, which means that if the elements of an array are references to some objects, the copy of that array will contain references to the same objects as the cloned array.

```
int[] Numbers = {4, 2, 7, 1, 5, 6, 3, 9, 8};
int[] NumbersCopy = (int[]) Numbers.Clone(); // copy of the array
```

Methods operating on arrays cont.2

- ***IndexOf*** - Returns the index of the first element whose value matches the value of the argument passed to this method . If no element with the specified value is found in the array, the value is returned
  -1:

```
int[] Numbers = {4, 2, 7, 4, 2, 7, 4, 2, 7};
int where = System.Array.IndexOf(Numbers, 7); // 2
if (where == -1) Console.WriteLine("There is no element in the array.");
```

- ***LastIndexOf*** - returns the index of the last element whose value matches the argument value passed to this method . If no element with the specified value is found in the array, the value -1 is returned:

```
int[] Numbers = {4, 2, 7, 4, 2, 7, 4, 2, 7};
int where = System.Array.LastIndexOf (Numbers, 7); // 8
if (where == -1) Console.WriteLine ("There is no element in the array.");
```

- ***Reverse*** - Reverses the order of items in an array .

```
int[] Numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
System.Array.Reverse (Numbers); // 9, 8, 7, 6, 5, 4, 3, 2, 1
```

Returning arrays from methods

- Arrays like other variables can be returned from methods . In this case, the reference type for the corresponding array is specified in the signature of the method, and an instance of the array is returned from the method.

```
class   Test
{
      static int[] CreateArays (int Size)
      {
          int[] tab = new int [Size]
          return tab;
      }
      static void Main ()
      {
          int[] Numbers = CreateArrays (10);
      }
}
```

Specifying the size of the array in the signature is an error (we only specify the type of the return value)

Passing arrays to methods

- Arrays can be passed to methods as arguments . By passing an array as an argument, a copy of the reference variable is created that is a reference to an instance of the same array (no new instance of the array is created , passing the argument is immediate).

```
class Test
{
   static void Print (int[] Argument)
    {
        for (int i = 0; i <Argument.Length; i ++)
          Console.WriteLine (Argument [i]);
    }
    static void Main ()
    {
        int[] Table = {0, 1, 2, 3};
        Print(Table);
    }
}
```

The argument array of Main

- When starting the program, the first method to be executed is the method *Main* .

- By running the program from the command line, you can pass additional input parameters to the program.

- These parameters are passed to the *Main method* in the form of an argument which is an array of strings with a length equal to the number of parameters passed to the program .

Array of Main arguments - example

```
using System;
class Test
{
    static void Main (string[] args)
    {
       if (args.Length> 0)
       {
         Console.WriteLine ("Parameters:");
         foreach (string str in args)
            Console.WriteLine (str);
       }
       else
            Console.WriteLine ("No parameters.");
    }
}
```

Execution:

```
test.exe -x ala.txt /s:s "Ala has a cat"
```

Strings - The String class

- In C#, the *string* type is used to store and manipulate a string in the Unicode standard.

- The *string type* is an alias to the *System.String class* , so both variable declarations can be used interchangeably.

```
string strString;
System.String strAnotherString;
```

- After declaring a variable of this type, you can assign its value (create an instance of the *String class):*

```
string str =     "Test";
```

The String class

- Individual characters in a string can be accessed using the indexing operator :

```
if  (str[0]  ==  'T') {...}
```

- *string* type is quite specific. The text stored in the string after the creation of an instance of the String class cannot be changed :

```
string str = "Test";
str[0]  =  'X';  // ERROR
```

String class - creating strings

- Any changes to the chain require the use of appropriate methods, properties, and operators, which will result in a new and modified instance of the class *String.*

```
string String1 = "You are", String2 = "Cool", String3;

// create an instance of String3 and initialize it with the value of String1
String3 = String1;

// create a new instance of String3 and initialize it with the combined value
// the previous value of String3, a space character and the value of String2
String3 + = " " + String2;
```

Fields, properties, and methods of the String class

- *String* class offers many useful methods, fields, and properties that you can use when working with strings. The most important and most frequently used are:

  - ***Empty*** - (static field) representation of the empty string.

```
static string Label(int Number)
{
    if (Number == 0) return "Salt";
    if (Number == 1) return "Sugar";
    return String.Empty;
}
```

Fields, properties, and methods of String class cont.1

- ***Length*** - (property) contains the current number of characters in the string (string length).

```
string str =     "Test";
if   (str.Length>     0)
    Console.WriteLine ("String: " + str);
else
    Console.WriteLine ("The string is empty.");
```

The methods of the String class: Compare

- ***Compare*** - (static method) lexically compares two strings . The comparison is based on alphabetical order. The return value from the comparison can be:
  - negative (in case the first string is placed before the second one),
  - zero (when both strings are the same),
  - positive (in case when the first string is after the second).

```
string str1   =   "Ala";
string str2    =   "Cat";
int out = String.Compare (str1, str2); // negative value
```

The method compares strings by default in case-sensitive way. In case the case is to be ignored, another form of the method should be used for comparison.

Methods of the String class: Concat, Contains

- ***Concat*** - (static method) combines two or more strings into one.

```
string str1    =   "Ala";
string str2 = " has a cat";
string strNote = String.Concat (str1, str2); // "Ala has a cat"

string[] strings = {"Ala ", "has ", "a ", "cat"}; // array of
strings string String = String.Concat (strings); // "Ala has
a cat"
```

- ***Contains*** - (method) checks if a string contains the specified string.

```
string strl = "This furniture doesn't look very impressive";
string str2 = "furniture";
bool b Contains = strl.Contains (str2); // true
```

Methods of the String class: Copy, Format

- ***Copy*** *-* (static method) creates a new instance of the *String class* with identical content.

```
string str1   =   "Test";
string str2 = String.Copy (str1); // "Test"
```

Methods of the String class: IndexOf, IndexOfAny

- *IndexOf*- (method) returns the position of the first character or start of a matching substring pattern in a given string.

```
string str1 = "This furniture doesn't look very impressive";
string str2 = "furniture";
int pos;
pos = str1.IndexOf (str2); // pos = 5
pos = str1.IndexOf ('e'); // pos = 13
```

- *IndexOfAny* - (method) returns the position of any of the characters searched for in the string.

```
string str = "This furniture is not very impressive";
int pos = str.IndexOfAny (new char[] {'a', 'e', 'i'}); // pos = 2
```

Methods of the String class: IsNullOrEmpty, LastIndexOf

- ***LastIndexOf*** - (method) Returns the position of the last character or beginning of the last matching substring in the given string.

```
string strl = "This furniture doesn't look very impressive";
string str2 = "furniture";
int pos;
pos = strl.LastIndexOf (str2); // pos = 5
pos = strl.LastIndexOf ('e'); // pos = 42
```

- ***IsNullOrEmpty*** - (static method) Checks whether a chain was created (instantiated) or that an existing chain is empty.

```
string str1 = null;
string str2 = String.Empty;
if (String.IsNullOrEmpty (strl) && String.IsNullOrEmpty (str2))
    {str1 = "Test";
      str2 = "Test";

  }
```

String class Methods: PadLeft, PadRight

- ***PadLeft*** - (method) completes the string on the left to the given length with the given characters (spaces by default). The number of added characters is equal to the given value minus the string length (if it is smaller, nothing is added).

```
string str1    =   "string1";
string str2    =   "string2";
str1 = str1.PadLeft (5);            // "     string1"
str2 = str2. PadLeft (5, '.'); // ".....string2"
```

- ***PadRight*** - (method) completes the string to the length on the right with the given characters (spaces by default). The number of added characters is equal given value minus the string length (if it is smaller, nothing is added).

String class Methods: Remove, Replace

- ***Remove*** - (method) removes the specified number of characters, starting with the character at the given index . If the number of characters is not specified, characters from the given index to the end of the string are assumed to be stripped.

```
string str = "0123 ... 789", result;
result = str.Remove (4); // "0123"
result = str.Remove (4, 3); // "0123789"
```

- ***Replace*** - (method) replaces all occurrences of the specified character or substring with the given one .

```
string str = "0123 ... 789", result;
result = str.Replace ('.', 'X'); // "0123XXX789"
result = str.Replace ("0123 ...", "Number"); // "Number 789"
```

String class Methods: Split, Substring

- ***Split*** - (method) splits a string into substrings based on the specified separators.

```
string str = "run /a /b:cc file.txt";
string [] Commands = str.Split (");
foreach (string command in Commands)
    Console.WriteLine (command);
```

Result:

```
run
/a
/b:cc
file.txt
```

- ***Substring*** - (method) returns a fragment of a character string starting from the element with the specified starting index and the specified character length (if the length is not specified, the fragment is returned from the given index to the end of the string).

```
string str = "This furniture is not very impressive";
string results = str. Substring (5, 9); // "furniture"
string result2 = str.Substring (27); // "impressive"
```

String class Methods: ToCharArray, ToLower

- ***ToCharArray*** - (method) copies the elements of a string to a character array.

```
string str = "This furniture is not very impressive";
char[] characters = str.ToCharArray ();
```

- ***ToLower*** - (method) converts all uppercase letters in a string to lowercase.

```
string   str =    "ALA HAS A CAT";
string result = str.ToLower (); // "ala has a cat"
```

- ***ToUpper*** - (method) converts all lowercase letters in a string to uppercase.

```
string str =     "ala has a cat";
string result = str.ToUpper (); // "ALA HAS A CAT"
```

Methods of the String class: Trim

- ***Trim*** - (method) removes the specified characters from the beginning and the end of the string (by default, only whitespace is removed if you do not specify what characters are to be removed).

```
string str =    "          ... cat ...   "
string results = str.Trim (); // "... cat ..."
string result2 = str.Trim (new char [] {' ', '.'}); // "cat"
```

- ***TrimStart*** - (method) removes the given characters from the beginning of the string.

```
string str =    "          ... cat ...   "
string results = str.Trim (); // "... cat ...        "
string result2 = str.Trim (new char [] {' ', '.'}); // "cat ...    "
```

- ***TrimEnd*** - (method) removes the given characters from the beginning and end of the string

```
string str =    "          ... cat.
string results = str.Trim (); // "   ... cat ..."
string result2 = str.Trim (new char [] {' ', '.'}); // "      ... cat"
```

Building chains - class StringBuilder

- *StringBuilder* class is designed to modify strings without the need to create a new instance of the class .

- Any changes made to an instance of the *StringBuilder class* modify the string directly.

- To operate on a string using the methods and properties of this class, you must instantiate it.

```
string text = "Test string";
StringBuilder sbt = new StringBuilder (text);
```

Properties *StringBuilder: Capacity, Length*

- ***Capacity*** - (a property) allows you to read or set the maximum number of characters that can be stored in memory allocated to an object.

```
string text = "Test string";
StringBuilder sbn = new StringBuilder (text);
sbn.Capacity = 50; // up to 50 characters
```

- ***Length*** - (property) allows you to read or set the current number of characters stored in memory.

```
string text = "Test string";
StringBuilder sbn = new StringBuilder (text);
sbn.Length = 5;
text      =   sbn.ToString ();  //   "Test "
```

The StringBuilder methods: Append, AppendFormat

- ***Append*** - (method) allows you to attach any data type representation to a string .

```
string text = "Test string: ";
StringBuilder sbn = new StringBuilder (text);
sbn.Append (1700); // "Test string: 1700"
sbn.Append (8.8d); // "Test string: 17008.8"
```

The StringBuilder methods: AppendLine, Insert

- ***AppendLine*** - (method) allows you to append a line break to the string.

```
string text = "Price";
StringBuilder sbn = new StringBuilder (text);
sbn.AppendLine ();
sbn.Append ("Different Price");
Console.WriteLine (sbn.ToString ());
```

Result:

```
Price
Different price
```

- ***Insert*** - (method) allows you to insert a string representation of any data type at the appropriate place into a string.

```
string text = "Test string:";
StringBuilder sbn = new StringBuilder (text);
sbn.Insert (2, 1700); // "Te1700st string:"
sbn.Insert (7, 8.8d); // "Te1700s8,8t string:"
```

StringBuilder methods: Remove, Replace

- ***Remove*** - (method) allows you to remove a specified number of characters from a string , starting with a given starting index.

```
string text = "Test string:";
StringBuilder sbn = new StringBuilder (text);
sbn.Remove (7, 8); // "Test st"
```

- ***Replace*** - (method) allows you to replace all occurrences of characters or substrings within a string.

```
string text = "Test string:";
StringBuilder sbn = new StringBuilder (text);
sbn.Replace (":", "->"); // "Test string ->"
```

Collections

- In C #, a collection is a group of items where each item is an object.
- variety of different types of collections in the .NET Framework : lists, queues, stacks, and dictionaries.
- Collections are in the namespace *System.Collections.*

Collections - classes

- The most popular classes defining collections from the *System.Collections space* include:

  - *ArrayList -* represents an array, the size of which can be increased when needed;
  - *Hashtable -* represents a collection of key / value pairs;
  - *Queue -* represents a queue (FIFO - first in, first out);
  - *SortedList -* represents a list of sorted items;
  - *Stack -* represents the stack (LIFO - last in, first out).

The ArrayList class

- You can think of an ArrayList as a better array.
- The items stored in an *ArrayList object* can be of any type *.*

ArrayList Properties:

- ***Count*** - (property) allows you to read the current number of items in the collection.

```
ArrayList arl = new ArrayList (10);
if (arl.Count> 0)
{
    ....
}
```

ArrayList methods: Add, AddRange

- ***Add*** - (method) adds an item to the end of the collection.

```
ArrayList words = new ArrayList ();
words.Add ("ala");
words.Add ("has");
words.Add ("cat");
```

- ***AddRange*** - (method) allows you to add another collection of items to the collection.

```
ArrayList words1 = new ArrayList ();
ArrayList words2 = new ArrayList ();
words1.Add ("ala");
words2.Add ("has");
words2.Add ("cat");
words1.AddRange (words2); // add the content of the words2
collection
```

ArrayList methods: Clear, Contains

- ***Clear*** - (method) removes all items from the collection.

```
words.Clear (); // remove elements
```

- ***Contains*** - (method) allows you to determine whether there is an item in the collection.

```
ArrayList words = new ArrayList ();
words.Add ("ala");
words.Add ("has");
words.Add ("cat");
if (words.Contains ("has")) // check if it contains the word {
    .....
}
```

ArrayList methods: GetRange, IndexOf

- ***GetRange*** - (method) returns a subset of the elements of the given collection . The returned subset contains the specified number of elements starting at the specified index (index - first argument and length - the second).

```
ArrayList words = new ArrayList ();
words.Add ("ala");
words.Add ("has");
words.Add ("cat");
ArrayList subset = words.GetRange (1, 2); // "has", "cat"
```

- ***IndexOf*** - (method) Returns the index number of the first item matching the search pattern.

```
ArrayList words = new ArrayList ();
words.Add ("cat");
words.Add ("dog");
words.Add ("dog");
int pos = words.IndexOf ("dog"); // pos = 1
```

ArrayList methods: Insert, InsertRange

- ***Insert*** - (method) allows you to insert an element into the collection at the position specified by the index .

```
ArrayList words = new ArrayList ();
words.Add ("cat");
words.Add ("dog");
words.Insert (0, "mouse"); // words = {"mouse", "cat", "dog")
```

- ***InsertRange*** - (method) allows you to insert items from another collection into the collection at the specified index .

```
ArrayList words1 = new ArrayList ();
words1.Add ("cat");
words1.Add ("dog");
ArrayList words2 = new ArrayList ();
words2.Add ("mouse");
words1.InsertRange (0, words2); // words1 = {"mouse", "cat", "dog")
```

ArrayList methods: Remove, RemoveAt, RemoveRange

- ***Remove*** - (method) removes the first item from the collection matching the pattern .

```
ArrayList words = new ArrayList ();
words.Add ("cat");
words.Add ("dog");
words.Add ("cow");
words.Add ("dog");
words.Remove ("dog"); // words = {"cat", "cow", "dog")
```

- ***RemoveAt*** - (method) removes an item with the specified one from the collection index.

```
ArrayList words = new ArrayList ();
words.Add ("cat");
words.Add ("dog");
words.Add ("cow");
words.Add ("dog");
words.RemoveAt (1); // words = {"cat", "cow", "dog")
```

- ***RemoveRange*** - (method) removes the specified number of items from the collection , starting at the specified start index .

ArrayList methods - other methods

- ***Reverse*** - (method) reverses the order (order) of the items in the collection.
- ***SetRange*** - (method) copies elements of another collection to the collection, overwriting existing ones , starting at the specified index.
- ***Sort*** - (method) Sorts the items in the collection.

```
ArrayList words = new ArrayList ();
words.Add ("mouse");
words.Add ("dog");
words.Add ("cat");
words.Sort (); // words = {"cat", "dog", "mouse")
```

- ***ToArray*** - (method) copies the collection items to new array

Hashtable class

- ***Hashtable*** class offers many different methods and properties that you can use when working with hashtables .
- When adding an element to the hash table, not only the element should be specified, but also a unique key through which we will access this element.
- Both the key and the element can be objects of any type .
- Elements are added to the hash table using the **Add() method** .

Example of Hashtable class use

```
using System;
using System.Collections;
public class HashtableDemo
{
  private static Hashtable ages = new Hashtable ();
  public static void Main()
  {
     // Adding elements to the hash table,
     // each element is assigned the string
     ages.Add ("Scott", 25);
     ages.Add ("Sam", 6);
     ages.Add ("Jisun", 25);
     // Accessing an item with a given key
     if (ages.ContainsKey ("Scott"))
     {
        int scottsAge = (int) ages ["Scott"];
        Console.WriteLine ("Scott is " + scottsAge.ToString ());
     }
     else
        Console.WriteLine ("Scott's data is not in the array ...");
   }
}
```

Queue class

- *Queue* class provides many different methods and properties that you can use when working with FIFOs.

Queue class - properties, methods

- *Count -* (property) allows you to read the current number of items in the collection.

- *Clear -* (method) removes the contents of the collection.

- *Contains -* (method) checks if the specified item is in the collection.

- *Dequeue -* (method) removes an item from the collection and returns its value (if there is no item in the collection, calling this method will raise an exception).

- *Enqueue -* (method) adds an item to the queue.

- *Peek -* (method) returns the value of the first item in the queue without removing it.

- *ToArray -* (method) copies the collection items to the new array.

Queue class - example

```
using System;
using System.Collections.Generic;
class Example
{
  public static void Main ()
  {
    Queue <string> numbers = new Queue <string> ();
    numbers.Enqueue ("one");
    numbers.Enqueue ("two");
    numbers.Enqueue ("three");
    numbers.Enqueue ("four");
    numbers.Enqueue ("five");

    foreach (string number in numbers)
    {
      Console.WriteLine (number);
    }
   string element = numbers.Dequeue ();
   Console.WriteLine (element);
  }
 }
```

The SortedList class

- This class represents a collection (key and values) sorted according to the keys and accessed using key and index.
- This class is a hybrid of hash tables (Hashtable) and arrays (ArrayList).

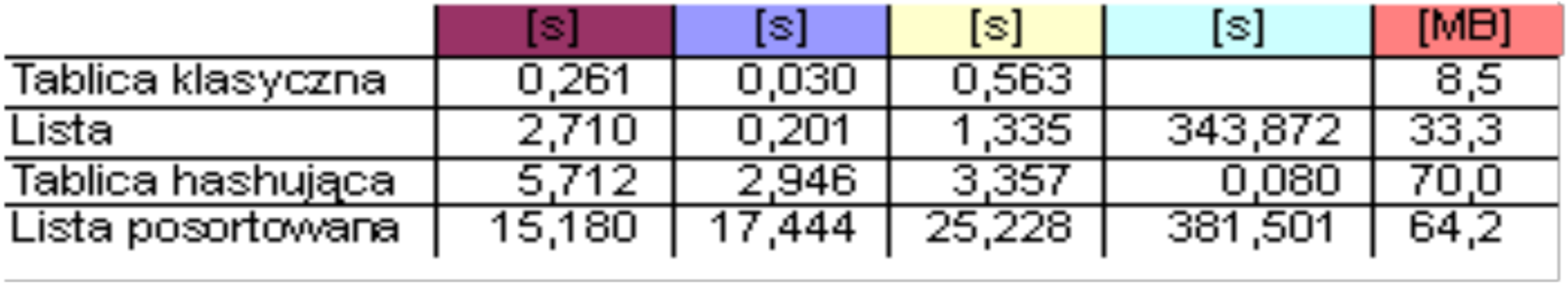
SortedList class - properties, methods

- ***Count*** - (property) allows you to read the current number of items in the collection.
- ***Keys*** - (property) contains the collection of ordered list keys.
- ***Values*** - (property) contains a collection of ordered list values.
- ***Add*** - (method) adds an item with a specific key and value to the end of the collection.
- ***Clear*** - (method) removes the contents of the collection.
- ***ContainsKey*** - (method) checks if the collection contains the specified key.
- ***ContainsValue*** - (method) checks if the collection contains the specified value.
- ***GetByIndex*** - (method) Returns the value of the collection item at the specified index.
- ***GetKey*** - (method) Returns the key of the collection item at the specified index.

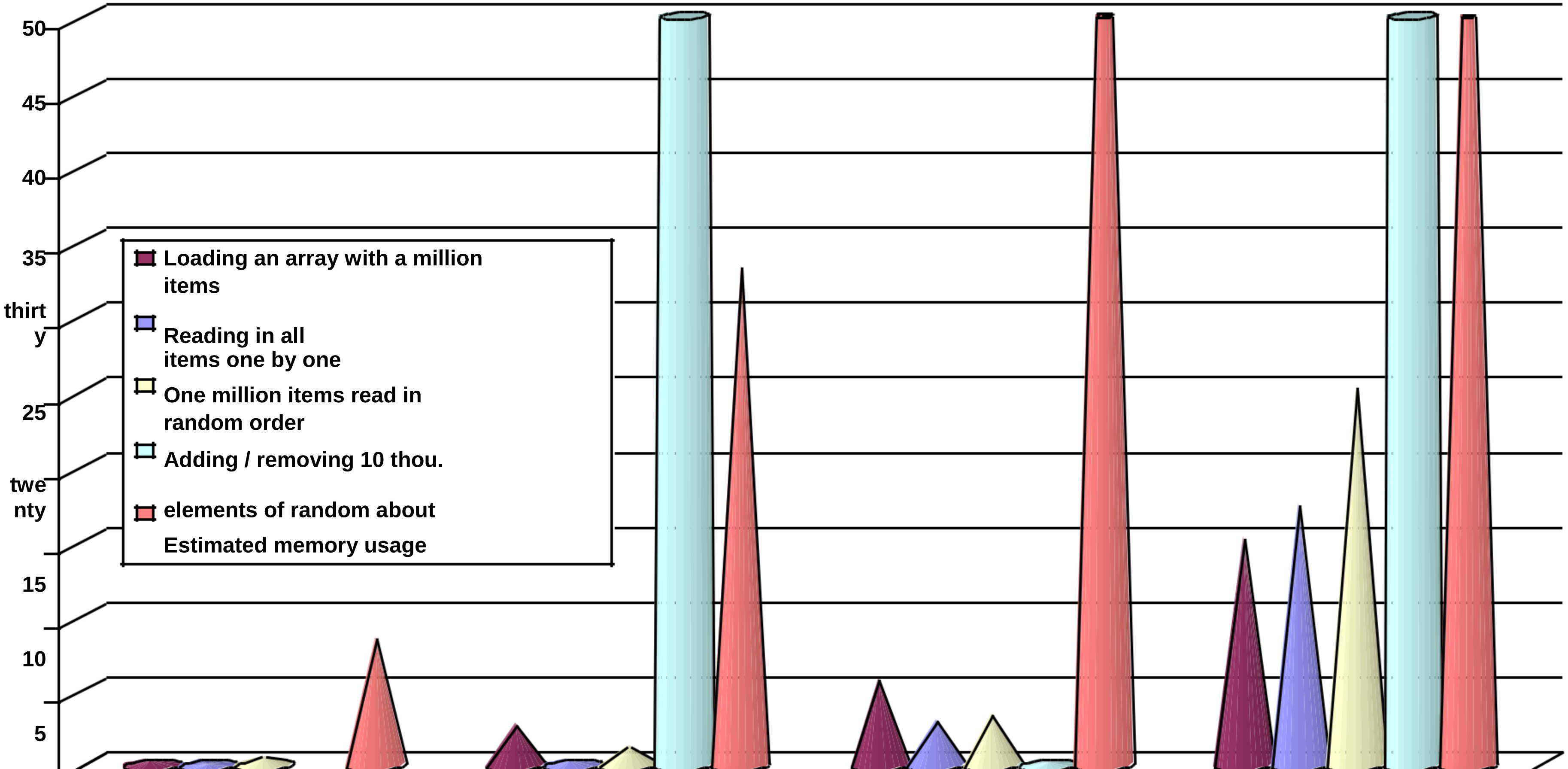
SortedList class - example of use

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

class Example
{
    public static void Main ()
    {
            SortedList sit = new SortedList ();
            sit.Add ("Ala", "cat");
            sit.Add ("Zbyszek", "dog");
            sit.Add ("Ania", "cow");

            foreach (object obj in sit.Keys)
               Console.WriteLine (obj.ToString ());
      }
}
```

Comparing the execution time on different types of arrays



	[s]	[s]	[s]	[s]	[MB]
Tablica klasyczna	0,261	0,030	0,563		8,5
Lista	2,710	0,201	1,335	343,872	33,3
Tablica hashująca	5,712	2,946	3,357	0,080	70,0
Lista posortowana	15,180	17,444	25,228	381,501	64,2



0				
	Classic array	List	The hash table	Sorted list

This is the end ...