SoftServe | Empowering your Business through Software Development

C# Collections. Generic Collections

SoftServe University

17/09/2014 by A. Korkuna
Reviewed by Mykola Kushnir

Agenda

- Array
- System.Collections
- Hashtables
- Stack, Queue
- SortedList
- Collection Interfaces
- System.Collections.Generic
- List<T>

Array

- Array is a data structure that contains several variables of the same type.

```
type [ ]   arrayName;
```

- Array has the following properties:
  - array can be **Single-dimensional**, **Multidimensiona**l or **Jagged**.
  - The default value of **numeric** array elements are set to **zero**, and **reference** elements are set to **null**.
  - Arrays are **zero indexed**: an array with **n** elements is indexed from **0** to **n–1**.
  - Array elements can be of **any type**, including an array type.
- Array types are reference types derived from the abstract base type **Array**. It implements **IEnumerable** and **IEnumerable<(Of <(T>)>)**, for using in **foreach**

Array. Examples

create

```
int[] a = new int[5];
int [,] myMatrix=new int [6,8];
```

element access

```
a[0] = 17;
a[1] = 32;
int x = a[1];
```

number of elements

```
int l = a.Length;
```

default to `false`

```
bool[] a = new bool[10];
```

default to `0`

```
int[]  b = new int[5];
```

set to given values

```
int[]  c = new int[5] { 48,
2, 55, 17, 7 };

int [] ages={5,6,8,9,2,0};
```

Array. Examples

- Multidimensional arrays:

```
string [ , ] names = new string[5,4];
```

- Array-of-arrays (jagged):

```
byte [ ][ ] scores = new byte[ 5 ][ ];
for ( int i = 0; i < scores.Length; i++)
{
        scores[i] = new byte[4];
}
```

- Three-dimensional rectangular array:

```
int [  ,  ,  ] buttons = new int [ 4, 5, 3];
```

Array. Benefits. Limitations

- **Benefits of Arrays**:
  - **Easy** to use: arrays are used in almost every programming language
  - **Fast** to change **elements.**
  - **Fast** to **move** through elements: Because an array is stored continuously in memory, it's **quick** and easy to cycle through the elements one-by-one from start to finish in a loop.
  - You can specify the type of the elements: When you create an array, you can **define** the **datatype**.
- **Limitations of Arrays**:
  - **Fixed size**: Once you have created an array, it will not automatically items onto the end.
  - **Inserting** elements mid-way into a filled array is difficult.

System.Collections. ArrayList

- **System.Collections** namespace
- ArrayList, HashTable, SortedList, Queue, Stack:
  - A collection can contain an **unspecified** number of members.
  - Elements of a collection do not have to share the same **datatype**.
  - An object's **position** in a collection can **change** whenever a change occurs in the whole, herefore, the position of a specific object in the collection can vary.

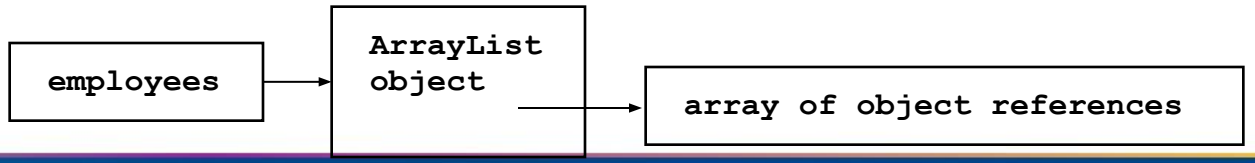
ArrayList

- `ArrayList` is a **special array** that provides us with some functionality over and above that of the standard Array.
- We can dynamically resize it by simply adding and removing elements.

create `ArrayList` to store `Employees`

```
using System.Collections;

class Department
{
  ArrayList employees = new
ArrayList();
  ...
}
```



ArrayList services

add new elements →

remove →

containment testing →

read/write existing element →

control of memory in underlying array →

```
public class ArrayList : IList, ICloneable
{
  int  Add   (object value) // at the end
  void Insert(int index, object value) ...

  void Remove  (object value) ...
  void RemoveAt(int    index) ...
  void Clear   () ...

  bool Contains(object value) ...
  int  IndexOf (object value) ...

  object this[int index] { get... set.. }

  int  Capacity { get... set... }
  void TrimToSize() //minimize memory
  ...
}
```

ArrayList. Benefits and Limitation

- **Benefits** of `ArrayList`:
  - Supports automatic **resizing.**
  - **Inserts** elements: An ArrayList starts with a collection containing no elements.
  - Flexibility when **removing elements**.
  - **Easy** to use.
- **Limitation** of `ArrayLists`:
  - There is **one major limitation** to an `ArrayList`: speed.
  - The flexibility of an `ArrayList` comes at a cost, and since memory allocation is a very expensive business the fixed structure of the simple array makes it a lot faster to work with.

Stack

- `Stack`: last-in-first-out

create `Stack` to store sequence of method calls →

```
using System.Collections;

class Trace
{
  Stack callChain = new
Stack();
  ...
}
```

```
Stack s = new Stack();

s.Push("aaa");
s.Push("bbb");

string t =
(string)s.Peek();

string u = (string)s.Pop();
...
```

add →

examine →

remove →

Queue

- `Queue`: first-in-first-out

create `Queue` to store events →

```
using System.Collections;

class Watcher
{
  Queue events = new
Queue();
  ...
}
```

```
Queue q = new Queue();

q.Enqueue("aaa");
q.Enqueue("bbb");
q.Enqueue("ccc");

string s = (string)q.Peek();

string t =
 (string)q.Dequeue();
```

add →

examine →

remove →

Hashtable

- Represents a collection of **key/value pairs** that are organized based on the hash code of the key.
- The objects used as keys must override the **GetHashCode** method and the **Equals** method.

- **Benefits** of Hashtable:
  - **Non-numeric indexes** allowed. **Key** can be numeric, textual, or even in form of a date. But can't be null reference.
  - Easy **inserting** elements.
  - Easy **removing** elements.
  - Fast **lookup**.

create →
add →
update →
retrieve →

```
Hashtable ages = new Hashtable();

ages["Ann"] = 27;
ages["Bob"] = 32;
ages.Add("Tom", 15);

ages["Ann"] = 28;

int a = (int)ages["Ann"];
```

Hashtable

- **Limitations** of Hashtable:
  - **Performance** and **speed**: `Hashtable` objects are **slower to update** but **faster to use** in a look-up than `ArrayList` objects.
  - **Keys** must be **unique**: An array automatically keeps the index values unique. In a `Hastable` we must monitor the key uniqueness.
  - No useful **sorting**: The items in a `Hashtable` are **sorted internally** to make it easy to find objects very quickly. It's not done by keys or values, the items may as well not be sorted at all.

enumerate entries →

get key and value →

```
Hashtable ages = new Hashtable();

ages["Ann"] = 27;
ages["Bob"] = 32;
ages["Tom"] = 15;

foreach (DictionaryEntry entry in
ages)
{
  string name = (string)entry.Key;
  int    age  = (int)   entry.Value;
  ...
}
```

SortedList

- Represents a collection of **key/value pairs** that are **sorted** by the keys
- Are accessible by **key** and by **index**.
- A SortedList object internally maintains two arrays to store the elements of the list
- Use the new keyword when creating the object. Each adding item is automatically inserted in the correct position in the list, according to a specific IComparer implementation .

```
SortedList stlShippers = new SortedList();

stlShippers["cp"]="Canada Post";

stlShippers["fe"]="Federal Express";

stlShippers["us"]="United State Postal Service";

foreach (DictionaryEntry de in stlShippers)

{

    Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);

}
```

SortedList

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public class SortedList : IDictionary,
                          ICollection,
                          IEnumerable,
                          ICloneable
{...}
```

ICollection<T>	Визначає методи для керування універсальними колекціями.
IComparer<T>	Визначає метод, який реалізується типом для порівняння двох об'єктів.
IDictionary<TKey, TValue>	Представляє універсальну колекцію пар ключ/значення.
IEnumerable<T>	Надає перечислювач, який підтримує простий перебір елементів в колекції
IEnumerator<T>	Підтримує простий перебір елементів універсальної коллекції.
IEqualityComparer<T>	Визначає методи для підтримки операції порівняння об'єктів по відношенню рівності
IList<T>	Представляє колекцію об' єктов, доступ до яких можна отримати додатково за індексом.
ISet<T>	Надає основний інтерфейс для абстракції наборів.

Collections. Drawbacks

- No type checking enforcement at compile time
  - Doesn't prevent adding unwanted types
  - Can lead to difficult to troubleshoot issues
  - Runtime errors!
- All items are stored as objects
  - Must be cast going in and coming out
  - Performance overhead of boxing and unboxing specific types

boxed →

unboxed →

```
ArrayList a = new ArrayList();

int x = 7;

a.Add(x);

int y = (int)a[0];
```

System.Collections.Generic

- **Open constructed types**
  - Classes defined without a specific type
  - Type is specified when instantiated
- **Provides type safety at compile time**

System.Collections.Generic	System.Collections
List<T>	ArrayList
Dictionary<K,T>	HashTable
SortedList<K,T>, SortedDictionary<K,T>	SortedList
Stack<T>	Stack
Queue<T>	Queue
LinkedList<T> O(1)	-
IList<T>	IList
IDictionary<K,T>	IDictionary
ICollection<T>	ICollection
IEnumerator<T>	IEnumerator
IEnumerable<T>	IEnumerable
IComparer<T>	IComparer
IComparable<T>	IComparable

List<T>

- **List generic** class:

```
[SerializableAttribute]
public class List<T> : IList<T>, ICollection<T>,
    IEnumerable<T>, IList, ICollection, Ienumerable
```

- The **List class** is the generic **equivalent** of the **ArrayList** class. It implements the `IList` generic interface using an array whose size is dynamically increased as required.
- The List class **uses** both an equality comparer and an ordering comparer.
- Methods such as **Contains**, **IndexOf**, **LastIndexOf**, and **Remove** use an equality comparer for the list elements.
- If type **T** implements the `IEquatable` generic interface, then the equality comparer **is the Equals method** of that interface; otherwise, the default equality comparer is `Object.Equals(Object)`.

List<T>

- Methods such as **BinarySearch** and **Sort** use an ordering comparer for the list elements.
- The List is not guaranteed to be sorted. You must sort the List before performing operations (such as BinarySearch) that require the List to be sorted.
- Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.
- List accepts a **null** reference **as a valid** value for reference types and allows duplicate elements.

```
{
    public static void Main()
    {
        List<string> dinosaurs = new List<string>();

        Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);

        dinosaurs.Add("Tyrannosaurus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Mamenchisaurus");
        dinosaurs.Add("Deinonychus");
        dinosaurs.Add("Compsognathus");

        Console.WriteLine();
        foreach(string dinosaur in dinosaurs)
        {
            Console.WriteLine(dinosaur);
        }

        Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
        Console.WriteLine("Count: {0}", dinosaurs.Count);

        Console.WriteLine("\nContains(\"Deinonychus\"): {0}",
            dinosaurs.Contains("Deinonychus"));

        Console.WriteLine("\nInsert(2, \"Compsognathus\")");
        dinosaurs.Insert(2, "Compsognathus");

        Console.WriteLine();
        foreach(string dinosaur in dinosaurs)
        {
            Console.WriteLine(dinosaur);
        }


}
```

```
Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);

        Console.WriteLine("\nRemove(\"Compsognathus\")");
        dinosaurs.Remove("Compsognathus");

        Console.WriteLine();
        foreach(string dinosaur in dinosaurs)
        {
            Console.WriteLine(dinosaur);
        }

        dinosaurs.TrimExcess();
        Console.WriteLine("\nTrimExcess()");
        Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
        Console.WriteLine("Count: {0}", dinosaurs.Count);

        dinosaurs.Clear();
        Console.WriteLine("\nClear()");
        Console.WriteLine("Capacity: {0}", dinosaurs.Capacity);
        Console.WriteLine("Count: {0}", dinosaurs.Count);
    }
```

Questions?