Store data in and retrieve data from collections

Gulnaz Zhomartkyzy
D. Serikbayev EKSTU

WORKING WITH DATA COLLECTIONS

Arrays

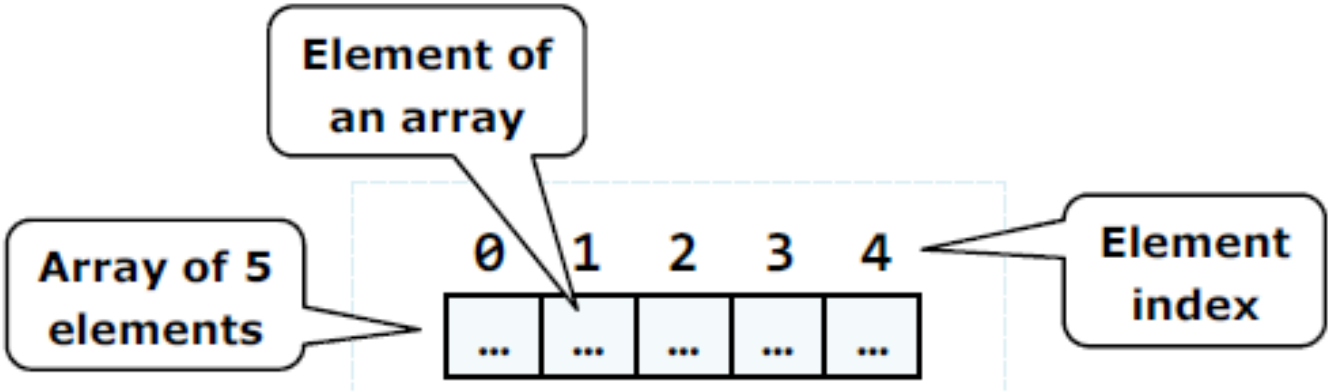
All arrays inherit from the base class System.Array.

The two most commonly used properties of an array are –

Length and Rank.

The **Rank** property - indicates the number of dimension in the array. These properties are helpful when determining the bounds of an array when doing *for* or *while* loops.

The **Clone** - method is used to make a *shallow copy* of the array, while the CopyTo method copies the elements of the array to another array.

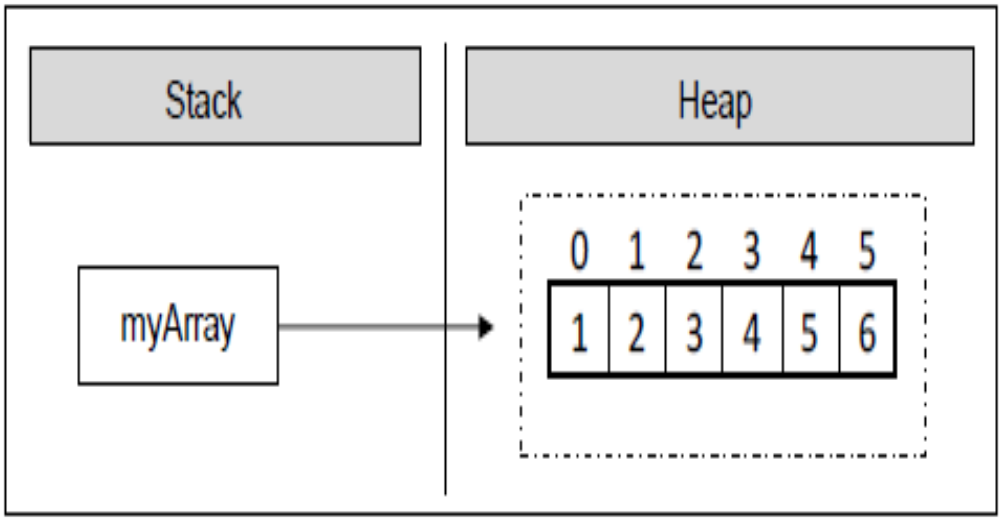


Arrays

An *array* is the most basic type used to store a set of data. An array contains elements, and they are referenced by their index using square brackets, [].

The following example creates a single dimensional array of integers:

```
int[] myArray = new int[6];
myArray[0] = 1;
myArray[1] = 2;
myArray[2] = 3;
myArray[3] = 4;
myArray[4] = 5;
myArray[5] = 6;
```



Two-dimensional array

```
int[,] mySet2 = new int[3, 2];
            mySet2[0, 0] = 1;
            mySet2[0, 1] = 2;
            mySet2[1, 0] = 3;
            mySet2[1, 1] = 4;
            mySet2[2, 0] = 5;
            mySet2[2, 1] = 6;
```

1	2
3	4
5	6

Two-dimensional array

Collections

Collections is a generic term for special classes in C# that are more flexible than arrays. These classes enable you to dynamically add or subtract elements after they have been initialized, associate keys for elements, automatically sort the elements, and allows for elements to be different types or type specific.

Some of the classes are List, List<T>, Dictionary, Dictionary<T>, Stack, and Queue. These classes all have slightly different functionality and are explained in detail in next few sections.

The namespaces for the collection classes are System.Collections, System.Collections.Generic

System.Collections

The System.Collections namespace contains classes for use when you do not have the same type of elements stored within the collection. These collections can mix int, string, classes, or structs within the same collection. Table 9-2 lists the types in the System.Collections namespace. Each of these types is discussed in more detail in the following sections.

System.Collections

Table1 – lists the types in the System.Collections namespace

COLLECTION NAME	DESCRIPTION
ArrayList	Creates a collection whose size is dynamic and can contain any type of object
HashTable	Creates a collection with a key\value pair whose size is dynamic and contains any type of object
Queue	Creates a collection that is first-in-first-out for processing
SortedList	Creates a collection of key\value pairs whose elements are sorted by the key value
Stack	Creates a collection that is last-in-first-out for processing

ArrayList

An ArrayList is a class that enables you to dynamically add or remove elements to the array. This is different from the simple array, which does not enable you to change the dimensions after it is initialized. The ArrayList class is useful when you don’t know the number of elements at the time of creation and also if you want to store different types of data in the array. In the Array examples, all elements of the mySet array had to be an int. An ArrayList has an Add method that takes an object as a parameter and enables you to store any type of object. The following code creates an ArrayList object and adds three elements of different types to the ArrayList:

```
ArrayList myList = new ArrayList();
myList.Add(1);
myList.Add("hello world");
myList.Add(new DateTime(2012, 01, 01));
```

ArrayList

TABLE 2 : Common System Array Properties

PROPERTY	DESCRIPTION
Capacity	Gets or sets the number of elements in the ArrayList
Count	Gets the number of actual elements in the ArrayList
Item	Gets or sets the element at the specified index

ArrayList

TABLE 3: Common System Array Methods

METhOD	DESCRIPTION
Add	Adds an element at the end of the ArrayList
AddRange	Adds multiple elements at the end of the ArrayList
BinarySearch	Searches the sorted ArrayList for an element using the default com-parer and returns the index of the element
Clear	Removes all the elements from the ArrayList
Contains	Determines if an element is in the ArrayList
CopyTo	Copies the ArrayList to a compatible one-dimensional array
IndexOf	Searches the ArrayList and returns the index of the first occurrence within the ArrayList

ArrayList

TABLE 3: Common System Array Methods

METHOD	DESCRIPTION
Insert	Inserts an element into the ArrayList at a specific index
Remove	ARemoves an element from the ArrayList
RemoveAt	Removes an element from the ArrayList by index
Reverse	Reverses the order of the elements in the ArrayList
Sort	Sort the elements in the ArrayList

ArrayList

Example for a simple sorting exercise:

```
ArrayList myList = new ArrayList();
   myList.Add(4);
   myList.Add(1);
   myList.Add(5);
   myList.Add(3);
   myList.Add(2);
   myList.Sort();
   foreach (int i in myList)
   {
    Console.WriteLine(i.ToString());
   }
```

ArrayList

```
class MyObject
{
    public int ID { get; set; }
}

ArrayList myList = new ArrayList();
            myList.Add(new MyObject() { ID = 4 });
            myList.Add(new MyObject() { ID = 1 });
            myList.Add(new MyObject() { ID = 5 });
            myList.Add(new MyObject() { ID = 3 });
            myList.Add(new MyObject() { ID = 2 });
            myList.Sort();
            foreach (MyObject i in myList)
            {
                Console.WriteLine(i.ID.ToString());
            }
```

ArrayList

```
class MyObject : IComparable
{   public int ID { get; set; }
    public int CompareTo(object obj)
    {   MyObject obj1 = obj as MyObject;
        return this.ID.CompareTo(obj1.ID);
    }
}

ArrayList myList = new ArrayList();
            myList.Add(new MyObject() { ID = 4 });
            myList.Add(new MyObject() { ID = 1 });
            myList.Add(new MyObject() { ID = 5 });
            myList.Add(new MyObject() { ID = 3 });
            myList.Add(new MyObject() { ID = 2 });
            myList.Sort();
            foreach (MyObject i in myList)
            {
                Console.WriteLine(i.ID.ToString());
            }
```

ArrayList

```
ArrayList myList = new ArrayList();
myList.Add(new MyObject() { ID = 4 });
myList.Add(new MyObject() { ID = 1 });
myList.Add(new MyObject() { ID = 5 });
myList.Add(new MyObject() { ID = 3 });
myList.Add(new MyObject() { ID = 2 });
myList.Sort();
int foundIndex = myList.BinarySearch(new MyObject() { ID = 4 });
  if (foundIndex >= 0)
  {

Console.WriteLine(((MyObject)myList[foundIndex]).ID.ToString());
  }
  else
  { Console.WriteLine("Element not found");       }
```

The ability to search the array

Hashtable

A Hashtable enables you to store a key\value pair of any type of object. The data is stored accord-ing to the hash code of the key and can be accessed by the key rather than the index of the element. The following sample creates a Hashtable and stores three elements with different keys. You can then reference the elements in the Hashtable by its key.

```
Hashtable myHashtable = new Hashtable();
myHashtable.Add(1, "one");
myHashtable.Add("two", 2);
myHashtable.Add(3, "three");
Console.WriteLine(myHashtable[1].ToString());
Console.WriteLine(myHashtable["two"].ToString());
Console.WriteLine(myHashtable[3].ToString());
```

The preceding code will produce the following output:

one
2
three

Queue

A Queue is **a first-in-first-out** collection. Queues can be useful when you need to store data in a specific order for sequential processing. The following code will create a Queue, add three elements, remove each element, and print its value to the Output window:

```
Queue myQueue = new Queue();
myQueue.Enqueue("first");
myQueue.Enqueue("second");
myQueue.Enqueue("third");
int count = myQueue.Count;
for (int i = 0; i < count; i++)
{
    Console.WriteLine(myQueue.Dequeue());
}
```

- adds an element to the end of the *Queue*

- removes the oldest element from the *Queue*

The preceding code produces the following output:

SortedList

A SortedList is a collection that contains key\value pairs but it is different from a Hashtable because it can be referenced by the key or the index and because it is sorted. The elements in the SortedList are sorted by the IComparable implementation of the key or the IComparer implementation when the SortedList is created. The following code creates a SortedList, adds three elements to the list, and then prints the elements to the Output window:

```
SortedList mySortedList = new SortedList();
mySortedList.Add(3, "three");
mySortedList.Add(2, "second");
mySortedList.Add(1, "first");
foreach (DictionaryEntry item in mySortedList)
    {      Console.WriteLine(item.Value);     }
```

The preceding code produces the following output:

```
first
second
third
```

Stack

A Stack collection is **a last-in-first-out** collection. It is similar to a Queue except that the last element added is the first element retrieved. To add an element to the stack, you use the **Push** method. To remove an element from the stack, you use the **Pop** method.

The following code creates a Stack object, adds three elements, and then removes each element and prints the value to the Output window:

```
Stack myStack = new Stack();
myStack.Push("first");
myStack.Push("second");
myStack.Push("third");
 count = myStack.Count;
for (int i = 0; i < count; i++)
{    Console.WriteLine(myStack.Pop());   }
```

The preceding code produces the following output:

third
second
first

System.Collections.Generic

The System.Collections.Generic namespace contains classes that are used when you know the type of data to be stored in the collection and you want all elements in the collection to be of the same type. Table 9-5 lists the types in the System.Collections.Generic namespace. These types are described in detail in the following sections.

Use Generic Type Whenever Possible

It is considered best practice to use a collection from the Generic namespace because they provide type-safety along with performance gains compared to the non-generic collections.

System.Collections.Generic

TABLE 4: System Collections Generic

COLLECTION NAME	DESCRIPTION
Dictionary<TKey, TValue>	Creates a collection of key\value pairs that are of the same type
List<T>	Creates a collection of objects that are all the same type
Queue<T>	Creates a first-in-first-out collection for objects that are all the same type
SortedList<TKey, TValue>	Creates a collection of key\value pairs that are sorted based on the key and must be of the same type
Stack<T>	Creates a collection of last-in-first-out object that are all of the same type

Dictionary

A Dictionary type enables you to store a set of elements and associate a key for each element. The key, instead of an index, is used to retrieve the element from the dictionary. This can be useful when you want to store data that comes from a table that has an Id column. You can create an object that holds the data and use the record’s Id as the key.

Dictionary

```
static void Sample2()
{
    Dictionary<int, MyRecord1> myDictionary = new Dictionary<int,
MyRecord1>();
    myDictionary.Add(5, new MyRecord1() { ID = 5, FirstName = "Bob",
LastName = "Smith" });
    myDictionary.Add(2, new MyRecord1() { ID = 2, FirstName = "Jane",
LastName = "Doe" });
    myDictionary.Add(10, new MyRecord1() { ID = 10, FirstName = "Bill",
LastName = "Jones" });
    Console.WriteLine(myDictionary[5].FirstName);
    Console.WriteLine(myDictionary[2].FirstName);
    Console.WriteLine(myDictionary[10].FirstName);
}
```

The preceding code will write "**Bob", "Jane",** and "**Bill**" to the Output window.

Dictionary

If you want to know how many elements are in the Dictionary object, you use the Count property,

TABLE 5: Common System Collections Generic Dictionary Methods

METHOD	DESCRIPTION
Add	Adds a key and value to the dictionary
Clear	Removes all the keys and values in the dictionary
ContainsKey	Returns true if the dictionary contains the specified key
ContainsValue	Returns true if the dictionary contains the specified value
Remove	Removes the element with the specified key

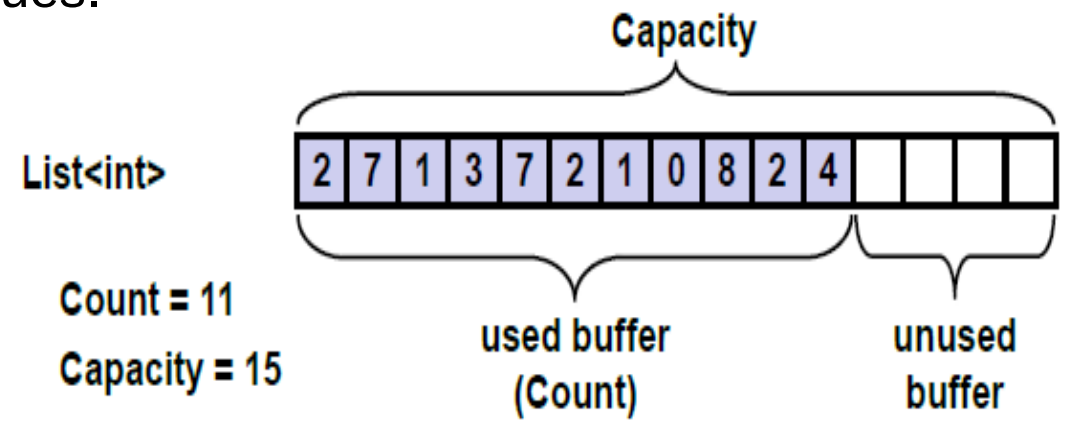
List

A List class is a strongly typed collection of objects. It is similar to an ArrayList except all elements of the List must be of the same type. It is different from a Dictionary because there is no Key, and elements are referenced by index. When you declare the List object, you specify the type of elements it can contain.

```
List<int> myList = new List<int>();
```

When you add elements to the list, they must be of that type, or you get an error. The preceding code created a List object that can contain only int values.

```
myList.Add(1);
myList.Add(2);
myList.Add(3);
```



Summary

Which type of collection class to use based on a specific set of requirements?

Remember the following points:

1. Generic collections are used when you have the same type for all elements.

2. Lists and ArrayLists are referenced by index and do not have a key.

3. Dictionaries, SortedLists, and Hashtables have a key\value pair.

4. Queues and Stacks are used when you have a specific order of processing.

Summary

- The .NET Framework offers both generic (строгие коллекции) and nongeneric collections (нестрогие коллекции). When possible, you should use the generic version.
- Array is the most basic type to store a number of items. It has a fixed size.
- List is a collection that can grow when needed. It’s the most-used collection.
- Dictionary stores and accesses items using key/value pairs.
- *HashSet* stores unique items and offers set operations that can be used on them.
- A *Queue* is a first-in, first-out (FIFO) collection.
- A *Stack* is a first-in, last-out (FILO) collection.
- You can create a custom collection by inheriting from a collection class or inheriting from one of the collection interfaces.