

Unit 1.3: Collections and Generics

Lecturer: Le Thi Bich Tra



Content

- › Lesson 1: Collections
- › Lesson 2: Generics



Lesson 1: Collections

- § Why use collections?
- § Some useful collections
- § ArrayList Class

Collections

- › To help overcome the limitations of a simple array.
- › Collection classes are built to dynamically resize themselves on the fly as you insert or remove item.
- › Many of the collection classes offer increased type safety and are highly optimized to process the contained data in a memory-efficient manner.

Collections

- › A collection class can belong to one of two broad categories:
 - Nongeneric collections (primarily found in the System.Collections namespace)
 - Generic collections (primarily found in the System.Collections.Generic namespace)

Collections

› Some useful collections

System.Collections Class	Meaning in Life	Key Implemented Interfaces
<code>ArrayList</code>	Represents a dynamically sized collection of objects listed in sequential order.	<code>IList</code> , <code>ICollection</code> , <code>IEnumerable</code> , and <code>ICloneable</code>
<code>BitArray</code>	Manages a compact array of bit values, which are represented as Booleans, where true indicates that the bit is on (1) and false indicates the bit is off (0).	<code>ICollection</code> , <code>IEnumerable</code> , and <code>ICloneable</code>
<code>Hashtable</code>	Represents a collection of key/value pairs that are organized based on the hash code of the key.	<code>IDictionary</code> , <code>ICollection</code> , <code>IEnumerable</code> , and <code>ICloneable</code>
<code>Queue</code>	Represents a standard first-in, first-out (FIFO) collection of objects.	<code>ICollection</code> , <code>IEnumerable</code> , and <code>ICloneable</code>
<code>SortedList</code>	Represents a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.	<code>IDictionary</code> , <code>ICollection</code> , <code>IEnumerable</code> , and <code>ICloneable</code>
<code>Stack</code>	A last-in, first-out (LIFO) stack providing push and pop (and peek) functionality.	<code>ICollection</code> , <code>IEnumerable</code> , and <code>ICloneable</code>

ArrayList

- › ArrayList is a non-generic type of collection in C#.
- › It can contain elements of any data types. It is similar to an array, except that it grows automatically as you add items in it.
- › Unlike an array, you don't need to specify the size of ArrayList.
- › Initialize an ArrayList:

```
ArrayList myArrayList = new ArrayList();
```

ArrayList

› Some important properties of ArrayList class:

Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
Item	Gets or sets the element at the specified index.

ArrayList

› Some important methods of ArrayList class:

Add()/AddRange()	Add single element/multiple elements at the end of ArrayList.
Insert()/InsertRange()	Insert a single element/multiple elements at the specified index in ArrayList
Remove()/RemoveRange()	Removes the specified element /elements from the ArrayList.
RemoveAt()	Removes the element at the specified index from the ArrayList.
Sort()	Sorts entire elements of the ArrayList.
Reverse()	Reverses the order of the elements in the entire ArrayList.
Contains	Checks whether specified element exists in the ArrayList or not. Returns true if exists otherwise false.
Clear	Removes all the elements in ArrayList.
CopyTo	Copies all the elements or range of elements to compatible Array.
GetRange	Returns specified number of elements from specified index from ArrayList.
IndexOf	Search specified element and returns zero based index if found. Returns -1 if element not found.
ToArray	Returns compatible array from an ArrayList.

ArrayList

› E.g:

```
ArrayList arrayList1 = new ArrayList();  
arrayList1.Add(1);  
arrayList1.Add("Two");  
arrayList1.Add(3);  
arrayList1.Add(4.5);  
ArrayList arrayList2 = new ArrayList();  
arrayList2.Add(100);  
arrayList2.Add(200);  
//adding entire collection  
arrayList.AddRange(arrayList2);
```

› You can also add items when you initialize it using object initializer syntax.

```
ArrayList arrayList = new ArrayList{ 100, "Two", 12.5, 200 };
```

ArrayList

› Access ArrayList Elements

- ArrayList elements can be accessed using indexer, in the same way as an array. However, you need to cast it to the appropriate type or use the implicit type *var* keyword while accessing it.

```
ArrayList myArrayList = new ArrayList();  
myArrayList.Add(1);  
myArrayList.Add("Two");  
myArrayList.Add(3);  
myArrayList.Add(4.5);  
//Access individual item using indexer  
int firstElement = (int) myArrayList[0]; //returns 1  
string secondElement = (string) myArrayList[1]; //returns "Two"  
int thirdElement = (int) myArrayList[0]; //returns 1  
int fourthElement = (int) myArrayList[0]; //returns 1  
  
//use var keyword  
var firstElement = myArrayList[0]; //returns 1
```

ArrayList

- › Iterate an ArrayList:
 - Use for
 - Use foreach
- › Sort an ArrayList
 - Sort() method arranges elements in ascending order. However, all the elements should have same data type so that it can compare with default comparer otherwise it will throw runtime exception.
 - When sorting arraylist which contains objects, object must implement Comparable or Comparer interface.

Exercises

- › SortedList
- › Stack
- › Queue
- › Hashtable



Lesson 2: Generics

- § Advantage of Generics?
- § Some useful Generics
- § Class Generics
- § Method Generics

The problem of non-generic collections

- › The issue of Performance
 - Problem with boxing and unboxing

```
// Value types are automatically boxed when  
// passed to a member requesting an object.
```

```
ArrayList myInts = new ArrayList();  
myInts.Add(10);  
myInts.Add(20);  
myInts.Add(35);
```

```
// Unboxing occurs when a object is converted back to  
// stack-based data.
```

```
int i = (int)myInts[0];
```

```
// Now it is reboxed, as WriteLine() requires object types!
```

```
Console.WriteLine("Value of your int: {0}", i);
```

The problem of non-generic collections

- › The issue of Type Safety
 - Custom collections for each unique data type

```
public class CarCollection : IEnumerable
{
    private ArrayList arCars = new ArrayList();

    // Cast for caller.
    public Car GetCar(int pos)
    { return (Car) arCars[pos]; }

    // Insert only Car objects.
    public void AddCar(Car c)
    { arCars.Add(c); }

    public void ClearCars()
    { arCars.Clear(); }

    public int Count
    { get { return arCars.Count; } }

    // Foreach enumeration support.
    IEnumerator IEnumerable.GetEnumerator()
    { return arCars.GetEnumerator(); }
}
```


Generics

- › Use a Generic collection class to provide many benefits:
 - Generics provide better performance because they do not result in boxing or unboxing penalties when storing value types.
 - Generics are type safe because they can contain only the type of type you specify.
 - Generics greatly reduce the need to build custom collection types because you specify the “type of type” when creating the generic container

Generics

- › Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at compile time.
- › A generic class can be defined using angle brackets <>
- › Use generic classes in System.Collection.Generic namespace.
- › E.g: List<T>, myGenericClass<U>,...
- › You can take any character or word instead of T.

Generics

› E.g:

```
static void UseGenericList()
{
    // This List<> can hold only Person objects.
    List<Person> morePeople = new List<Person>();
    morePeople.Add(new Person ("Frank", "Black", 50));
    Console.WriteLine(morePeople[0]);

    // This List<> can hold only integers.
    List<int> moreInts = new List<int>();
    moreInts.Add(10);
    moreInts.Add(2);
    int sum = moreInts[0] + moreInts[1];

    // Compile-time error! Can't add Person object
    // to a list of ints!
    // moreInts.Add(new Person());
}
```


Generics

- › Support a handful of generic members (methods and properties):

```
int[] myInts = { 10, 4, 2, 33, 93 };  
  
// Specify the placeholder to the generic  
// Sort<>() method.  
Array.Sort<int>(myInts);
```

- › Support various framework behaviors (cloning, sorting, and enumeration)

```
// IComparable implementation.  
int IComparable.CompareTo(object obj)  
{  
    Car temp = obj as Car;  
    if (temp != null)  
    {...}  
    else  
        throw new ArgumentException("Parameter is not a Car!");  
}
```



```
public class Car : IComparable<Car>  
{  
    ...  
    // IComparable<T> implementation.  
    int IComparable<Car>.CompareTo(Car obj)  
    {  
        return this.CarID.CompareTo(obj.CarID);  
    }  
}
```

Generics

› Generics class/methods:

```
public class MyGenericClass<T>
{
    public static void display(T[] inputArray)
    {
        foreach(T element in inputArray)
            Console.Write(element+ " ");
    }
    public static T[] Sort(T[] inputArray)
    {
        //Code to sort a generic array
        return inputArray;
    }
}
```

```
MyGeneri cCl ass<int> intGeneri cCl ass = new MyGeneri cCl ass<int>();
```

Exercises:

- › List of ProductOrder.
- › Write a swap generic method to swap 2 integer, 2 object of Person type.
- › Practice with Stack, Queue, SortedList

The image features two vertical bars on the left and right sides. Each bar is composed of two parallel vertical rectangles. The left bar has a dark blue outer rectangle and a lighter blue inner rectangle. The right bar has a medium blue outer rectangle and a dark blue inner rectangle.

Thank You !