عنوان مضمون

Visual Programming-II

توسط : صفری

بهار1398

collections

collections

- Arrays, however, have their limitations.
  - they have a fixed size,
    - so you can’t add new items to the end of an existing array without creating a new one. This often means that the syntax used to manipulate arrays can become overly complicated.
  - All items have same data type

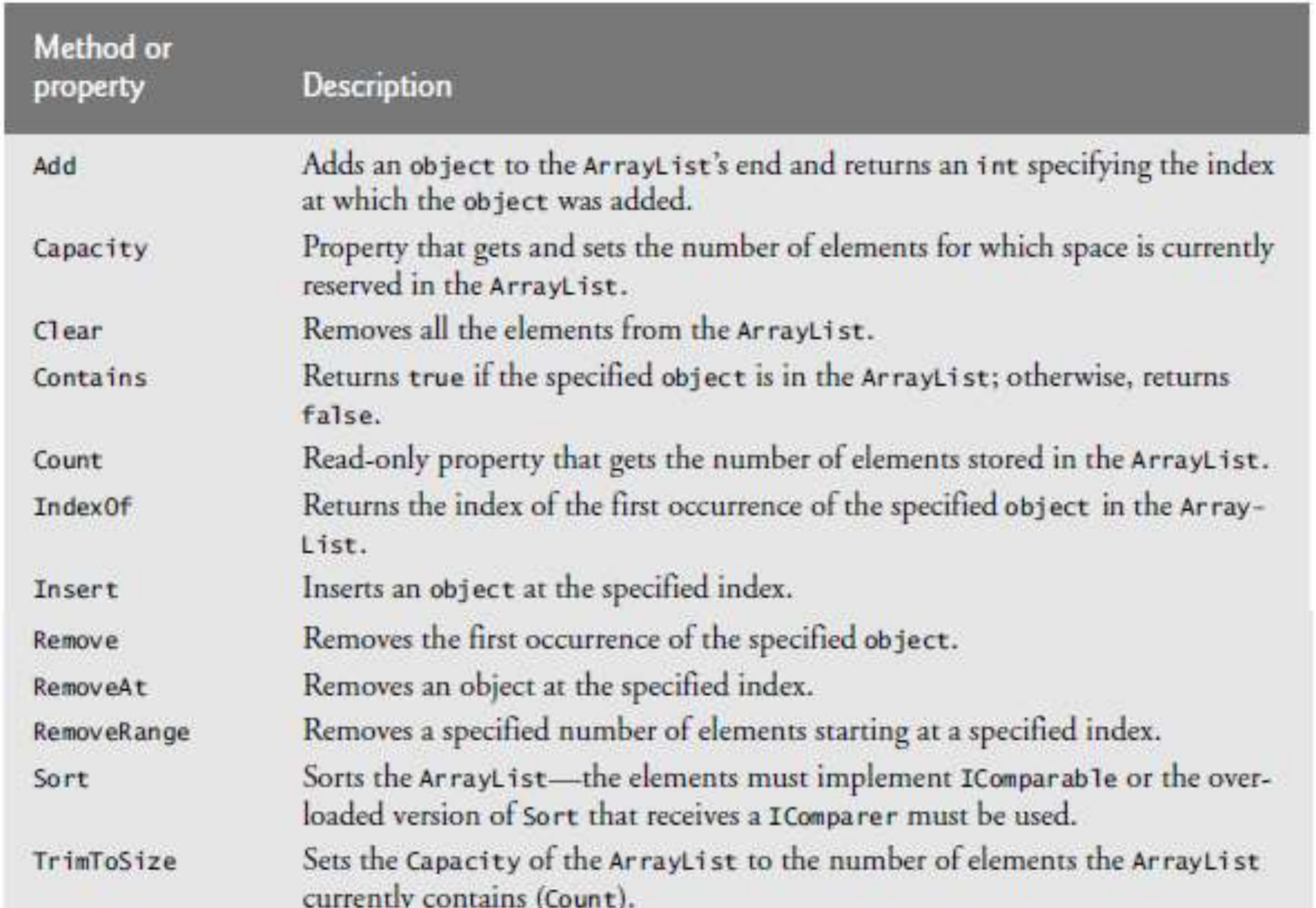
collections

- Arrays in C# are implemented as instances of the System.Array class and are just one type of what are known as *collection classes*.
- Collection classes in general are used for maintaining lists of objects, and they may expose more functionality than simple arrays. Much of this functionality comes through implementing interfaces from the System.Collections namespace, thus standardizing collection syntax. This namespace also contains some other interesting things, such as classes that implement these interfaces in ways other than System.Array.

Class ArrayList

- The .NET Framework’s **ArrayList** collection class mimics the functionality of conventional arrays and provides dynamic resizing of the collection through the class’s methods.
- At any time, an ArrayList contains a certain number of elements less than or equal to its **capacity**—the number of elements currently *reserved* for the ArrayList.
- ArrayLists store references to objects. All classes derive from class object, so an ArrayList can contain objects of any type.

lists some methods and properties of class ArrayList.

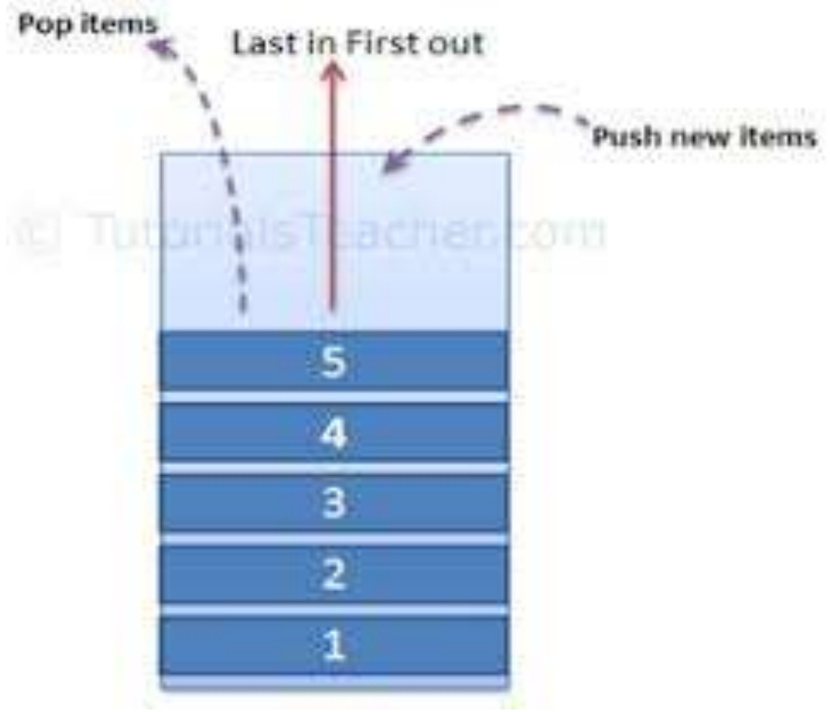


Method or property	Description
`Add`	Adds an `object` to the `ArrayList`'s end and returns an `int` specifying the index at which the `object` was added.
`Capacity`	Property that gets and sets the number of elements for which space is currently reserved in the `ArrayList`.
`Clear`	Removes all the elements from the `ArrayList`.
`Contains`	Returns `true` if the specified `object` is in the `ArrayList`; otherwise, returns `false`.
`Count`	Read-only property that gets the number of elements stored in the `ArrayList`.
`IndexOf`	Returns the index of the first occurrence of the specified `object` in the `Array-List`.
`Insert`	Inserts an `object` at the specified index.
`Remove`	Removes the first occurrence of the specified `object`.
`RemoveAt`	Removes an object at the specified index.
`RemoveRange`	Removes a specified number of elements starting at a specified index.
`Sort`	Sorts the `ArrayList`—the elements must implement `IComparable` or the over-loaded version of `Sort` that receives a `IComparer` must be used.
`TrimToSize`	Sets the `Capacity` of the `ArrayList` to the number of elements the `ArrayList` currently contains (`Count`).

```
ArrayList al = new ArrayList();
al.Add(45);
al.Add(78);
Console.WriteLine("Capacity: {0} ", al.Capacity);
Console.WriteLine("Count: {0}", al.Count);
Console.Write("Content: ");
foreach (int i in al) {
  Console.Write(i + " ");
}
Console.Write("Sorted Content: ");
        al.Sort();
        foreach (int i in al) {
          Console.Write(i + " ");
        }
```

Class Stack

- It represents a last-in, first out collection of object. It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item.



lists some methods and properties of class Stack.

Method & Property	**Description**
Count	Gets the number of elements contained in the Stack
Clear();	Removes all elements from the Stack.
Contains(object obj);	Determines whether an element is in the Stack.
Peek();	Returns the object at the top of the Stack without removing it
Pop();	Removes and returns the object at the top of the Stack.
Push(object obj);	Inserts an object at the top of the Stack.
ToArray();	Copies the Stack to a new array.

```
Stack stack = new Stack();
bool aBoolean = true;
char aCharacter = '$';
int anInteger = 34567;
string aString = "hello";
stack.Push( aBoolean );
stack.Push( aCharacter );
stack.Push( anInteger );
stack.Push( aString );
```

```
Console.WriteLine( "The top element of the stack is {0}\n", stack.Peek());
try
{
        while ( true )
        {
                object removedObject = stack.Pop();
                Console.WriteLine( removedObject + " popped" );
        }
} // end try
catch ( InvalidOperationException exception )
{Console.Error.WriteLine( exception );
}
```

the method print stack

```
pablic void PrintStack( Stack stack )
 { if ( stack.Count == 0 )
            Console.WriteLine( "stack is empty\n" ); // the stack is empty
            else
             {
             Console.Write( "The stack is: " );
            foreach ( var element in stack )
            Console.Write( "{0} ", element );
            Console.WriteLine( "\n" );
            } // end else
} // end method PrintStack
```

Class Hashtable

- When an app creates objects of new or existing types, it needs to manage those objects efficiently. This includes sorting and retrieving objects. Sorting and retrieving information with arrays is efficient if some aspect of your data directly matches the key value and if those keys are *unique* and *tightly packed*.
- If you have 100 employees with nine-digit social security numbers and you want to store and retrieve employee data by using the social security number as a key, it would nominally require an array with 1,000,000,000 elements, because there are 1,000,000,000 unique nine-digit numbers. If you have an array that large, you could get high performance storing and retrieving employee records by simply using the social security number as the array index, but it would be a huge waste of memory.

Class Hashtable

- The Hashtable class represents a collection of **key-and-value pairs** that are organized based on the hash code of the key. It uses the key to access the elements in the collection.
- A hash table is used when you need to access elements by using **key**, and you can identify a useful key value. Each item in the hash table has a key/value pair. The key is used to access the items in the collection.

Property in Class Hashtable

Property	Description
Count	Gets the number of key-and-value pairs contained in the Hashtable.
IsFixedSize	Gets a value indicating whether the Hashtable has a fixed size.
IsReadOnly	Gets a value indicating whether the Hashtable is read-only.
Keys	Gets an ICollection containing the keys in the Hashtable.
Values	Gets an ICollection containing the values in the Hashtable.

Method in Class Hashtable

Method	Description
Add(object key, object value);	Adds an element with the specified key and value into the Hashtable.
Clear();	Removes all elements from the Hashtable.
ContainsKey(object key);	Determines whether the Hashtable contains a specific key.
ContainsValue(object value);	Determines whether the Hashtable contains a specific value.
Remove(object key);	Removes the element with the specified key from the Hashtable.

Example of **Class Hashtable**

```
Hashtable ht = new Hashtable();

    ht.Add("001", "Zara Ali");
    ht.Add("002", "Abida Rehman");
    ht.Add("003", "Joe Holzner");
    ht.Add("004", "Mausam Benazir Nur");
    ht.Add("005", "M. Amlan");
    ht.Add("006", "M. Arif");
    ht.Add("007", "Ritesh Saikia");
```

Example of Class Hashtable

```
if (ht.ContainsValue("Nuha Ali")) {
          Console.WriteLine("This student name is already in the list");
       } else {
          ht.Add("008", "Nuha Ali");
       }
 ICollection key = ht.Keys;

       foreach (string k in key) {
          Console.WriteLine(k + ": " + ht[k]);
       }
       Console.ReadKey();
```

Class SortedList

- The SortedList class represents a collection of key-and-value pairs that are sorted by the keys and are accessible by key and by index.
- A sorted list is a combination of an array and a hash table. It contains a list of items that can be accessed using a key or an index. If you access items using an index, it is an ArrayList, and if you access items using a key, it is a Hashtable. The collection of items is always sorted by the key value.

Methods and Properties of the SortedList Class

Property	Description
Capacity	Gets or sets the capacity of the SortedList.
Count	Gets the number of elements contained in the SortedList.
IsFixedSize	Gets a value indicating whether the SortedList has a fixed size.
IsReadOnly	Gets a value indicating whether the SortedList is read-only.
Item	Gets and sets the value associated with a specific key in the SortedList.
Keys	Gets the keys in the SortedList.
Values	Gets the values in the SortedList.

Methods and Properties of the SortedList Class

Method	Description
Add(object key, object value);	Adds an element with the specified key and value into the SortedList.
Clear();	Removes all elements from the SortedList.
ContainsKey(object key);	Determines whether the SortedList contains a specific key.
ContainsValue(object value);	Determines whether the SortedList contains a specific value.
GetByIndex(int index);	Gets the value at the specified index of the SortedList.
GetKey(int index);	Gets the key at the specified index of the SortedList.

Methods and Properties of the SortedList Class

GetKeyList();	Gets the keys in the SortedList.
GetValueList();	Gets the values in the SortedList.
IndexOfKey(object key);	Returns the zero-based index of the specified key in the SortedList.
IndexOfValue(object value);	Returns the zero-based index of the first occurrence of the specified value in the SortedList.
Remove(object key);	Removes the element with the specified key from the SortedList.
RemoveAt(int index);	Removes the element at the specified index of SortedList.
TrimToSize();	Sets the capacity to the actual number of elements in the SortedList.

```
SortedList sl = new SortedList();
     sl.Add("001", "Zara Ali");
     sl.Add("002", "Abida Rehman");
     sl.Add("003", "Joe Holzner");
     sl.Add("004", "Mausam Benazir Nur");
     sl.Add("005", "M. Amlan");
     sl.Add("006", "M. Arif");
     sl.Add("007", "Ritesh Saikia");
if (sl.ContainsValue("Nuha Ali")) {
       Console.WriteLine("This student name is already in the list");
     } else {
       sl.Add("008", "Nuha Ali");
     }
     // get a collection of the keys.
     ICollection key = sl.Keys;
     foreach (string k in key) {
       Console.WriteLine(k + ": " + sl[k]);
     }
```

BitArray Class

- The BitArray class 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).
- It is used when you need to store the bits but do not know the number of bits in advance. You can access items from the BitArray collection by using an integer index, which starts from zero.

Methods and Properties of the BitArray Class

Property	Description
Count	Gets the number of elements contained in the BitArray.
IsReadOnly	Gets a value indicating whether the BitArray is read-only.
Item	Gets or sets the value of the bit at a specific position in the BitArray.
Length	Gets or sets the number of elements in the BitArray.

Methods and Properties of the BitArray Class

Method	Description
public BitArray And(BitArray value);	Performs the bitwise AND operation on the elements in the current BitArray against the corresponding elements in the specified BitArray.
public bool Get(int index);	Gets the value of the bit at a specific position in the BitArray.
public BitArray Not();	Inverts all the bit values in the current BitArray, so that elements set to true are changed to false, and elements set to false are changed to true.
public BitArray Or(BitArray value);	Performs the bitwise OR operation on the elements in the current BitArray against the corresponding elements in the specified BitArray.
public void Set(int index, bool value);	Sets the bit at a specific position in the BitArray to the specified value.
public void SetAll(bool value);	Sets all bits in the BitArray to the specified value.
public BitArray Xor(BitArray value);	Performs the bitwise eXclusive OR operation on the elements in the current BitArray against the corresponding elements in the specified BitArray.

```
BitArray ba1 = new BitArray(8);
BitArray ba2 = new BitArray(8);

byte[] a = { 60 };
byte[] b = { 13 };

//storing the values 60, and 13 into the bit arrays
ba1 = new BitArray(a);
ba2 = new BitArray(b);

//content of ba1
Console.WriteLine("Bit array ba1: 60");

for (int i = 0; i < ba1.Count; i++) {
   Console.Write("{0, -6} ", ba1[i]);
}
Console.WriteLine();
```

```
//content of ba2
        Console.WriteLine("Bit array ba2: 13");

        for (int i = 0; i < ba2.Count; i++) {
           Console.Write("{0, -6} ", ba2[i]);
        }
        Console.WriteLine();
        BitArray ba3 = new BitArray(8);
        ba3 = ba1.And(ba2);

        //content of ba3
        Console.WriteLine("Bit array ba3 after AND operation: 12");

        for (int i = 0; i < ba3.Count; i++) {
           Console.Write("{0, -6} ", ba3[i]);
        }
        Console.WriteLine();
```

```
ba3 = ba1.Or(ba2);

    //content of ba3
    Console.WriteLine("Bit array ba3 after OR operation: 61");

    for (int i = 0; i < ba3.Count; i++) {
      Console.Write("{0, -6} ", ba3[i]);
    }
    Console.WriteLine();

    Console.ReadKey();
```