Collections in C#

The Collections in C# are a set of predefined classes that are present in the `System.Collections` namespace which provides greater capabilities than the traditional arrays. The collections in C# are reusable, more powerful, more efficient and most importantly they have been designed and tested to ensure quality and performance. We can say a Collection in C# is a dynamic array. That means the collections in C# have the capability of storing multiple values but with the following features.

- Size can be increased dynamically.
- We can insert an element into the middle of a collection.
- It also provides the facility to remove or delete elements from the middle of a collection.

The collection which is from.Net framework 1.0 is called simply collections or Non-Generic collections in C#. These collection classes are present inside the `System.Collections` namespace. The example includes are `Stack`, `Queue`, `LinkedList`, `SortedList`, `ArrayList`, `HashTable`, etc.

Auto-Resizing of collections

The capacity of a collection increases dynamically i.e. when we keep adding new elements, then the size of the collection keeps increasing automatically. Every collection class has three constructors and the behavior of collections will be as following when created using a different constructor.

1. Default Constructor: It Initializes a new instance of the collection class that is empty and has the default initial capacity as zero which becomes four after adding the first element and whenever needed the current capacity becomes double.
2. Collection (int capacity)**:** This constructor initializes a new instance of the collection class that is empty and has the specified initial capacity, here also when the requirement comes current capacity doubles.
3. Collection (Collection): It Initializes a new instance of the collection class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied, here also when the requirement comes current capacity doubles.

`ArrayList`

The `ArrayList` in C# is a collection class that works like an array but provides the facilities such as dynamic resizing, adding and deleting elements from the middle of a collection. It implements the `System.Collections.IList` interface using an array whose size is dynamically increased as required.

The following are some important methods and properties provided by the `ArrayList` collection class in C#.

- `Add(object)` : This method is used to add an object to the end of the collection.
- `AddRange(ICollection)` **:** Adds the elements of an `ICollection` to the end of the `ArrayList`.

- `Remove(object)` : This method is used to remove the first occurrence of a specific object from the collection.
- `RemoveAt(int)`: This method takes the index position of the elements and remove that element from the collection.
- `Insert(int, Object)` : This method is used to inserts an element into the collection at the specified index.
- `InsertRange(Int32, ICollection)` : Inserts the elements of a collection into the `ArrayList` at the specified index.
- `Clear()` : Removes all elements from the `ArrayList`.
- `Clone()` : Creates a shallow copy of the `ArrayList`.
- `IndexOf(Object)` : Searches for the specified Object and returns the zero-based index of the first occurrence within the entire `ArrayList`.
- `LastIndexOf(Object)` : Searches for the specified Object and returns the zero-based index of the last occurrence within the entire `ArrayList`.
- `Reverse()` : Reverses the order of the elements in the entire `ArrayList`.
- `Sort()` : Sorts the elements in the entire `ArrayList`.
- `Sort(IComparer)` : Sorts the elements in the entire `ArrayList` using the specified comparer.
- `ToArray()` : Copies the elements of the `ArrayList` to a new Object array.
- `TrimToSize()` : Sets the capacity to the actual number of elements in the `ArrayList`.
- `Capacity`: This property gives you the capacity of the collection means how many elements you can insert into the collection.
- `Count`: Gets the number of elements actually contained in the `ArrayList`.
- `Item[Int32]` : Gets or sets the element at the specified index.

Example:

```
using System;
using System.Collections;
public class SamplesArrayList
{
   public static void Main()
   {
      // Creates and initializes a new ArrayList.
      ArrayList myAL = new ArrayList();
      myAL.Add("Hello");
      myAL.Add("World");
      myAL.Add("!");
      // Displays the properties and values of the ArrayList.
      Console.WriteLine( "myAL" );
      Console.WriteLine( "    Count:    {0}", myAL.Count );
      Console.WriteLine( "    Capacity: {0}", myAL.Capacity );
      Console.Write( "    Values:" );
      PrintValues( myAL );
   }
```

```
   public static void PrintValues( IEnumerable myList )
   {
      foreach ( Object obj in myList )
         Console.Write( "   {0}", obj );
      Console.WriteLine();
   }
}
/*
This code produces output similar to the following:
myAL
    Count:    3
    Capacity: 4
    Values:   Hello   World   !
*/
```

HashTable

In the case of `Array` and `ArrayList` in C#, we can access the elements from the collection using a key. That key is nothing but the index position of the elements which is starts from zero (0) to the number of elements – 1. But in reality, it's very difficult to remember the index position of the element in order to access the values.

The `Hashtable` in C# is a collection that stores the element in the form of "Key-Value pairs". The data in the `Hashtable` are organized based on the hash code of the key. The key in the `HashTable` is defined by us and more importantly, that key can be of any data type. Once we created the `Hashtable` collection, then we can access the elements by using the keys. The `Hashtable` class comes under the `System.Collections` namespace.

Note: The `Hashtable` computes a hash code for each key. Then it uses that hash code to lookup for the elements very quickly which increases the performance of the application.

The following are some important methods and properties provided by the `HashTable` collection class in C#.

- `Count:` : Gets the number of key/value pairs contained in the `Hashtable`.
- `Item[Object]` : Gets or sets the value associated with the specified key.
- `Keys` : Gets an `ICollection` containing the keys in the `Hashtable`.
- `Values` : Gets an `ICollection` containing the values in the `Hashtable`.
- `Add(Object, Object)` : Adds an element with the specified key and value into the `Hashtable`.
- `Clear()` : Removes all elements from the `Hashtable`.
- `Clone()` : Creates a shallow copy of the `Hashtable`.
- `ContainsKey(Object)` : Determines whether the `Hashtable` contains a specific key.
- `ContainsValue(Object)` : Determines whether the `Hashtable` contains a specific value.
- `GetHash(Object)` : Returns the hash code for the specified key.
- `Remove(Object)` : Removes the element with the specified key from the `Hashtable`.

Example:

```
using System;
using System.Collections;
class Example
{
    public static void Main()
    {
        // Create a new hash table.
        //
        Hashtable openWith = new Hashtable();
        // Add some elements to the hash table. There are no
        // duplicate keys, but some of the values are duplicates.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");
        // The Add method throws an exception if the new key is
        // already in the hash table.
        try
        {
            openWith.Add("txt", "winword.exe");
        }
        catch
        {
            Console.WriteLine("An element with Key = \"txt\" already exists.");
        }
        // The Item property is the default property, so you
        // can omit its name when accessing elements.
        Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);
        // The default Item property can be used to change the value
        // associated with a key.
        openWith["rtf"] = "winword.exe";
        Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);
        // If a key does not exist, setting the default Item property
        // for that key adds a new key/value pair.
        openWith["doc"] = "winword.exe";
        // ContainsKey can be used to test keys before inserting
        // them.
        if (!openWith.ContainsKey("ht"))
        {
            openWith.Add("ht", "hypertrm.exe");
            Console.WriteLine("Value added for key = \"ht\": {0}",
openWith["ht"]);
        }
        // When you use foreach to enumerate hash table elements,
        // the elements are retrieved as KeyValuePair objects.
        Console.WriteLine();
        foreach( DictionaryEntry de in openWith )
        {
            Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
        }
```

```
        // To get the values alone, use the Values property.
        ICollection valueColl = openWith.Values;
        // The elements of the ValueCollection are strongly typed
        // with the type that was specified for hash table values.
        Console.WriteLine();
        foreach( string s in valueColl )
        {
            Console.WriteLine("Value = {0}", s);
        }
        // To get the keys alone, use the Keys property.
        ICollection keyColl = openWith.Keys;
        // The elements of the KeyCollection are strongly typed
        // with the type that was specified for hash table keys.
        Console.WriteLine();
        foreach( string s in keyColl )
        {
            Console.WriteLine("Key = {0}", s);
        }
        // Use the Remove method to remove a key/value pair.
        Console.WriteLine("\nRemove(\"doc\")");
        openWith.Remove("doc");
        if (!openWith.ContainsKey("doc"))
        {
            Console.WriteLine("Key \"doc\" is not found.");
        }
    }
}
/* This code example produces the following output:

An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
Value added for key = "ht": hypertrm.exe

Key = dib, Value = paint.exe
Key = txt, Value = notepad.exe
Key = ht, Value = hypertrm.exe
Key = bmp, Value = paint.exe
Key = rtf, Value = winword.exe
Key = doc, Value = winword.exe

Value = paint.exe
Value = notepad.exe
Value = hypertrm.exe
Value = paint.exe
Value = winword.exe
Value = winword.exe

Key = dib
Key = txt
Key = ht
Key = bmp
```

```
Key = rtf
Key = doc

Remove("doc")
Key "doc" is not found.
 */
```

It is not recommend that you use the `Hashtable` class for new development. Instead you can use the generic `Dictionary<TKey,TValue>` class.