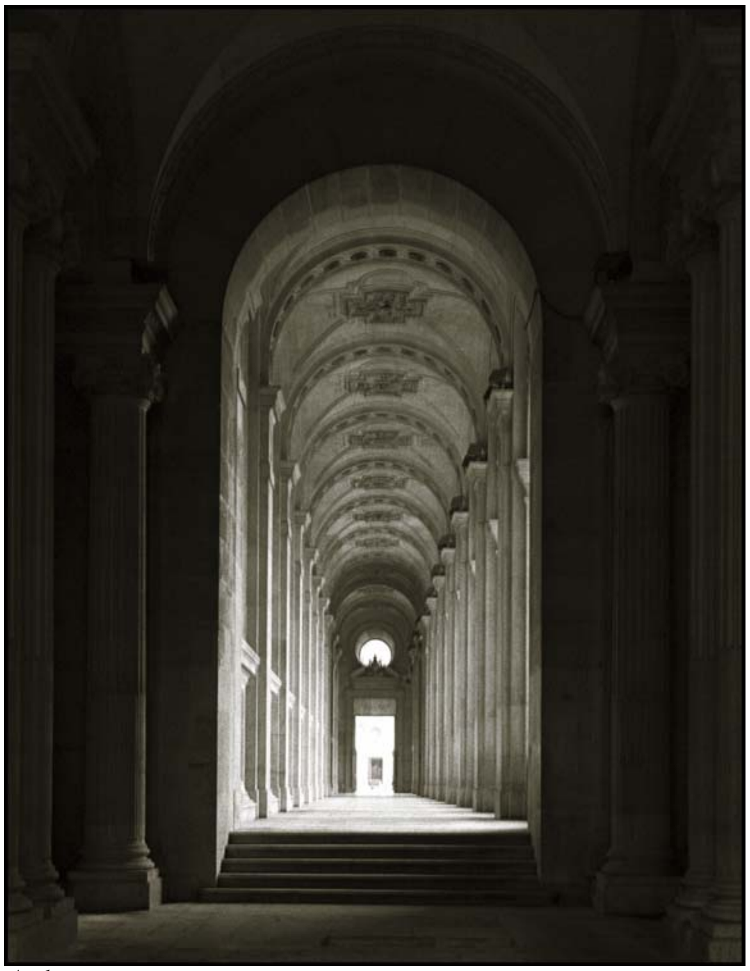
Chapter 1


Contax T — Kodak Tri-X Professional

Archway

Collections Quick Start

Learning Objectives

- *Quickly put collection classes to work in your programs*
- *Create and utilize an ArrayList collection*
- *Create and utilize a List<T> collection*
- *Add elements to an ArrayList and List<T> collection*
- *Access elements in an ArrayList and List<T> collection*
- *Cast objects to their proper type upon retrieval from an ArrayList collection*

Introduction

This chapter, as its name implies, provides you with a "quick start" introduction to the .NET collections framework. My intent is to show you how to put collection classes to work immediately solving real programming problems that require the manipulation of a collection of objects.

I will focus the discussion on the use of the ArrayList and List<T> collection classes. Together, these two classes easily represent the most frequently used collection types. Along the way I will explain the difference between early, non-generic .NET collection classes, which in this book I will refer to as "old school" collection classes, and the more recent (as of .NET 2.0) generic collection classes. I still believe it is a good idea to discuss the old school collections because you may find yourself maintaining legacy .NET code.

You will be surprised at how productive you can be by simply arming yourself with the knowledge of how to use these two classes. However, as you progress through this book, you will quickly learn that to gain full advantage of the .NET collection framework you must cultivate a deep understanding of special coding techniques, especially when you want to manipulate collections of your very own user-defined data types.

Why Use a Collection?

The short answer to this question is, "Because it will save you time, and lots of it!" The .NET collections framework, comprised of classes, structures, and interfaces found in four namespaces: *System.Collections*, *System.Collections.Generic*, *System.Collections.ObjectModel*, and *System.Collections.Specialized*, makes your programming life seriously easy by providing a ready-made set of collection types that offer a solution to just about any collection requirement you will encounter.

Before you can tap its full potential you will need to investment some time up front studying the collections framework. You'll need to understanding how one type of collection differs from another, how the underlying data structure used to implement a particular collection affects its performance, and how to code your custom user-defined data types to behave well in a collection. I discuss these and many other topics in detail throughout the book. In the end, you will be generously rewarded for your effort.

However, in the meantime, to demonstrate the power of collections I will show you how to use two popular collection types: ArrayList and List<T>. You would select either of these collection types in situations where you would ordinarily use an array, but wanted more specialized behavior than a simple array offered. Both the ArrayList and List<T> collection classes can be manipulated like an array, and in fact, both collections are array based, so if you already know how to manipulate an array, you already know how to use part of the interface to both of these collection classes. I'll start the discussion with the old school ArrayList class.

ArrayList Class

The ArrayList class is a "non-generic" collection that behaves like an array on steroids. By non-generic I mean it has been around since the early days of the .NET framework and allows you to insert any type of object into it, and retrieve only System.Object type objects from it, which must be cast to the appropriate type before calling any type-specific interface methods on the retrieved object. This "object in, object out" behavior characterizes the early .NET collections framework.

Another feature of the ArrayList class is its ability to dynamically grow or expand when necessary to accommodate more objects. An ordinary array does not possess this capability. To grow an ordinary array, you must copy its elements to a temporary array, create a bigger permanent array, and then copy the elements from the temporary array back to the new larger permanent array. (Whew!) All this is done automagically for you with an ArrayList collection.

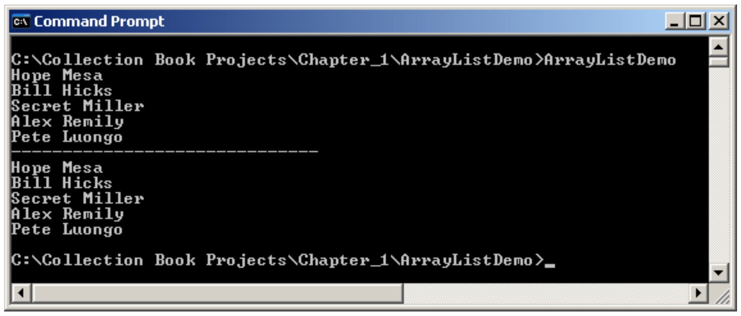
Example 1.1 shows an ArrayList in action manipulating a collection of Strings.

1.1 ArrayListDemo.cs

```
using System;
using System.Collections;

```

```
4        public class ArrayListDemo {
5          public static void Main(){
6            ArrayList list = new ArrayList();
7
8            //Add elements using the Add() method
9            list.Add("Hope Mesa");
10           list.Add("Bill Hicks");
11           list.Add("Secret Miller");
12           list.Add("Alex Remily");
13           list.Add("Pete Luongo");
14
15           //access elements using array indexer notation
16           for(int i=0; i<list.Count; i++){
17             Console.WriteLine(list[i]);
18           }
19
20           Console.WriteLine("-----------------------------");
21
22           //or, use the foreach statement which hides the complexity of the enumerator
23           foreach(string s in list){
24             Console.WriteLine(s);
25           }
26         }
27       }
```

Referring to example 1.1 — note that to use the ArrayList class you must add the `using` directive as is shown on line 2 to provide shortcut naming access to the members of the System.Collections namespace. On line 6 an ArrayList object is created and assigned to the reference named `list`. Lines 9 through 13 show string objects being added to the list. The `for` statement beginning on line 16 iterates over the list of strings and prints their values to the console. The `foreach` statement that starts on line 23 shows an alternative way to iterate over the collection with the help of an *iterator*. However, the `foreach` statement hides the complexity of the iterator so there's not much to see from looking at the code. Figure 1-1 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_1\ArrayListDemo>ArrayListDemo
Hope Mesa
Bill Hicks
Secret Miller
Alex Remily
Pete Luongo
-----------------------------
Hope Mesa
Bill Hicks
Secret Miller
Alex Remily
Pete Luongo

C:\Collection Book Projects\Chapter_1\ArrayListDemo>_
```

Figure 1-1: Results of Running Example 1.1

In example 1.1, the ArrayList contains only strings, but there's nothing stopping you, except your good sense, from adding any kind of object to the list. To demonstrate, I'll create a custom data type named Dog, the code for which appears in example 1.2.

1.2 Dog.cs

```
public class Dog {
  private string _first_name;
  private string _last_name;
  private string _breed;

  public Dog(string breed, string f_name, string l_name){
    _breed = breed;
    _first_name = f_name;
    _last_name = l_name;
  }

  public string FirstName {
    get { return _first_name; }
    set { _first_name = value; }
  }

  public string LastName {
    get { return _last_name; }
    set { _last_name = value; }
  }

  public string Breed {
    get { return _breed; }
```

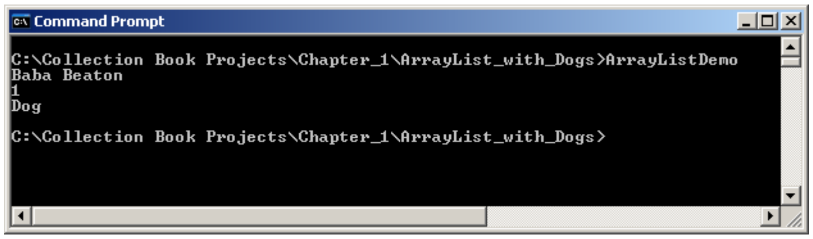
```
      set { _breed = value; }
    }

    public string FullName {
      get { return FirstName + " " + LastName; }
    }

    public string BreedAndFullName {
      get { return Breed + ": " + FullName; }
    }
  }
```

Referring to example 1.2 — the Dog class contains five public properties: FirstName, LastName, Breed, FullName, and BreedAndFullName. The first three are read-write properties and the last two are read-only. Example 1.3 shows an ArrayList being used to store various types of objects: string, integer, and Dog.

1.3 ArrayListDemo.cs (mod 1)

```
1       using System;
2       using System.Collections;
3
4       public class ArrayListDemo {
5         public static void Main(){
6           ArrayList list = new ArrayList();
7
8           //Add various types of objects to the ArrayList
9           list.Add("Baba Beaton");
10          list.Add(1);
11          list.Add(new Dog("Boxer", "Sammy", "Socks"));
12
13          //Access each object in the collection and print out its value
14          foreach(object o in list){
15            Console.WriteLine(o);
16          }
17        }
18      }
```

Referring to example 1.3 — notice now on lines 9 through 11 that I'm adding three different types of objects to the list. The `foreach` statement on line 14 iterates over the list and prints the value of each element to the console. But what value will be printed for the Dog object? Figure 1-2 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_1\ArrayList_with_Dogs>ArrayListDemo
Baba Beaton
1
Dog

C:\Collection Book Projects\Chapter_1\ArrayList_with_Dogs>
```

Figure 1-2: Results of Running Example 1.3

Referring to figure 1-2 — the reason this codes works at all is because when the value of each object is printed to the console, its ToString() method is called automatically. This is what happens when supplying an object as a argument to the overloaded Console.WriteLine() method.

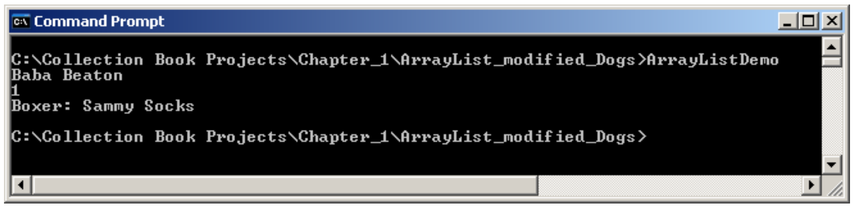
When using user-defined types where the Object.ToString() method has **not** been overloaded, the default behavior results in the class name being returned, which is what printed to the console in the case of the Dog object. If you want another, more meaningful, value to be printed to console instead of the class name, you must override the Object.ToString() method in the Dog class as example 1.4 illustrates.

1.4 Dog.cs (with overriden ToString() method)

```
  public class Dog {
    private string _first_name;
    private string _last_name;
    private string _breed;

    public Dog(string breed, string f_name, string l_name){
      _breed = breed;
      _first_name = f_name;
      _last_name = l_name;
    }

    public string FirstName {
      get { return _first_name; }
      set { _first_name = value; }
    }
```

```
16
17        public string LastName {
18          get { return _last_name; }
19          set { _last_name = value; }
20        }
21
22        public string Breed {
23          get { return _breed; }
24          set { _breed = value; }
25        }
26
27        public string FullName {
28          get { return FirstName + " " + LastName; }
29        }
30
31        public string BreedAndFullName {
32          get { return Breed + ": " + FullName; }
33        }
34
35        //override System.Object.ToString() method
36        public override string ToString(){
37          return BreedAndFullName;
38        }
39      }
```

Referring to example 1.4 — the ToString() method has been overridden on line 36 and returns the BreedAnd-FullName string. Figure 1-3 shows the results of running example 1.3 with this modified version of the Dog class.



```
C:\Collection Book Projects\Chapter_1\ArrayList_modified_Dogs>ArrayListDemo
Baba Beaton
1
Boxer: Sammy Socks

C:\Collection Book Projects\Chapter_1\ArrayList_modified_Dogs>
```

Figure 1-3: Results of Running Example 1.3 with Overridden ToString() Method in the Dog Class

Referring to figure 1-3 — notice now that you get a more meaningful value from a Dog object because of the overridden ToString() method.

Polymorphic Behavior

The previous example illustrated how, when manipulating a collection of objects, polymorphic behavior can be utilized to treat all objects as if they were the same type, usually some targeted base type. In the case of an ArrayList, it can hold any type of object, but all the contained objects, when accessed, are returned as a System.Object. Thus, as long as your code targets the System.Object interface, all the list's contained objects can be treated uniformly or polymorphically.

Casting to a Specific Type

If, however, you want to access a member on an object retrieved from an ArrayList that does not belong to the System.Object's interface, you must attempt to convert that object into the specified type with a casting operation. Example 1.5 shows how a list of Dog objects must be cast to the Dog type before Dog-specific interface properties can be accessed.

1.5 ArrayListCastingDemo.cs

```
using System;
using System.Collections;

public class ArrayListCastingDemo {
  public static void Main(){
    ArrayList list = new ArrayList();
    list.Add(new Dog("Boxer", "Sammy", "Socks"));
    list.Add(new Dog("Golden Retriever", "Woody", "Miller"));
    list.Add(new Dog("Yellow Lab", "Austin", "Miller"));

    //explicitly cast each retrieved object to the Dog type
    for(int i=0; i<list.Count; i++){
      Console.WriteLine(((Dog)list[i]).BreedAndFullName);
14          }
15
16          Console.WriteLine("------------------------");
17
18          //the foreach statement does the casting for you...
19          foreach(Dog d in list){
20            Console.WriteLine(d.BreedAndFullName);
21          }
22        }
23      }
```

Referring to example 1.5 — I inserted three Dog objects into the list. The `for` loop on line 12 shows how each object must be explicitly cast to the Dog type upon retrieval from the list before a Dog-specific interface member can be accessed. If you use a `foreach` statement the compiler does the casting for your if you specify the type of objects you want out of the list. For each of these casting operations to succeed, objects actually contained in the list must successfully cast to the specified type, otherwise you'll throw an InvalidCastException.

Qick Review

An ArrayList is a non-generic collection that behaves like an array on steroids. You can insert any type of object into an ArrayList, but upon retrieval, all contained objects are returned as type System.Object. If you want to access type-specific members on a retrieved object, you must cast the retrieved object to that type.

An ArrayList will dynamically grow to accommodate more objects. This dynamic growth is handled automatically. When the number of inserted objects reaches a certain threshold, the list is resized.

List<T> Class

The .NET framework 2.0 introduced generic collections. A generic collection allows you to specify the type of objects you want to insert into the collection and thus eliminates the need to cast upon retrieval. This "specific type in, specific type out" behavior characterizes the members of the generic collection classes.

The List<T> class is the generic replacement for the old-school ArrayList class. Nowadays, you should prefer the use of generics over the use of old-school collections when writing new code.

Example 1.6 demonstrates the use of the List<T> collection class.

1.6 ListTDemo.cs

```
1       using System;
2       using System.Collections.Generic;
3
4       public class ListTDemo {
5         public static void Main(){
6           List<String> list = new List<String>();
7
8           //Add elements using the Add() method
9           list.Add("Hope Mesa");
10          list.Add("Bill Hicks");
11          list.Add("Secret Miller");
12          list.Add("Alex Remily");
13          list.Add("Pete Luongo");
14
15          //access elements using array indexer notation
16          for(int i=0; i<list.Count; i++){
17            Console.WriteLine(list[i].ToUpper());
18          }
19
20          Console.WriteLine("------------------------------");
21
22          //or, use the foreach statement which hides the complexity of the enumerator
23          foreach(string s in list){
24            Console.WriteLine(s);
25          }
26        }
27      }
```

Referring to example 1.6 — a List<T> object is declared and created on line 6. The <T> in the angle brackets represents the type placeholder. Replace the T with whatever type you want the list to contain. In this example, I've specified a list of Strings (e.g., List<String>). If you're completely new to generics the syntax takes some getting used to. Note how you must repeat the type specification in the angle brackets when making the constructor call. Once you've created the list, you use it just like an ArrayList. The only exception is that now you know what types of objects the list contains because you specified the type when you created the list. This eliminates the need to cast retrieved objects from the list. The `for` loop on line 16 demonstrates this by calling the ToUpper() method on each string object in the list without casting to the string type. The `foreach` statement remains unchanged from earlier examples.

Quick Review

The generic List<T> class allows you to specify the type of objects the list will contain when you create the list. This eliminates the need to cast retrieved objects.

When writing new code, prefer the use of the generic collection classes over non-generic old-school collections.

Manipulating Lists

There's much more to the ArrayList and List<T> classes than just dynamic growth. Both classes provide a wide assortment of interface methods that allow you to manipulate the collection in many ways. In this section I will show you how to sort the elements of a list, search a list for a specific entry, and reverse the elements of a list. I will use a List<T> collection to demonstrate these operations but you can perform the same operations on an ArrayList.

Sorting, Searching, and Reversing a List<T> Collection

The sorting, searching, and reversing operations, along with many others, are part of the List<T> collection's interface. These three operations take the form of the Sort(), BinarySearch(), and Reverse() methods, all of which are overloaded, meaning there is more than one way to sort, search, and reverse a list.

Default Sorting Behavior

Before you sort a list, you must be aware of what type of objects the list contains. Before two objects can be compared with each other, they must implement the IComparer or IComparer<T> interface or there must exist one or more comparer objects that derive from System.Comparer or System.Comparer<T> that instructs the Sort() method on how to compare each object.

In the case of built-in .NET framework classes, you don't have to worry about such matters. Classes and structures that are meant to be sorted, like strings, integers, characters, etc., already implement the IComparer and IComparer<T> interfaces.

Sort Before Calling The BinarySearch() Method

The title of this section says it all. The BinarySearch() method works on a sorted list, so be sure you have sorted the list before calling the BinarySearch() method.

Example 1.7 demonstrates the use of the Sort(), BinarySearch(), and Reverse() methods on a list of strings.

1.7 ListManipulationDemo.cs

```
using System;
using System.Collections.Generic;

public class ListManipulationDemo {
  public static void Main(){
    List<String> list = new List<String>();

    //Add elements using the Add() method
    list.Add("Hope Mesa");
    list.Add("Bill Hicks");
    list.Add("Secret Miller");
    list.Add("Alex Remily");
    list.Add("Pete Luongo");

    //Access elements using array indexer notation
    for(int i=0; i<list.Count; i++){
```

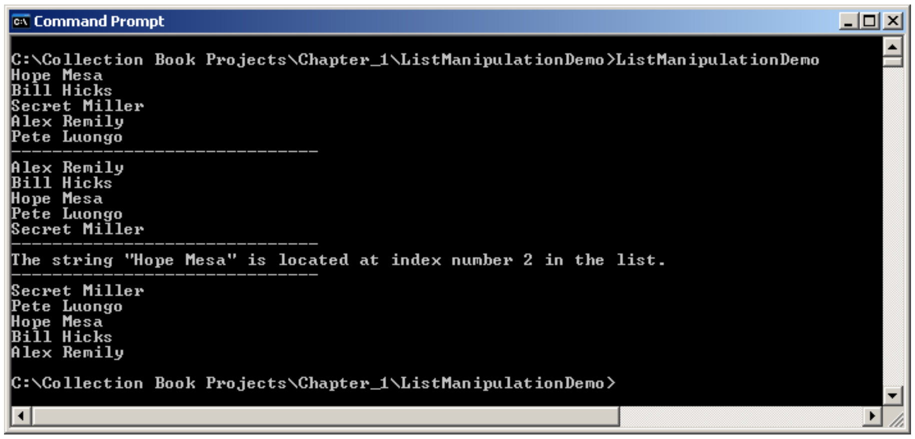
```
17            Console.WriteLine(list[i]);
18          }
19
20          Console.WriteLine("------------------------------");
21
22          //Sort the list using the natural ordering of the String class
23          list.Sort();
24
25          //Print the sorted list to the console
26          foreach(string s in list){
27            Console.WriteLine(s);
28          }
29
30           Console.WriteLine("------------------------------");
31
32           //Search the list for a specific string
33           Console.WriteLine("The string \"Hope Mesa\" is located at index number "
34                            + list.BinarySearch("Hope Mesa") + " in the list.");
35
36           Console.WriteLine("------------------------------");
37
38           //Now, reverse the list
39           list.Reverse();
40
41           //Print the reversed list to the console
42          foreach(string s in list){
43            Console.WriteLine(s);
44          }
45        }
46      }
```

Referring to example 1.7 — a list of strings is declared and created on line 6 and then populated with string objects on lines 9 through 13. The `for` loop on line 16 writes the strings to the console. On line 23, the list is sorted with a call to its Sort() method and the list of sorted strings is then written to the console.

Line 34 demonstrates the use of the BinarySearch() method. The BinarySearcy() method searches the list for the specified object, in this case a string (e.g., "Hope Mesa") and returns its index.

On line 39, the list is reversed with a call to the Reverse() method. The reversed list is then written to the console with the `foreach` statement on line 42.

Figure 1.4 shows the results of running this program.



```
C:\Collection Book Projects\Chapter_1\ListManipulationDemo>ListManipulationDemo
Hope Mesa
Bill Hicks
Secret Miller
Alex Remily
Pete Luongo
------------------------------
Alex Remily
Bill Hicks
Hope Mesa
Pete Luongo
Secret Miller
------------------------------
The string "Hope Mesa" is located at index number 2 in the list.
------------------------------
Secret Miller
Pete Luongo
Hope Mesa
Bill Hicks
Alex Remily

C:\Collection Book Projects\Chapter_1\ListManipulationDemo>
```

Figure 1-4: Results of Running Example 1.7

Quick Review

The ArrayList and List<T> collection classes provide a wide assortment of methods that let you manipulate the collections in many ways. Three helpful operations include Sort(), BinarySearch(), and Reverse().

Before one object can be compared to another they must be comparable. This means they must implement the IComparable or IComparable<T> interface or one or more comparer objects must exist that extend the Comparer or Comparer<T> classes.

.NET framework classes that are meant to be sorted already implement both the IComparable and IComparable<T> interfaces.

Remember to sort a list before calling the BinarySearch() method.

Where To Go From Here

If you haven't already done so, now would be a good time to explore the .NET framework documentation on the Microsoft Developer Network (MSDN) website (www.msdn.com). Pay particular attention to the members of the four collection namespaces: *System.Collections*, *System.Collections.Generic*, *System.Collections.ObjectModel*, and *System.Collections.Specialized*. Look up the ArrayList and List<T> classes and study their methods and properties.

Summary

An ArrayList is a non-generic collection that behaves like an array on steroids. You can insert any type of object into an ArrayList, but upon retrieval, all contained objects are returned as type System.Object. If you want to access type-specific members on a retrieved object, you must cast the retrieved object to that type.

An ArrayList will dynamically grow to accommodate more objects. This dynamic growth is handled automatically. When the number of inserted objects reaches a certain threshold, the list is resized.

The generic List<T> class allows you to specify the type of objects the list will contain when you create the list. This eliminates the need to cast retrieved objects.

When writing new code, prefer the use of the generic collection classes over non-generic old-school collections.

The ArrayList and List<T> collection classes provide a wide assortment of methods that let you manipulate the collections in many ways. Three helpful operations include Sort(), BinarySearch(), and Reverse().

Before one object can be compared to another they must be comparable. This means they must implement the IComparable or IComparable<T> interface or one or more comparer objects must exist that extend the Comparer or Comparer<T> classes.

.NET framework classes that are meant to be sorted already implement both the IComparable and IComparable<T> interfaces.

Remember to sort a list before calling the BinarySearch() method.

References

.NET Framework 3.5 Reference Documentation, Microsoft Developer Network (MSDN) [www.msdn.com]

Notes