Chapter 8 – Data Containers

# Objectives

* Arrays and Related Functionality
* Collections
* Dictionaries and SortedDictionaries
* Stacks and Queues
* Lists and SortedLists
* LinkedLists
* HashSets and SortedSets
* Hashtables
* ArrayLists
* Specialized Data Structures

So far we’ve pretty much limited ourselves to working with simple data types and classes. This chapter is going to examine how to start working with more rich types of data in Visual Basic that can be used to store large amounts of information. We will begin with the concept of arrays, which is a convenient way to hold large amounts of data that are all of a single type. We’ll move on to Collections, which are like bags that will hold whatever information is placed into them. From there, we’ll start venturing into some of the other built-in data structures that can save you a lot of time and effort.

# Arrays

Arrays are probably the simplest composite data type to work with. The idea is that an array can hold a number of items of the same type. For example, you can create an array that holds Integers or an array that holds Strings. About the only thing that you can't do is create an array and then try to place multiple data types in it – the rule is arrays only hold one type of data.

## Declaring Arrays

Arrays are declared much like other variables – using the Dim statement. Here are some examples of simple array declarations:

'Note that this array is 16 elements long

'where each element can hold a long

Dim lngSalary(15) As Long

'This array is also 16 elements long

'(elements 0 through 15) – each element in

'this array holds a string though

Dim txtNames(15) As String

'We can declare an array and initialize it at the same time –

'these next examples should be very familiar to Java programmers.

Dim Numbers As Integer() = New Integer() {1, 2, 4, 8}

Dim Names() As String = {"Scott", "Sue", "Pete"}

'We can also create a "placeholder" variable for an array and

'then actually allocate the space for it later on...

Dim MyArray As Double() 'Declare Array Variable

MyArray = New Double(10) {} 'Allocate space for it and default init

The first example set shows how to create two arrays, each of which has elements starting at (0) and ranging to (15). VB.NET forces all arrays to be zero based, hence you don't have the creation choices available to you like you did in older versions of VB. Therefore, be careful if you are porting older VB code to .NET: most arrays were 1 based, but could really start at any element number of the programmer’s choosing, which was kind of slick.

The second example set declared and allocated an array with 4 elements and then initialized it, all using one line of code. The next line in that example showed declaring and initializing a String array in one line of code.

Finally, the third example set declared an array on one line and then allocated the space (and initialized the array using the Double data type's default) on the second line.

Now that we have examined how to declare single dimension arrays, it should be no problem for you to expand your declaration knowledge into multi-dimensional arrays. Multi-dimensional arrays can be declared as follows:

'Creates an 11 by 11 array of integers (0 to 10 by 0 to 10)

Dim intBoard(10, 10) As Integer

'Declare an array variable and separately initialize it

'Notice row magnitude order is used when initializing...

Dim Array2D As Integer(,)

Array2D = New Integer(,) {{1, 2, 3}, {4, 5, 6}}

'The resulting array would look like this:

'+---+---+---+

'| 1 | 2 | 3 |

'+---+---+---+

'| 4 | 5 | 6 |

'+---+---+---+

While most of the time we want our arrays to be rectangular in shape, there is no reason they have to be. Many languages (including Java) support the idea of a jagged or ragged array. Think about the distance array you see on physical paper maps (or used to; maybe you’ll need to go to a museum) – it is not rectangular, but actually jagged. Why? (1) to save space – it doesn’t take up as much room on the map (or memory in the computer) to store “half” of a rectangular array and (2) there is a lot of redundant data, e.g. the distance between Chicago and New York is the same as the distance between New York and Chicago, so why show both? Put one value in the table and allow the user to index it either way.

'Create a "jagged" 2D array

'It will be a non-size specified 2D integer array which we

'will initialize to a 2D array of integers that has

'three rows (0 to 2), with each row holding another

'array of integers that varies in size.

Dim Jarray As Integer()() = New Integer(2)() {}

'Now initialize each "row" as hold an integer array with

'varying amounts of elements

Jarray(0) = New Integer() {1}

Jarray(1) = New Integer() {2, 3}

Jarray(2) = New Integer() {4, 5, 6}

'When we are done, the array would look like the following:

' +---+

' | 1 |

' +---+---+

' | 2 | 3 |

' +---+---+---+

' | 4 | 5 | 6 |

' +---+---+---+

## Array Element Referencing

One dimensional array elements can be individually referenced using the () notation as in MyArray(3) = MyArray(2) + MyArray(1). Multidimensional array elements work exactly the same way except that they have each dimension comma separated. So we could say something like MessageBox.Show(Array2D(1,2)), which in the rectangular example above would print out a "6"; the jagged example would throw an exception since there is no element (1,2)…

## Array Methods

VB.Net defines an Array class that has a ton of useful methods in it. How many times have you been burned in other programming languages by not knowing at runtime what the upper or lower bounds of an array were? VB solves all of this, plus you get built in support for searching and sorting – pretty cool, eh? Here is a list of the more relevant Array class methods and properties:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| AsReadOnly | Returns a read-only wrapper around the array |
| BinarySearch | Searches a one-dimensional array for a value (requires array to be sorted) |
| Clear | Sets a range of elements to zero, null or False |
| Clone | Returns a shallow copy of the array |
| ConstrainedCopy | Returns a range of elements from this array and sends them to another array at a programmer specified element index |
| ConvertAll | Converts an array of one type to an array of another type |
| Copy | Copies the entire array or a range of elements in a 1D array to another array |
| CopyTo | Specifies the starting element to copy to in the target array |
| Equal | Returns whether the specified array is equal to the current array |
| Exists | Determines whether the specified array contains elements that match some specified condition |
| Find | Searches for an element that matches some specified condition and returns the first occurrence |
| FindAll | Like Find, but returns all elements that match the specified condition |
| FindIndex | Searches for an element that matches some condition and returns the index of the first occurrence |
| FindLast | Searches for an element that matches some condition and returns the last occurrence |
| FindLastIndex | Searches for an element that matches some condition and returns the index of the last occurrence |
| GetLength | Returns the length of the array (32-bit) |
| GetLongLength | Returns the length of the array (64-bit) |
| GetLowerBound | Returns the lower bound element of a dimension |
| GetUpperBound | Returns the upper bound element of a dimension |
| GetValue | Returns the value at a specified index in any dimension |
| IndexOf | Returns the index at the first occurrence of the value searched for |
| IsFixedSize | Does the array have a fixed size? |
| IsReadOnly | Is the array read only? |
| LastIndexOf | Returns the index at the last occurrence of the value searched for |
| Length | Returns the length of a 1D array |
| Rank | Returns the number of dimensions in the array |
| Reverse | Reverses the elements in a 1D array |
| SetValue | Sets the value at the specified index in an array |
| Sort | Performs the built-in quicksort sorting operation on a one-dimensional array |
| ToArray | Converts the elements to a new array |
| TrueForAll | Determines whether every element in the array matches some defined condition |

Here's a snippet of code that exercises some of the nicer features of the class:

Dim intArray As Integer() = {95, 45, 23, 15, 60, 23}

Dim intLoop As Integer

Debug.WriteLine("Lower Bound is: " & intArray.GetLowerBound(0))

Debug.WriteLine("Upper bound is: " & intArray.GetUpperBound(0))

Debug.WriteLine("Length is: " & intArray.GetLength(0))

Debug.WriteLine("Rank of array " & intArray.Rank)

Debug.Write("Array before sort: ")

For intLoop = intArray.GetLowerBound(0) To intArray.GetUpperBound(0)

Debug.Write(intArray(intLoop) & " ")

Next

Debug.WriteLine("")

Debug.Write("Array after sort: ")

'Sort the array

Array.Sort(intArray)

'Notice we are printing using iterator for-next syntax

For Each intNum In intArray

Debug.Write(intNum & " ")

Next

Debug.WriteLine("")

'Let's ask the array some questions...

Debug.WriteLine("Location of 23 in array? " &

Array.BinarySearch(intArray, 23))

Debug.WriteLine("First occurrence of 23: " & Array.IndexOf(intArray, 23))

Debug.WriteLine("Last occurrence of 23: " &

Array.LastIndexOf(intArray, 23))

Debug.WriteLine("Does the array contain a 45? " & intArray.Contains(45))

Output:

Lower Bound is: 0

Upper bound is: 5

Length is: 6

Rank of array 1

Array before sort: 95 45 23 15 60 23

Array after sort: 15 23 23 45 60 95

Location of 23 in array? 2

First occurrence of 23: 1

Last occurrence of 23: 2

Does the array contain a 45? True

Because the array implements what is known as an enumerable interface (e.g. we could use the for-next syntax to “walk” across all of the elements), we have access to a host of other very cool enumerable features, many of which are common programming tasks we usually need to code. We will only look at a few of these methods now, but realize that we will visit most of the full set when we talk about Linq (Language INtegrated Query).

Select Enumerable methods pertaining to arrays:

|  |  |
| --- | --- |
| *Name* | *Description* |
| [Average](http://msdn.microsoft.com/en-us/library/bb354760.aspx) | Computes the average of a sequence of numeric values. |
| [Concat](http://msdn.microsoft.com/en-us/library/bb302894.aspx) | Concatenates two sequences. |
| [Contains](http://msdn.microsoft.com/en-us/library/bb352880.aspx) | Determines whether a sequence contains a specified element by using the default equality comparer. |
| [Count](http://msdn.microsoft.com/en-us/library/bb338038.aspx) | Returns the number of elements in a sequence. |
| [ElementAt](http://msdn.microsoft.com/en-us/library/bb299233.aspx) | Returns the element at a specified index in a sequence. |
| [ElementAtOrDefault](http://msdn.microsoft.com/en-us/library/bb494386.aspx) | Returns the element at a specified index in a sequence or a default value if the index is out of range. |
| [Empty](http://msdn.microsoft.com/en-us/library/bb341042.aspx) | Returns an empty [IEnumerable(OfT)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) that has the specified type argument. |
| [Except](http://msdn.microsoft.com/en-us/library/bb300779.aspx) | Produces the set difference of two sequences by using the default equality comparer to compare values. |
| [First](http://msdn.microsoft.com/en-us/library/bb291976.aspx) | Returns the first element of a sequence. |
| [FirstOrDefault](http://msdn.microsoft.com/en-us/library/bb340482.aspx) | Returns the first element of a sequence, or a default value if the sequence contains no elements. |
| [Intersect](http://msdn.microsoft.com/en-us/library/bb460136.aspx) | Produces the set intersection of two sequences by using the default equality comparer to compare values. |
| [Last](http://msdn.microsoft.com/en-us/library/bb358775.aspx) | Returns the last element of a sequence. |
| [LastOrDefault](http://msdn.microsoft.com/en-us/library/bb301849.aspx) | Returns the last element of a sequence, or a default value if the sequence contains no elements. |
| [LongCount](http://msdn.microsoft.com/en-us/library/bb353539.aspx) | Returns an [Int64](http://msdn.microsoft.com/en-us/library/system.int64.aspx) that represents the total number of elements in a sequence. |
| [Max](http://msdn.microsoft.com/en-us/library/bb335614.aspx) | Returns the maximum value in a sequence of numeric values. |
| [Min](http://msdn.microsoft.com/en-us/library/bb298087.aspx) | Returns the minimum value in a sequence of numeric values. |
| [Reverse](http://msdn.microsoft.com/en-us/library/bb358497.aspx) | Inverts the order of the elements in a sequence. |
| [Sum](http://msdn.microsoft.com/en-us/library/bb298138.aspx) | Computes the sum of a sequence of [numeric](http://msdn.microsoft.com/en-us/library/system.decimal.aspx) values. |
| [ToArray](http://msdn.microsoft.com/en-us/library/bb298736.aspx) | Creates an array from a[IEnumerable(Of T)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx). |
| [ToDictionary](http://msdn.microsoft.com/en-us/library/bb549277.aspx) | Creates a [Dictionary(Of TKey, TValue)](http://msdn.microsoft.com/en-us/library/xfhwa508.aspx) from an [IEnumerable(OfT)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) according to a specified key selector function. |
| [ToList](http://msdn.microsoft.com/en-us/library/bb342261.aspx) | Creates a [List(Of T)](http://msdn.microsoft.com/en-us/library/6sh2ey19.aspx) from an[IEnumerable(Of T)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx). |
| [ToLookup](http://msdn.microsoft.com/en-us/library/bb549073.aspx) | Creates a [Lookup(Of TKey, TElement)](http://msdn.microsoft.com/en-us/library/bb460184.aspx) from an[IEnumerable(Of T)](http://msdn.microsoft.com/en-us/library/9eekhta0.aspx) according to a specified key selector function. |
| [Union](http://msdn.microsoft.com/en-us/library/bb341731.aspx) | Produces the set union of two sequences by using the default equality comparer. |

Here’s some code showing the uses of various enumerable methods:

Dim intArray1 As Integer() = {1, 7, 15, 23}

Dim intArray2 As Integer() = {9, 23, 47}

Dim intArray3 As Integer()

Debug.WriteLine("Basic statistics on intArray1")

Debug.WriteLine("Min of intArray1 : " & intArray1.Min())

Debug.WriteLine("Max of intArray1 : " & intArray1.Max())

Debug.WriteLine("Sum of intArray1 : " & intArray1.Sum())

Debug.WriteLine("Count of intArray1 : " & intArray1.Count())

Debug.WriteLine("Average of intArray1 : " & intArray1.Average())

Debug.WriteLine("First in intArray1 : " & intArray1.First())

Debug.WriteLine("Last in intArray1 : " & intArray1.Last())

Debug.Write("Array1 concat Array2 ")

'Make sure that the concatentation returned is an array

intArray3 = (intArray1.Concat(intArray2)).ToArray()

For Each intValue In intArray3

Debug.Write(intValue & " ")

Next

Debug.WriteLine("")

Debug.Write("Array1 union Array2 ")

intArray3 = (intArray1.Union(intArray2)).ToArray()

For Each intValue In intArray3

Debug.Write(intValue & " ")

Next

Debug.WriteLine("")

Debug.Write("Array1 intersect Array2 ")

intArray3 = (intArray1.Intersect(intArray2)).ToArray()

For Each intValue In intArray3

Debug.Write(intValue & " ")

Next

Debug.WriteLine("")

Debug.Write("Array1 except Array2 ")

intArray3 = (intArray1.Except(intArray2)).ToArray()

For Each intValue In intArray3

Debug.Write(intValue & " ")

Next

Debug.WriteLine("")

Debug.WriteLine("Does intArray3 have a 47 in it? " &

intArray3.Contains(47))

Debug.WriteLine("Print intArray3's element(0)) " & intArray3.ElementAt(0))

Debug.WriteLine("Print intArray3's default (3)) " &

intArray3.ElementAtOrDefault(3))

Debug.Write("Array3 printed normal: ")

For Each intValue In intArray3

Debug.Write(intValue & " ")

Next

Debug.WriteLine("")

Debug.Write("Array3 printed reversed: ")

intArray3 = (intArray3.Reverse).ToArray()

For Each intValue In intArray3

Debug.Write(intValue & " ")

Next

Debug.WriteLine("")

Here’s the output from the program:

Basic statistics on intArray1

Min of intArray1 : 1

Max of intArray1 : 23

Sum of intArray1 : 46

Count of intArray1 : 4

Average of intArray1 : 11.5

First in intArray1 : 1

Last in intArray1 : 23

Array1 concat Array2 1 7 15 23 9 23 47

Array1 union Array2 1 7 15 23 9 47

Array1 intersect Array2 23

Array1 except Array2 1 7 15

Does intArray3 have a 47 in it? False

Print intArray3's element(0)) 1

Print intArray3's default (3)) 0

Array3 printed normal: 1 7 15

Array3 printed reversed: 15 7 1

Pretty easy and slick considering that we didn’t have to write “real” code to get any of this carried out!

## Dynamic Arrays

One of the most commonly encountered problems in working with arrays is the fact that we often don't know how big to make an array. If we make the array too big and don't fill it with data, then we are wasting space. If we make the array too small and attempt to overfill it, then we will get a programming error. Dynamic arrays fix these problems by allowing the size of the array to be changed at runtime, and VB supports this construct.

To declare the most useful dynamic array, begin with one of the following array definitions:

Dim MyArray(,,) 'Notice no element size and no data type were

'given but we did specify the number of dimensions

'using the comma placeholders.

Dim MyArray2(,,) As Integer 'You can (and probably should) specify

'the data type of the array to avoid any

'confusion later on.

Either array can then be resized later while your program is running by using the ReDim command:

ReDim MyArray(9, 9, 9) 'Note – The array is 3D with a total of 1000

'elements in it: (0 to 9) x (0 to 9) x (0 to 9)

Once you ReDim an array, its number of dimensions cannot be changed. The other nice thing regarding the first array declaration that I showed you, MyArray, is that it allows any type of data to be placed in the array since we didn't specify a type. We can stick in anything that we want at runtime – once we have put in a single element though, the data type has been determined and we can't change the type of that array.

If you want to keep the contents of the array intact while you are enlarging the array, you can use the Preserve keyword. There is one limitation in doing this however, and that is that fact that you can only change the size of the last dimension on a multi-dimensional array when using the Preserve keyword.

ReDim Preserve MyArray(9, 9, 10)

Finally, you can also deallocate the memory allocated for a dynamic array by using the Erase keyword as show in the following example:

Erase MyArray

A dynamic array that has been Erased will need to be ReDimmed before it can be used again. While there are a few caveats to be aware of when working with dynamic arrays, the benefits clearly outweigh the issues.

## Checking to See If a Variable is an Array

Remember that Visual Basic provides a function named IsArray() that will return a Boolean value that indicates whether or not a variable is an array. For example:

Dim MyArray(10) As String

Dim intNum As Integer

Debug.WriteLine(IsArray(MyArray)) 'True

Debug.WriteLine(IsArray(intNum)) 'False – this is a string

## Passing Arrays to Subprograms

When you pass an array to a subprogram it is passed by reference and there's not really anything you can do about it. Here's a sample code snippet showing how to pass an entire array from one part of a program to a subprogram. Notice the ByRef in the subprogram heading.

Sub SortIt(ByRef MyArray() As Integer)

'Do something with the array here

End Sub

.

.

.

SortIt(MyArray)

While you can try deleting the ByRef from the heading, which defaults to a ByVal or explicitly type in ByVal, neither change will matter – arrays are automatically passed by reference! If you really want to protect the data in your array from being changed inside of a subprogram, there are a couple of different approaches you can take.

1. An obvious thing to do is to make a copy of the array and pass the copy to the subprogram. The only downside to this tactic is that you will use up more memory since you will have two copies of an array.
2. The second option is to just send the elements you need. This is useful if you don't need to send a whole array to a subprogram, but rather you want to send just an element or two. Since an element is usually a simple data type, you can pass it to the subprogram by value if you want.
3. Finally, the third trick which only works on one-dimensional arrays would be to make the array AsReadOnly and send that copy through to the routine that tries to change it. Even if the argument says ByRef, it behaves just like a ByVal now:

Module Module1

Sub TryToChange(ByRef MyArray() As Integer)

Dim intLoop As Integer

Debug.Write("In TryToChange: ")

For intLoop = 0 To 9

MyArray(intLoop) \*= 2

Debug.Write(MyArray(intLoop) & " ")

Next

Debug.WriteLine(" ")

End Sub

Sub Main()

Dim MyArray() As Integer = {1, 2, 3, 4, 5, 10, 20, 30, 40, 50}

Dim intLoop As Integer

'We call the Array AsReadOnly to put a read only wrapper

'around the original array and then cast the returned object

'into 1-D array so we can send it through to TryToChange

TryToChange(Array.AsReadOnly(MyArray).ToArray())

Debug.Write("In Main: ")

For intLoop = 0 To 9

Debug.Write(MyArray(intLoop) & " ")

Next

Debug.WriteLine(" ")

End Sub

End Module

The output from the code:

In TryToChange: 2 4 6 8 10 20 40 60 80 100

In Main: 1 2 3 4 5 10 20 30 40 50

## ParamArrays

Another array related item is that of the ParamArray statement which is useful when an unknown number of arguments of the same type need to be passed to a subprogram. ParamArrays can only be one dimensional arrays.

The following example shows multiple calls being made to the MySub subroutine. The interesting thing to observe is that one call has no arguments, while another has 10 arguments. Programmatically, this is easy to handle since we can declare a variable to be of type ParamArray in the subroutine heading and then simply use our Array class methods to determine how many parameters are there.

Sub MySub(ByVal ParamArray array1 As Integer())

Dim intLoop As Integer

'Figure out how many elements are in the array and then iterate

'through them using a For loop

For intLoop = array1.GetLowerBound(0) To array1.GetUpperBound(0)

Debug.Write(array1(intLoop) & " ")

Next

Debug.WriteLine(" ")

End Sub

Sub Main()

MySub()

MySub(1)

MySub(1, 2, 3)

MySub(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

End Sub

Output:

1

1 2 3

1 2 3 4 5 6 7 8 9 10

While it might seem restrictive that you can only receive parameters of the same type, you could create the ParamArray to be an array of String to accommodate "different" data types. While it's true that everything would arrive as a string, you could write conversion functions to convert string data back into other types. Therefore, calls such as MySub(5,16.5,"Scott") and MySub("Bill",2,"Sue",3) would pose no problems.

# Collections

Collections are the first of a series of built-in .NET data types that we want to examine. Collections prove to be a very useful data structure in comparison to arrays. Collections can hold any kind of data and the data can be placed into the collection with a key which will quickly allow the associated data to be returned. This makes retrieving information from a collection very easy.

For example, let’s say that we want to know what the temperature is in Detroit. It sure would be nice to be able to say Temperature(“Detroit”) in a temperature array and get the information back – while arrays can’t do this, collections can.

Collection Class Methods/Properties:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Add | Adds items to the collection |
| Clear | Empties the collection |
| Contains | Returns whether or not the collection contains a particular key |
| Count | Returns the last item number in the collection (It is 1 based) |
| Item | Returns an item by index or by key (if the key doesn't exist, you will get an exception) |
| Remove | Deletes an item from the collection by index or by key |

Here’s an example of implementing a collection that will store the temperatures of some cities in Michigan. After the collection is created, the user will be allowed to input a city that s/he wants to know the temperature of. The example will conclude with a few methods will be illustrated:

Sub Main()

Dim Temperatures As New Collection

Dim strCity As String

Dim intTemp As Integer

Dim intTotal As Integer

'Load the collection up...

'The first piece is the data, second is the key

Temperatures.Add(76, "Detroit")

Temperatures.Add(85, "Flint")

Temperatures.Add(80, "Saginaw")

Temperatures.Add(79, "SVSU")

Try

strCity = InputBox("What City?")

'Look up city name in collection to get temperature

intTemp = Temperatures.Item(strCity)

MsgBox(intTemp)

Catch ex As Exception

MsgBox("City does not exist in the collection!")

End Try

'Now let's play with some of the collection methods:

'Zap an entry by key (or we could delete it by index)

Temperatures.Remove("SVSU") 'or Temperatures.Remove(3)

'Iterate through with For Each to get average temperature of cities

For Each intTemp In Temperatures

intTotal = intTotal + intTemp 'intTemp contains the default

'value in the current

'collection item being

'examined

Next

MsgBox("There are " & Temperatures.Count & " items")

MsgBox("The average temperature is " & intTotal / Temperatures.Count)

MsgBox("The value for Saginaw is found? " &

Temperatures.Contains("Saginaw"))

Temperatures.Remove(3)

MsgBox("There are now " & Temperatures.Count & " items")

MsgBox("The value for Saginaw is found? " &

Temperatures.Contains("Saginaw"))

End Sub

So, while the collection allows us to have any kind of data that we want placed in it searchable by a key, it does have a limitation: the data can only be iterated over in the order it was placed in – there is no way to tell the collection to sort itself, which could be a big problem if we wanted to traverse over the data in key order. Another item of interest is that we need to realize that the collection is 1 based – a throwback from pre-.NET VB. I am sure it is there for compatibility, but that’s still a pretty big drag. Also, we didn’t get any of the nice .Min/.Max type of functionality that the Array data type gave us.

# Dictionaries

A dictionary is another fairly old Visual Basic data structure, but a step up above the collection. Like the collection, a dictionary can hold any type of data with a key to locate the data. One of the big differences though is that we can enumerate over the keys container or the items container in the dictionary to print things out. We also can ask if a particular key or value exists within the dictionary too.

For simplicity’s sake, let’s stay with the temperatures example that we looked at above.

Sub Main()

'We create the dictionary and specify the type of key and

'then the type of data that will be going in

Dim Temperatures As New Dictionary(Of String, Integer)

'Load the dictionary up...

'The first piece is the key, second is the data -- this is opposite

'of the collection’s Add method!

Temperatures.Add("SVSU", 79)

Temperatures.Add("Flint", 85)

Temperatures.Add("Detroit", 76)

Temperatures.Add("Saginaw", 80)

'Look up city name in dictionary to get temperature

Debug.WriteLine("SVSU's temperature is " & Temperatures.Item("SVSU"))

'Delete SVSU from dictionary

Temperatures.Remove("SVSU")

'OK now we can walk over the values in the order we put them in...

Debug.Write("Walking over dictionary by values (in order added): ")

For Each intTemp In Temperatures.Values

Debug.Write(intTemp & " ")

Next

Debug.WriteLine(" ")

'Walk over dictionary by keys in order added

Debug.Write("Walking over dictionary by keys (in order added): ")

For Each strKey In Temperatures.Keys

Debug.Write(strKey & " ")

Next

Debug.WriteLine(" ")

'Walk over dictionary items using keys (still in order added)

Debug.Write("Walking over temperature by items (using keys in" &

"order added): ")

For Each strKey In Temperatures.Keys

Debug.Write(Temperatures.Item(strKey) & " ")

Next

Debug.WriteLine(" ")

Debug.WriteLine("There are " & Temperatures.Count & " items")

Debug.WriteLine("The value for Saginaw is found? " &

Temperatures.ContainsKey("Saginaw"))

End Sub

Here’s the output:

SVSU's temperature is 79

Walking over dictionary by values (in order added): 85 76 80

Walking over dictionary by keys (in order added): Flint Detroit Saginaw

Walking over temperature by items (using keys in order added): 85 76 80

There are 3 items

The value for Saginaw is found? True

Notice that regardless if we iterate over the data using the values, keys or items container, the data is always printed out in the order it was added to the dictionary. What’s the big deal of using the Dictionary over a Collection then? The advantage is that we can look up entries by either keys or values containers now. Collections did not return a keys container to us – we could only iterate over the items. Furthermore, collections allows you to access an item via an index number or key; dictionaries only permit item access via a key.

Dictionary Class Methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Add | Adds items to the dictionary |
| Clear | Empties the dictionary |
| ContainsKey | Returns whether or not the dictionary contains a particular key |
| ContainsValue | Returns whether or not the dictionary contains a particular value |
| Count | Returns the number of items in the dictionary |
| Item | Returns an item by key (if the key doesn't exist you will get an exception) |
| Keys | An enumerable set of all keys in the dictionary (in key order) |
| Remove | Deletes an item from the dictionary by index or by key |
| TryGetValue | Attempts to return a value from the dictionary, but if it doesn’t exist a catchable ArgumentNullException is thrown |
| Values | An enumerable set of all values in the dictionary (in key order) |

## SortedDictionary

The SortedDictionary class fixes the big problem that we saw in the dictionary – the keys will now be stored in dictionary order rather than in inserted order! The only difference in the code that I made was this change:

Dim Temperatures As New SortedDictionary(Of String, Integer)

Notice the difference in the output:

SVSU's temperature is 79

Walking over dictionary by values (in key order): 76 85 80

Walking over dictionary by keys (in key order): Detroit Flint Saginaw

Walking over temperature via key order: 76 85 80

There are 3 items

The value for Saginaw is found? True

You may have noticed there isn’t a methods/properties section listed here…that’s because they are essentially the same between the Dictionary and SortedDictionary data types.Stacks and Queues

Your friend the stack is also implemented in .NET. You remember the stack: you push values on and you pop them off. You only have access to the value that is at the top of the stack. Here is a simple Stack program to work with 5 values on the stack:

Sub Main()

'Notice we are creating a stack data structure

Dim aStack As New Stack() 'or Dim aStack as New Stack(Of String)

Dim intLoop As Integer

Dim aString As String

Dim strSentence As String

'Push

aStack.Push("Furiously")

aStack.Push("Sleep")

aStack.Push("Ideas")

aStack.Push("Green")

aStack.Push("Colorless")

aStack.Push("The")

Debug.WriteLine("There are " & aStack.Count & " items on the stack.")

'Check for membership on the stack

Debug.WriteLine("Does the stack contain the word The? : " &

aStack.Contains("The"))

'Figure out how many items using Count

For intLoop = 1 To aStack.Count

'Pop off an item

aString = aStack.Pop()

'Append it to the strSentence

strSentence &= aString & " "

Next

Debug.WriteLine(strSentence)

End Sub

Output:

There are 6 items on the stack.

Does the stack contain the word The? : True

The Colorless Green Ideas Sleep Furiously

Stack Class Methods/Properties:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Clear | Empties the stack |
| Clone | Returns a shallow copy of the stack |
| Contains | Returns whether or not the stack contains a particular item |
| Count | Returns the number of items on the stack |
| Peek | Returns the object at the top of the stack without removing it |
| Pop | Removes and returns the object from the top of the stack |
| Push | Inserts an object at the top of the stack |
| ToArray | Copies the stack to a new array |

## Queues

Queues, as you recall, have values placed in one end and removed out the other end. Remember that queues are like being in line at a grocery store – you enter the queue and all of the people in line in front of you have to be helped before you get your chance to be helped.

Sub Main()

'Notice we are creating a queue data structure

Dim aQueue As New Queue() 'or Dim aQueue as New Queue(Of String)

Dim aString As String

Dim intLoop As Integer

'Add

aQueue.Enqueue("One")

aQueue.Enqueue("Two")

aQueue.Enqueue("Three")

aQueue.Enqueue("Four")

aQueue.Enqueue("Five")

Debug.WriteLine("There are " & aQueue.Count & " items on the queue.")

'Check for Membership on the queue

Debug.WriteLine("Does the queue contain the word Three? : " &

aQueue.Contains("Three"))

'Figure out how many items using Count

For intLoop = 1 To aQueue.Count

'Remove an item

aString &= aQueue.Dequeue & " "

Next

Debug.WriteLine(aString)

End Sub

Output:

There are 5 items on the queue.

Does the queue contain the word Three? : True

One Two Three Four Five

Queue Class Methods/Properties:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Clear | Empties the queue |
| Clone | Returns a shallow copy of the queue |
| Contains | Returns whether or not the queue contains a particular item |
| Count | Returns the number of items in the queue |
| Dequeue | Removes and returns the object at the beginning/start of the queue |
| Enqueue | Adds an object to the end of the queue |
| Peek | Returns the object at the beginning of the queue without removing it |
| ToArray | Copies the queue to a new array |
| TrimToSize | Sets the capacity of the queue to the number of elements in the queue |

# Lists

Continuing along with the basic data structures you’ve seen in CS 216, it’s pretty obvious that both stacks and queues can be implemented as arrays. The List structure is another one that can be implemented as an array as well. Here are the properties and methods that we are interested on with a list – notice that many come from Array:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Add | Adds an object to the end of the list |
| AsReadOnly | Returns a read-only wrapper around the list (not surprising since it’s basically an array) |
| BinarySearch | Searches a list for a value (requires a list to be sorted first) |
| Clear | Removes all elements from the list |
| Contains | Determines whether an element is in the list |
| ConvertAll | Converts a list of one type to a list of another type |
| CopyTo | Copies the entire list to a target array |
| Count | Returns the number of elements on the List |
| Equals | Returns whether the specified list is equal to the current list |
| Exists | Determines whether the specified list contains elements that match some specified conditions. |
| Find | Searches for an element that matches some specified condition and returns the first occurrence |
| FindAll | Like Find, but returns all elements that match the specified condition |
| FindIndex | Searches for an element that matches some condition and returns the index of the first occurrence |
| FindLast | Searches for an element that matches some condition and returns the last occurrence |
| FindLastIndex | Searches for an element that matches some condition and returns the index of the last occurrence |
| IndexOf | Returns the index at the first occurrence of the value searched for |
| LastIndexOf | Returns the index at the last occurrence of the value searched for |
| MemberwiseClone | Creates a shallow copy of the current object |
| Remove | Removes the first occurrence of a specific element from the list |
| RemoveAll | Removes all the elements that match the conditions defined by the specified rule |
| Reverse | Reverses the elements in the list |
| Sort | Performs the built-in quicksort sorting operation on the list |
| ToArray | Copies the elements from the List into a new array |
| TrimExcess | Sets the capacity to the number of the actual elements in the list |
| TrueForAll | Determines whether every element in the array matches some defined condition |

Here’s a simple program showing the use of many of the List methods:

Sub Main()

Dim aList As New List(Of String)

aList.Add("Tom")

aList.Add("Bill")

aList.Add("Sue")

aList.Add("Elizabeth")

aList.Add("Robert")

Debug.Write("The original list : ")

For Each strItem In aList

Debug.Write(strItem & " ")

Next

Debug.WriteLine(" ")

Debug.WriteLine("The list has " & aList.Count & " items on it")

Debug.WriteLine("Tom is on the list? : " & aList.Contains("Tom"))

aList.RemoveAt(0) 'Bye Tom -- note we can use index numbers

aList.Sort()

Debug.Write("The sorted list : ")

For Each strItem In aList

Debug.Write(strItem & " ")

Next

Debug.WriteLine(" ")

Debug.WriteLine("The list has " & aList.Count & " items on it")

Debug.WriteLine("Tom is on the list? : " & aList.Contains("Tom"))

Debug.WriteLine("Binary Search for Tom found at element : " &

aList.BinarySearch("Tom"))

Debug.WriteLine("Binary Search for Elizabeth found at element : " &

aList.BinarySearch("Elizabeth"))

End Sub

Here’s the output:

The original list : Tom Bill Sue Elizabeth Robert

The list has 5 items on it

Tom is on the list? : True

The sorted list : Bill Elizabeth Robert Sue

The list has 4 items on it

Tom is on the list? : False

Binary Search for Tom found at element : -5

Binary Search for Elizabeth found at element : 1

Notice that the list elements are 0 based...

## SortedList

The SortedList is a cross between a List and a SortedDictionary: items are entered in Key/Value pairs and are kept in a key-sorted order as they are placed on the list. Here’s the last program again, modified to use a SortedList structure:

Sub Main()

Dim aList As New SortedList()

'Key/Value pairs

aList.Add("Tom", "Tom")

aList.Add("Bill", "Bill")

aList.Add("Sue", "Sue")

aList.Add("Elizabeth", "Elizabeth")

aList.Add("Robert", "Robert")

Debug.Write("The original list : ")

For Each strItem In aList.Keys

Debug.Write(strItem & " ")

Next

Debug.WriteLine(" ")

Debug.WriteLine("The list has " & aList.Count & " items on it")

Debug.WriteLine("Bill is on the list? : " & aList.Contains("Bill"))

aList.RemoveAt(0) 'Bye Bill

Debug.Write("The sorted list : ")

For Each strItem In aList.Keys

Debug.Write(strItem & " ")

Next

Debug.WriteLine(" ")

Debug.WriteLine("The list has " & aList.Count & " items on it")

Debug.WriteLine("Bill is on the list? : " & aList.Contains("Bill"))

Debug.WriteLine("Search for Bill found at element : " &

aList.IndexOfKey("Bill"))

Debug.WriteLine("Search for Elizabeth found at element : " &

aList.IndexOfKey("Elizabeth"))

End Sub

Output:

The original list : Bill Elizabeth Robert Sue Tom

The list has 5 items on it

Bill is on the list? : True

The sorted list : Elizabeth Robert Sue Tom

The list has 4 items on it

Bill is on the list? : False

Search for Bill found at element : -1

Search for Elizabeth found at element : 0

As you might guess, the properties/methods are very similar to the SortedDictionary:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Add | Adds items to the SortedList |
| Clear | Empties the SortedList |
| ContainsKey | Returns whether or not the SortedList contains a particular key |
| ContainsValue | Returns whether or not the SortedList contains a particular value |
| Count | Returns the last item number in the list |
| Item | Returns an item by key (if the key doesn’t exist you will get an exception) |
| IndexOfKey | Searches for a key and returns the zero-based index within the SortedList |
| IndexOfValue | Searches for a value and returns the zero-based index within the SortedList |
| MemberwiseClone | Creates a shallow copy of the SortedList |
| Keys | An enumerable set of all keys in the list (in key order) |
| Remove | Deletes an item from the SortedList by key |
| RemoveAt | Removes the element at the specified index |
| TrimExcess | Sets the capacity to the number of elements in the SortedList |
| TryGetValue | Attempts to return a value from the SortedList, but if it doesn’t exist a catchable ArgumentNullException is thrown |
| Values | An enumerable set of all values in the SortedList (in key order) |

# LinkedList

As previously mentioned the former lists were handled through an array data structure solution. Visual Basic does off a true linked list that built-in if that’s what you are after…

Sub Main()

Dim aList As New LinkedList(Of String)

aList.AddLast("Tom")

aList.AddLast("Bill")

aList.AddLast("Sue")

aList.AddLast("Elizabeth")

aList.AddLast("Robert")

Debug.Write("The original list : ")

For Each strItem In aList

Debug.Write(strItem & " ")

Next

Debug.WriteLine(" ")

Debug.WriteLine("The list has " & aList.Count & " items on it")

Debug.WriteLine("Tom is on the list? : " & aList.Contains("Tom"))

aList.RemoveFirst() 'Bye Tom

Debug.Write("The new list : ")

For Each strItem In aList

Debug.Write(strItem & " ")

Next

Debug.WriteLine(" ")

Debug.WriteLine("The list has " & aList.Count & " items on it")

Debug.WriteLine("Tom is on the list? : " & aList.Contains("Tom"))

Debug.WriteLine("Min list value : " & aList.Min())

Debug.WriteLine("Max list value : " & aList.Max())

End Sub

The output:

The original list : Tom Bill Sue Elizabeth Robert

The list has 5 items on it

Tom is on the list? : True

The new list : Bill Sue Elizabeth Robert

The list has 4 items on it

Tom is on the list? : False

Min list value : Bill

Max list value : Sue

LinkedList Properties and Methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| AddAfter | Adds the new node after the specified existing node |
| AddBefore | Adds the new node before the specified existing node |
| AddFirst | Adds the new node at the start of the list |
| AddLast | Adds the new node at the end of the list |
| Average | Returns the average value of the list |
| Clear | Removes all elements from the list |
| Contains | Determines whether an element is in the list |
| ConvertAll | Converts a list of one type to a list of another type |
| CopyTo | Copies the entire list to a target array |
| Count | Returns the number of elements on the List |
| Equals | Returns whether the specified list is equal to the current list |
| Find | Searches for an element that matches some specified condition and returns the first occurrence |
| FindLast | Searches for an element that matches some condition and returns the last occurrence |
| First | Gets the first node of the linked list |
| IndexOf | Returns the index at the first occurrence of the value searched for |
| Last | Gets the last node of the linked list |
| LastIndexOf | Returns the index at the last occurrence of the value searched for |
| Max | The largest value on the list |
| MemberwiseClone | Creates a shallow copy of the current object |
| Min | The smallest value on the list |
| Remove | Removes the first occurrence of a specific element from the list |
| RemoveFirst | Removes the first node from the list |
| RemoveLast | Removes the last node from the list |
| Sum | The total of all values on the list added together |

# HashSets and SortedSets

A set is a group of distinct values – think math sets. We can carry out operations such as intersections and unions on them.

Public Sub Main()

Dim evenNumbers As HashSet(Of Integer) = New HashSet(Of Integer)()

Dim oddNumbers As HashSet(Of Integer) = New HashSet(Of Integer)()

For i As Integer = 1 To 4

' Populate evenNumbers with only even numbers.

evenNumbers.Add(i \* 2)

' Populate oddNumbers with only odd numbers.

oddNumbers.Add((i \* 2) - 1)

Next i

Debug.WriteLine("evenNumbers contains elements: " & evenNumbers.Count)

DisplaySet(evenNumbers)

Debug.WriteLine("oddNumbers contains elements: " & oddNumbers.Count)

DisplaySet(oddNumbers)

' Create a new HashSet populated with even numbers.

Dim numbers As HashSet(Of Integer) = New HashSet(Of Integer)(evenNumbers)

Debug.WriteLine("numbers UnionWith oddNumbers...")

numbers.UnionWith(oddNumbers)

Debug.WriteLine("numbers contains elements: " & numbers.Count)

DisplaySet(numbers)

Debug.WriteLine("number min : " & numbers.Min)

Debug.WriteLine("number max : " & numbers.Max)

Debug.WriteLine("number average : " & numbers.Average)

End Sub

Sub DisplaySet(ByVal coll As HashSet(Of Integer))

Debug.Write("{")

For Each i As Integer In coll

Debug.Write(i & " ")

Next i

Debug.WriteLine(" }")

End Sub

Here’s the program’s output:

evenNumbers contains elements: 4

{2 4 6 8 }

oddNumbers contains elements: 4

{1 3 5 7 }

numbers UnionWith oddNumbers...

numbers contains elements: 8

{2 4 6 8 1 3 5 7 }

number min : 1

number max : 8

number average : 4.5

For the relevant properties and methods, take a look at the Array section in this chapter or consult the MSDN help.

## SortedSet

A SortedSet is to a HashSet as a SortedList was to a List – the items are kept in sorted order as they are inserted. Here’s a rework of the last program:

Public Sub Main()

Dim evenNumbers As SortedSet(Of Integer) = New SortedSet(Of Integer)()

Dim oddNumbers As SortedSet(Of Integer) = New SortedSet(Of Integer)()

For i As Integer = 1 To 4

' Populate evenNumbers with only even numbers.

evenNumbers.Add(i \* 2)

' Populate oddNumbers with only odd numbers.

oddNumbers.Add((i \* 2) - 1)

Next i

Debug.WriteLine("evenNumbers contains elements: " & evenNumbers.Count)

DisplaySet(evenNumbers)

Debug.WriteLine("oddNumbers contains elements: " & oddNumbers.Count)

DisplaySet(oddNumbers)

' Create a new HashSet populated with even numbers.

Dim numbers As SortedSet(Of Integer) = New SortedSet(Of

Integer)(evenNumbers)

Debug.WriteLine("numbers UnionWith oddNumbers...")

numbers.UnionWith(oddNumbers)

Debug.WriteLine("numbers contains elements: " & numbers.Count)

DisplaySet(numbers)

Debug.WriteLine("number min : " & numbers.Min)

Debug.WriteLine("number max : " & numbers.Max)

Debug.WriteLine("number average : " & numbers.Average)

End Sub

Sub DisplaySet(ByVal coll As SortedSet(Of Integer))

Debug.Write("{")

For Each i As Integer In coll

Debug.Write(i & " ")

Next i

Debug.WriteLine(" }")

End Sub

The output – the real difference is on the numbers set where we combined the two:

evenNumbers contains elements: 4

{2 4 6 8 }

oddNumbers contains elements: 4

{1 3 5 7 }

numbers UnionWith oddNumbers...

numbers contains elements: 8

{1 2 3 4 5 6 7 8 }

number min : 1

number max : 8

number average : 4.5

# Hashtables

A Hashtable is a collection of key/value pairs that are set up to be based on the hash code of the key. Think of it as a dictionary but with a backing unique hash for each key/value pair.

Here’s some sample code:

Sub Main()

Dim Temperatures As New Hashtable

'Load the Hashtable up...

'The first piece is the key, the second is the data

Temperatures.Add("SVSU", 79)

Temperatures.Add("Flint", 85)

Temperatures.Add("Detroit", 76)

Temperatures.Add("Saginaw", 80)

'Look up city name in hashtable to get temperature

Debug.WriteLine("SVSU's temperature is " & Temperatures.Item("SVSU"))

'Delete SVSU from hashtable

Temperatures.Remove("SVSU")

'OK now we can walk over the data in the order we put it in...

Debug.Write("Walking over hashtable by values (in order added): ")

For Each intTemp In Temperatures.Values

Debug.Write(intTemp & " ")

Next

Debug.WriteLine(" ")

'Walk over hashtable by keys

Debug.Write("Walking over hashtable by keys (in order added): ")

For Each strKey In Temperatures.Keys

Debug.Write(strKey & " ")

Next

Debug.WriteLine(" ")

'Walk over hashtable items in key order

Debug.Write("Walking over hashtable items via key order: ")

For Each strKey In Temperatures.Keys

Debug.Write(Temperatures.Item(strKey) & " ")

Next

Debug.WriteLine(" ")

Debug.WriteLine("There are " & Temperatures.Count & " items")

Debug.WriteLine("The Saginaw key is found? " &

Temperatures.ContainsKey("Saginaw"))

Debug.WriteLine("The 80 value is found? " &

Temperatures.ContainsValue(80))

End Sub

Here’s the output:

SVSU's temperature is 79

Walking over hashtable by values (in order added): 80 85 76

Walking over hashtable by keys (in order added): Saginaw Flint Detroit

Walking over hashtable items via key order: 80 85 76

There are 3 items

The Saginaw key is found? True

The 80 value is found? True

Like the Dictionary, iterators walk over the Hashtable in the order that items were added to it. We can access the items via Key or Value containers.

Hashtable Class Methods:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Add | Adds items to the hashtable |
| Clear | Empties the hashtable |
| Clone | Returns a shallow copy of the hashtable |
| Contains | Returns whether or not the hashtable contains an item with a particular key |
| ContainsKey | Returns whether or not the hashtable contains a particular key |
| ContainsValue | Returns whether or not the hashtable contains a particular value |
| Count | Returns the number of items in the hashtable |
| GetHash | Returns the hash code for the specified key |
| Item | Returns an item by key (if the key doesn’t exist you will get an exception) |
| KeyEquals | Compares a specific object with a specific key in the hashtable |
| Keys | An enumerable set of all keys in the hashtable (in order added) |
| Remove | Removes the element with the specified key from the hashtable |
| Values | An enumerable set of all values in the dictionary (in order added) |

# ArrayLists

ArrayLists are a very handy data structure. They give you all of the functionality of an array (so built-in sorting/searching, element indexing numbers) but with automatic resizing power of a list. Sound cool? It is!

Here are the properties and methods that we are primarily interested on with an ArrayList:

|  |  |
| --- | --- |
| *Method* | *Purpose* |
| Add | Adds an object to the end of the arraylist |
| BinarySearch | Searches a list for a value (req's the arraylist to be sorted) |
| Capacity | Get or set the number of elements that the arraylist can contain |
| Clear | Removes all elements from the arraylist |
| Contains | Determines whether an element is in the arraylist |
| CopyTo | Copies the entire list to a target array |
| Count | Returns the number of elements on the ArrayList |
| Equals | Returns whether the specified arraylist is equal to the current arraylist |
| Exists | Determines whether the specified list contains elements that match some specified condition |
| FixedSize | Returns an arraylist wrapper with a fixed size |
| IndexOf | Returns the index at the first occurrence of the value searched for |
| Item | Gets or sets the element at the specified index |
| Insert | Inserts an element into the arraylist at the specified index |
| LastIndexOf | Returns the index at the last occurrence of the value searched for |
| MemberwiseClone | Creates a shallow copy of the current object |
| ReadOnly | Returns a read-only arraylist wrapper |
| Remove | Removes the first occurrence of a specific element from the arraylist |
| RemoveAt | Removes the element at the specified index of the arraylist |
| Reverse | Reverses the elements in the arraylist |
| Sort | Performs the built-in quicksort sorting operation on the arraylist |
| ToArray | Copies the elements from the arrayList into a new array |
| TrimToSize | Sets the capacity to the number of the actual elements in the arraylist |

Here’s a simple program showing the use of many of the ArrayList methods:

Sub Main()

Dim aList As New ArrayList

aList.Add("Tom")

aList.Add("Bill")

aList.Add("Sue")

aList.Add("Elizabeth")

aList.Add("Robert")

Debug.Write("The original list : ")

For Each strItem In aList

Debug.Write(strItem & " ")

Next

Debug.WriteLine(" ")

Debug.WriteLine("The list has " & aList.Count & " items on it")

Debug.WriteLine("Tom is on the list? : " & aList.Contains("Tom"))

aList.RemoveAt(0) 'Bye Tom

aList.Sort()

Debug.Write("The sorted list : ")

For Each strItem In aList

Debug.Write(strItem & " ")

Next

Debug.WriteLine(" ")

Debug.WriteLine("The list has " & aList.Count & " items on it")

Debug.WriteLine("Tom is on the list? : " & aList.Contains("Tom"))

Debug.WriteLine("Binary Search for Tom found at element : " &

aList.BinarySearch("Tom"))

Debug.WriteLine("Binary Search for Elizabeth found at element : " &

aList.BinarySearch("Elizabeth"))

End Sub

Here’s the output:

The original list : Tom Bill Sue Elizabeth Robert

The list has 5 items on it

Tom is on the list? : True

The sorted list : Bill Elizabeth Robert Sue

The list has 4 items on it

Tom is on the list? : False

Binary Search for Tom found at element : -5

Binary Search for Elizabeth found at element : 1

# Specialized Data Structures

While many of the structures we looked at are pretty general in purpose, Visual Studio has also created several that have specific uses:

|  |  |
| --- | --- |
| BitArray | Provides an easy way to create and manipulate Boolean based array sets. |
| BlockingCollection | Provides blocking and bounding capabilities for thread-safe collections. |
| [CollectionsUtil](http://msdn.microsoft.com/en-us/library/system.collections.specialized.collectionsutil.aspx) | Creates collections that ignore the case in strings. |
| ConcurrentBag | Represents a thread-safe, unordered collection of objects. |
| ConcurrentDictionary | Represents a thread-safe collection of key/value pairs that can be accessed by multiple threads concurrently. |
| ConcurrentQueue | Represents a thread-safe first in-first out (FIFO) collection. |
| ConcurrentStack | Represents a thread-safe last-in first out (LIFO) collection. |
| [HybridDictionary](http://msdn.microsoft.com/en-us/library/system.collections.specialized.hybriddictionary.aspx) | Implements IDictionary by using a [ListDictionary](http://msdn.microsoft.com/en-us/library/system.collections.specialized.listdictionary.aspx) while the collection is small, and then switching to a[Hashtable](http://msdn.microsoft.com/en-us/library/system.collections.hashtable.aspx) when the collection gets large. |
| [ListDictionary](http://msdn.microsoft.com/en-us/library/system.collections.specialized.listdictionary.aspx) | Implements IDictionary using a singly linked list. Recommended for collections that typically include fewer than 10 items. |
| [NameValueCollection](http://msdn.microsoft.com/en-us/library/system.collections.specialized.namevaluecollection.aspx) | Represents a collection of associated [String](http://msdn.microsoft.com/en-us/library/system.string.aspx) keys and [String](http://msdn.microsoft.com/en-us/library/system.string.aspx) values that can be accessed either with the key or with the index. |
| [OrderedDictionary](http://msdn.microsoft.com/en-us/library/system.collections.specialized.ordereddictionary.aspx) | Represents a collection of key/value pairs that are accessible by the key or index. |
| [StringCollection](http://msdn.microsoft.com/en-us/library/system.collections.specialized.stringcollection.aspx) | Represents a collection of strings. |
| [StringDictionary](http://msdn.microsoft.com/en-us/library/system.collections.specialized.stringdictionary.aspx) | Implements a hash table with the key and the value strongly typed to be strings rather than objects. |

You really should spend some time looking over these various data structures and understand when, why and where you should use them as they will save you a considerable amount of effort and code in trying to quickly handle bulky sets of data.