Chapter 7: Introduction to Data Structures

In a previous chapter, we looked at arrays, which are the simplest data structures and have fixed sizes, although they can be redimensioned. .NET offers several other more dynamic data structures known as collection objects, which hold or contain groups of other objects such as, for example, a portfolio of options. These data structures are found in the System.Collections namespace, the most notable of which for right now are arraylists, queues, stacks, hashtables and sortedlists. Additionally, the System.Collections.Generic namespace contains generics, where are type-safe collections. By using generics, the collection will not incur the overhead or risk of runtime casts or boxing operations

The System.Collections namespace contains several classes for collections of objects. These collection classes differ from the Collection class. Notice, however, the inclusion in this namespace of the CollectionBase, which we looked at briefly in the previous chapter. Here is a list of the collection classes in the System.Collections namespace:

System.Collections Namespace Classes	**Description**
ArrayList	An array whose size is dynamic.
CollectionBase	The abstract base class for a collection.
DictionaryBase	The base class for a collection of key-and-value pairs.
Hashtable	A collection of key-and-value pairs organized by hash code.
Queue	A first-in, first-out collection of objects.
SortedList	A collection of key-and-value pairs sorted by key and are accessible by both key and index.
Stack	A last-in, first-out collection of objects.
Structure	**Description**
DictionaryEntry	Defines a dictionary key-and-value pair.

We will not discuss fully each of these classes. However, we will illustrate a hashtable and leave it to the reader to investigate the various members of each of the classes should the need for them arise. For now, be aware that they exist and understand their differing descriptions.

Creating a Customized Collection Class

In this simple example, we will create our own collection class that will hold a portfolio of options. This new collection class will allow us only to add option objects. As in the previous example, any object type, not just CallOptions, can be added to an instance of the generic Collection class since it is not strongly typed. There is an inherent advantage and a disadvantage with using this approach. The advantage is that any object representing a tradable instrument can be added to our m_Portfolio object. However, the disadvantage is that if we try to use a For Each CallOption In m_Portfolio…Next loop to process a portfolio of options, an error will occur since one element in M_Portfolio may be for example a GovtBond object.

In cases where we require a more robust collection, we can, through inheritance from the CollectionBase class, create our own collection class and add our own functionality. The CollectionBase class, found in the System.Collections.namespace, includes the public Clear method and the Count property, and a protected property called List which implements the IList interface. The methods and properties—Add, Remove, and Item--require that we codify the implementation as you will see.

In this example, we will create an OptionCollection that only accepts CallOptions as opposed to any object. Then we will add methods to buy, implementing IList.Add(), and sell, IList.RemoveAt(), CallOptions. Also we will need to implement the Item property that returns the CallOption at a specified index. This customized OptionCollection class will be zero based.

Step 1 Start a new Windows application and name it OptionCollection.

Step 2 In the same way as the previous example, add the code for the StockOption and CallOption classes.

Step 3 Now, add a code module for a third class called OptionCollection with the following code:

VB

```
Public Class OptionCollection
```

```
      Inherits System.Collections.CollectionBase
Public Sub Buy(ByVal m_Option As CallOption)
      List.Add(m_Option)
End Sub
Public Sub Sell(ByVal m_Index As Integer)
      List.RemoveAt(m_Index)
End Sub
Public ReadOnly Property Item(ByVal m_Index As Integer) As CallOption
      Get
            Return List.Item(m_Index)
      End Get
End Property
End Class
```

Notice that the public Buy and Sell methods implement the Add() and RemoveAt() methods and the Item property implements the Item property of the List property of the parent CollectionBase class.

Step 4 In the Form1_Load event, create an instance of the OptionCollection class called M_OptionPortfolio. Also, create two CallOption objects.

```
Dim m_FirstOption As New CallOption( "IBMDP", 1)
Dim m_SecondOption As New CallOption("SUQEX", 1)
```

Step 5 Add the two CallOptions to m_OptionPortfolio by "buying" them.

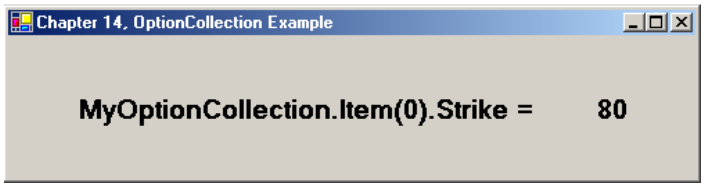
```
m_OptionPortfolio.Buy(m_FirstOption)
m_OptionPortfolio.Buy(m_SecondOption)
```

Step 6 Sell the IBMDP option.

```
m_OptionPortfolio.Sell(0)
```

Step 7 The SUQEX option is left in the portfolio.

```
Label1.Text = m_OptionPortfolio.Item(0).Strike
```



Hashtable

A hashtable is a collection of key and value pairs based upon the hash code of the element's key. Each element is then stored in a DictionaryEntry object. Because of the way they are constructed, hashtables allow for speedy retrieval of elements in the hashtable. When an application needs to store elements, it creates a scheme to convert the element's key value to a subscript, which then becomes the location of that object in the collection. To retrieve the object then, the program converts the key value using the same scheme to find and return the object from its location. This process is called hashing.

When we convert a key to an index value, we are scrambling the bits. Problems can arise, however, when two different keys hash into the same element in an array. Since we certainly cannot store two different records in the same location, we need to find an alternative location via some method. The VB.NET hash table class solves this problem by having each cell of the hash table be a bucket, which is a collection of all the key value pairs that hash to that cell. This entire process is invisible to us, since the hash tables hashing function

calculates where to put the value in the hash table. This function is applied to the key of the key/value pair of objects. Using this process, any object can be added to a hash table.

The .NET Hashtable class implements the IDictionary, ICollection, IEnumerable, ISerialization, IDeserializationCallback and ICloneable interfaces. As a result, there are several member variables, properties, and methods associated with Hashtable objects. Some of the more important members to be aware of are:

Public Constructor	**Description**
Constructor	Initializes a hashtable.
Public Properties	**Description**
Count	Returns the number of elements in the hashtable.
Item	Returns or sets the value of an element in the hashtable.
Keys	Returns a collection containing the keys in the hashtable.
Values	Returns a collection containing the values in the hashtable.
Public Methods	**Description**
Add	Adds an element to the hashtable.
Clear	Deletes all elements from the hashtable.
ContainsKey	Determines whether the hashtable contains a specific key.
ContainsValue	Determines whether the hashtable contains a specific value.
CopyTo	Copies the elements of the hashtables to a one-dimensional array.
Equals	Determines whether two objects are equal.
GetEnumerator	Returns an IDictionaryEnumerator that can iterate through the hashtable.
Remove	Deletes a single element from the hashtable.
Protected Methods	**Description**
GetHash	Returns the hash code for a specified key.
KeyEquals	Compares an object with a specific key in the hashtable.

Since the elements of a Hashtable may be of different types, we can loop through the elements in a Hashtable using an IDictionaryEnumerator. Here is an example from the program presented later in the chapter:

VB

```
Dim enum As IDictionaryEnumerator = m_Hashtable.GetEnumerator()

While enum.MoveNext()
      TextBox1.Text += enum.Value.ToString & vbCrLf
End While
```

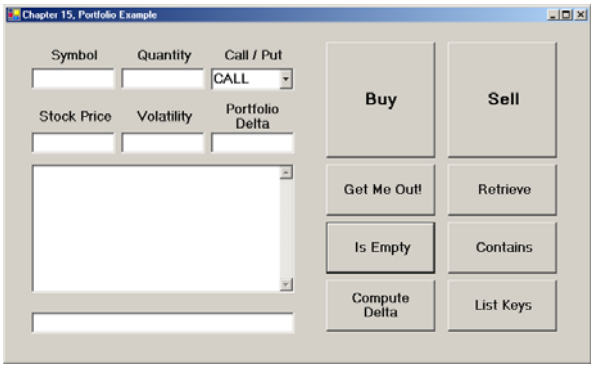
An IDictionaryEnumerator itemizes the elements of a DictionaryEntry object. When an enumerator is created, its position is before the first element in the dictionary. As a result, we must call MoveNext() method in order to advance to the first element. We can then use the Current() property or the Value() property to retrieve the element at which the enumerator is positioned. And then, we can call MoveNext() and iterate through all the elements in the Hashtable. If the enumerator runs off the end of the hashtable, the MoveNext() method will simply return a false value. So, we can loop while MoveNext() is true as in the example shown above. An enumerator will be invalidated if changes are made to the hashtable while it is being used. Here are the key properties and methods of an IDictionaryEnumerator.

Public Properties	**Description**
Current	Retrieves the current element in the dictionary.
Entry	Returns both the key and the value of the current dictionary entry.
Key	Returns the key of the current dictionary entry.
Value	Retrieves the current element in the dictionary.
Public Methods	**Description**
MoveNext	Moves the enumerator to the next element in the dictionary.
Reset	Moves the enumerator to the position before the first element.

Now, let's use a Hashtable to create a robust portfolio object with a great deal more functionality than the one using the Collection class that we looked at in the previous chapter.

Step 1 Open a new Windows application named Portfolio.

Step 2 Create the GUI shown.



Step 3 On your GUI, there should be seven textboxes. In the Properties windows, rename these controls txtSymbol, txtQuantity, txtStockPrice, txtVolatility, txtDelta, txtPortfolio and txtPortStatus. The large textbox in the middle, txtPortfolio, should have the multiline property set to True and the scroll bar property set to vertical. Also on your GUI, there should be a combo box. Rename this combo box cboCallPut. There should be eight buttons on your form. Rename these controls cmdBuy, cmdSell, cmdGetMeOut, cmdContains, cmdIsEmpty, cmdRetrieve, cmdComputeDelta, and cmdListKeys respectively.

Step 4 Now, to add some code. Add a reference to Options.dll and Import Options as well as System.Collections at the top of your Form1 code window.

```
Imports System.Collections
Imports Options
```

Step 5 In the general declarations section of the Form1 code window, create a new Hashtable object named m_Portfolio.

```
Dim m_Portfolio As New Hashtable()
```

Step 6 Add the following code to the cmdBuy_Click event subroutine.

VB

```
Private Sub cmdBuy_Click(ByVal sender As …) Handles cmdBuy.Click
      Dim m_Quantity As Integer = txtQuantity.Text
      Dim m_Symbol As String = txtSymbol.Text
      If cboCallPut.Text = "CALL" Then
            If m_Portfolio.ContainsKey(m_Symbol) = False Then
                Dim m_Option As New CallOption(txtSymbol.Text, _
                              m_Quantity)
                m_Portfolio.Add(m_Symbol, m_Option)
            Else
                m_Portfolio(strSymbol).Quantity += m_Quantity
                If m_Portfolio(m_Symbol).Quantity = 0 Then _
                              m_Portfolio.Remove(m_Symbol)
            End If
      Else
            If m_Portfolio.ContainsKey(m_Symbol) = False Then
                Dim m_Option As New PutOption(txtSymbol.Text, _
                             M_OptionQuantity)
                m_Portfolio.Add(m_Symbol, m_Option)
            Else
                m_Portfolio(m_Symbol).Quantity() += m_OptionQuantity
                If m_Portfolio(m_Symbol).Quantity = 0 Then _
```

```
                        m_Portfolio.Remove(m_Symbol)
            End If
      End If
      ListPortfolioElements()
End Sub
```

There are several things going on in this routine. First of all, the symbol and quantity are read into variables. Second, the code distinguishes between call and put. If Call is selected in the combo box, a CallOption object is added to m_Portfolio. Likewise, if Put is selected, a PutOption object is added. Before either one is added, however, the program checks to see if that particular option already exists in m_Portfolio using the m_Portfolio.ContainsKey(strSymbol) member function. If m_Portfolio already contains a position in that option, it simply increments the quantity of the current position. If there is no current position in that option in m_Portfolio, then it creates the new CallOption or PutOption object and adds it to m_Portfolio. Third and last, the procedure calls the ListPortfolioElements() subroutine, which we will look at shortly.

Step 7 Add the following code for the cmdSell_Click event subroutine. The cmdSell_Click event routine is the same as the cmdBuy routine, but subtracts the quantity rather than adds it.

VB

```
Private Sub cmdSell_Click(ByVal sender As …) Handles cmdSell.Click
      Dim m_Quantity As Integer = txtQuantity.Text
      Dim m_Symbol As String = txtSymbol.Text
      If cboCallPut.Text = "CALL" Then
            If m_Portfolio.ContainsKey(m_Symbol) = False Then
                Dim m_Option As New CallOption(txtSymbol.Text, _
                                -m_Quantity)
                m_Portfolio.Add(m_Symbol, m_Option)
            Else
                m_Portfolio(m_Symbol).Quantity -= m_Quantity
                If m_Portfolio(m_Symbol).Quantity = 0 Then _
                        m_Portfolio.Remove(m_Symbol)
            End If
      Else
            If m_Portfolio.ContainsKey(m_Symbol) = False Then
                Dim m_Option As New PutOption(txtSymbol.Text, _
                                - m_Quantity)
                m_Portfolio.Add(strSymbol, m_Option)
            Else
                m_Portfolio(m_Symbol).Quantity() -= m_Quantity
                If m_Portfolio(m_Symbol).Quantity = 0 Then _
                                m_Portfolio.Remove(m_Symbol)
            End If
      End If
      ListPortfolioElements()
End Sub
```

Step 8 Add the following code to the cmdRetrieve_Click event.

VB

```
Private Sub cmdRetrieve_Click(ByVal sender As …) Handles cmdRetrieve.Click
        Dim m_Symbol As String = txtSymbol.Text
        Dim resultOption As Object = m_Portfolio(m_Symbol)
        If Not resultOption Is Nothing Then
            txtPortStatus.Text = "Retrieved: " & resultOption.ToString()
        Else
            txtPortStatus.Text = txtSymbol.Text & " not in the Portfolio."
        End If
        ListPortfolioElements()
```

```
End Sub
```

The cmdRetrieve_Click event finds the specific element within m_Portfolio, if it exists. In this example, we are just printing out in a text box the fact that it was found. In more sophisticated production program and systems, we would probably want to do something more important.

Step 9 Add the following code to the cmdIsEmpty_Click event.

VB

```
Private Sub cmdIsEmpty_Click(ByVal sender As …) Handles cmdIsEmpty.Click
        If m_Portfolio.Count = 0 Then
            txtPortStatus.Text = "Portfolio is empty."
        Else
            txtPortStatus.Text = "Portfolio is not empty."
        End If
        ListPortfolioElements()
End Sub
```

This event simply uses the m_Portfolio.Count method as you can see. The simplicity of using System.Collections classes is what makes them so powerful.

Step 10 Add the following code to the cmdContains_Click event.

VB

```
Private Sub cmdContains_Click(ByVal sender As …) Handles cmdContains.Click
        Dim m_Symbol = txtSymbol.Text
        txtPortStatus.Text = "Contains: " & m_Portfolio.ContainsKey(m_Symbol)
End Sub
```

This subroutine simply calls the ContainsKey() method of m_Portfolio to check and see whether a specific element is present in the library. The ContainsKey() method returns a Boolean.

Step 11 Add the following code to the cmdGetMeOut_Click event.

VB

```
Private Sub cmdGetMeOut_Click(ByVal sender As …) Handles cmdGetMeOut.Click
        m_Portfolio.Clear()
        txtPortStatus.Text = "You are out. Portfolio is now empty."
        ListPortfolioElements()
End Sub
```

The cmdGetMeOut_Click event calls the Clear() method of the Hashtable object m_Portfolio which removes all the elements from the library.

Step 12 Add the following code to the cmdListKeys_Click event.

VB

```
Private Sub cmdListKeys_Click(ByVal sender As …) Handles cmdListKeys.Click
       Dim enumerator As IDictionaryEnumerator = m_Portfolio.GetEnumerator()
       txtPortfolio.Text = "PORTFOLIO KEYS:" & vbCrLf
       While enumerator.MoveNext()
           txtPortfolio.Text += enumerator.Key & vbCrLf
       End While
End Sub
```

Here we see the IDictionaryEnumerator at work as we discussed in the example.

Step 13 Add the following code for the ListPortfolioElements() subroutine.

VB

```
Private Sub ListPortfolioElements()
        Dim enumerator As IDictionaryEnumerator = m_Portfolio.GetEnumerator()
        txtPortfolio.Text = "PORTFOLIO ELEMENTS:" & vbCrLf
        While enumerator.MoveNext()
```

```
            txtPortfolio.Text += enumerator.Value.ToString & vbCrLf
        End While
End Sub
```

Here again we see the IDictionaryEnumerator at work calling the ToString() method of each successive enumerator.Value.

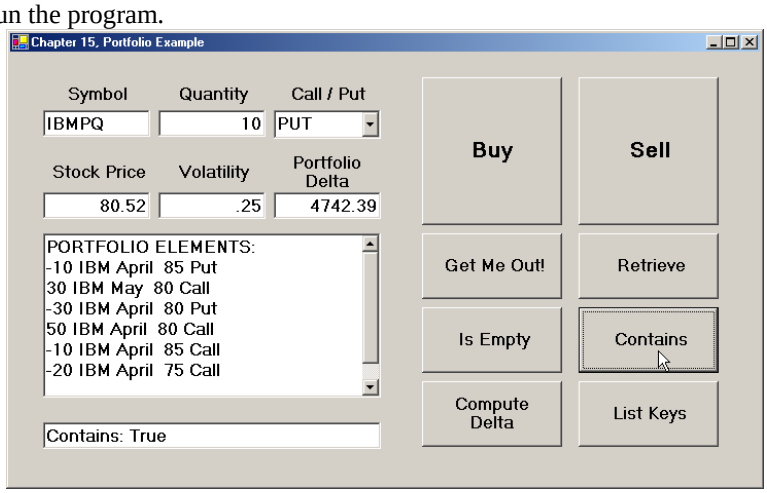
Step 14 Add the following code for the ComputeDelta_Click event.

VB

```
Private Sub ComputeDelta_Click(ByVal sender As …) Handles ComputeDelta.Click
        Dim enumerator As IDictionaryEnumerator = m_Portfolio.GetEnumerator()
        Dim m_Delta As Double = 0
        While enumerator.MoveNext()
            enumerator.Value.StockPrice() = Val(txtStockPrice.Text)
            enumerator.Value.Volatility() = Val(txtVolatility.Text)
            m_Delta += (enumerator.Value.Quantity * enumerator.Value.Delta)
        End While
        txtDelta.Text = Format(m_Delta * 100, "#.00")
End Sub
```

The portfolio delta calculation takes the individual option deltas times the number of contracts times 100 shares per contract to arrive at a portfolio delta.

Step 15 Run the program.



Generics

We can also create generic collections using the classes in the System::Collections::Generic namespace, including Dictionary, List and LinkedList. Generic collections allow us to create strongly typed collections that provide better type safety and performance than non-generic collections. Here is an example showing the use of the LinkedList generic class.

VB

```
Public Class Element

    Private m_Value As Integer

    Public Sub New(ByVal v As Integer)
        m_Value = v
    End Sub

    Public ReadOnly Property value()
```

```
        Get
            Return m_Value
        End Get
    End Property

End Class
```

VB

```
Imports System.Collections.Generic

Module Module1

    Sub Main()

        Dim m_List As LinkedList(Of Element) = New LinkedList(Of Element)
        Dim x As Integer
        For x = 0 To 10
            m_List.AddFirst(New Element(x))
        Next

        m_List.AddLast(New Element(25))
        m_List.RemoveFirst()

        Dim e As Element
        For Each e In m_List
            Console.WriteLine(e.value)
        Next

    End Sub

End Module
```

C#

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

namespace ConsoleApplication2
{
    class Element
    {
        private int m_Value;

        public Element( int v )
        {
            m_Value = v;
        }
        public int Value
        {
            get
            {
                return m_Value;
            }
        }
    }
}
```

C#

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

namespace ConsoleApplication2
{
   class Program
   {
        static void Main(string[] args)
        {
            LinkedList < Element > m_List = new LinkedList < Element >();
            for ( int x = 0; x < 10; x++ )
            {
                m_List.AddFirst( new Element( x ) );
            }
            m_List.AddLast( new Element( 25 ) );
            m_List.RemoveFirst();

            foreach (Element e in m_List )
            {
                Console.WriteLine( e.Value );
            }
        }
    }
}
```

Classes in the System::Collections namespace use the base Object type to store elements. Usually, we have to explicitly cast a pointer to an Object as a pointer to some other type when accessing it as a collection member. Strongly typed generic collections solve this problem.

There are subtle differences between generics and traditional C++ templates. For example, templates are instantiated at compile time, while generics are compiled at run time. However, both are type safe.

In this chapter we have examined the System.Collections and System.Collections.Generic namespaces. Because we use multiple instances of the same class so often, it is important to learn to manage groups of objects using collections.

VB

```
Imports System.Collections.Generic

Module Module1

    Sub Main()
       Dim m_Table as New Dictionary( Of Integer, Element )
       For Each e As KeyValuePair(Of Integer, Element) In m_Table
             Console.WriteLine("Key = {0}, Value = {1}", e.Key, e.Value)
       Next e
    End Sub

End Module
```

C#

```
using System.Collections.Generic;

static void Main(string[] args)
{
      Dictionary< int, Element> m_Table = new Dictionary<int, Element>();
      for ( int x = 0; x < 10; x++ )
      {
           m_Table.Add( x, new Element( x ) );
```

```
      }

      foreach (KeyValuePair<int, Element> e in m_Table)
      {
        Console.WriteLine("Key = {0}, Value = {1}", e.Key, e.Value.Value);
      }
}
```

Summary

In this brief chapter, we have illustrated the use of a Hashtable object. Several classes, including Hashtables, are defined in the System.Collections namespace. As you have seen, implementing collection objects greatly reduces the complexity of dealing with multiple objects of similar or even different types. In later chapter we will create .NET applications that simulate placing buy and sell orders on real derivatives markets. As trades are "executed," you should think about how you can manage your portfolio of positions as a collection of objects. This will make the jump to calculating portfolio hedge ratios rather simple.

Chapter Problems

1.) What is a Hashtable?
2.) Why are Generics?
3.) What is a LinkedList?
4.) What is a Dictionary?
5.) If our portfolio consisted of options on several different stocks, how could we keep track of the respective deltas?

Project 1

Turn the chapter problem on the portfolio of options from a hashtable to a Generic.Dictionary with for…each loops instead of using enumerators.

Project 2

Create a LinkedList of tick objects. Use a timer (see System.Windows.Forms.Timer class in the help files) to add 1000 ticks to the list at 1 second intervals with time, price and volumes. After 30 ticks, calculate the 30 tick moving averages and print them to the screen.