

Collections in Visual Basic

Well, they're everywhere. Forms, Controls, Printers... In fact, it's a safe bet if an object name ends with an s and it has a Count property it is a Collection. And what is even better, we are not limited to just the built-in Collections, we can make our own.

What is a Collection really? It is an Object. It usually holds other objects. Most of the time, these children are of the same class, but sometimes not (like the Controls collection.) It looks quite a bit like an array. In fact, when we use a control array, we are really using a specialized Collection. But it does things no ordinary array would even think of.

Look at this statement:

```
Dim objMyArray(9) As Object
```

What we have is an array that can hold 10 objects, numbered 0 to 9. Even if we only need 6, array space is already allocated for 10. If we might need 15 later, we are forced to using ReDim Preserve and the supporting code and logic to do it. If we know the "name" of a particular object in our array but not its index, we must iterate or loop through them all, extracting the name of each and comparing to the name we are looking for. All in all, this is not a very easy way of grouping and working with a set of objects.

Now look at this statement:

```
Dim colMyCollection as New Collection
```

We now have an object that can hold 0 to {available memory} objects. We only use the space we need, it dynamically grows and shrinks as we add and remove objects. We can reference a particular "member" by specifying an identifying string key.

Wow, if these collections can do all that, why do we even need arrays? Well, sized correctly, arrays generally are more efficient. They have a fixed numeric position for every element. Collections, on the other hand have a moving numeric position. If you add a member object to index 1, then add another one to index 1, the first member gets pushed down to 2. So arrays have their place and so do Collections. (I need to point out that, while many of the built-in collections are zero-based, the Collection class is one-based.)

The standard Collection class object has one property and three methods (four counting one hidden method.) The Count property always knows exactly how many members are contained. The Add method lets us add a new member, specifying a string key and a relative index position. The Remove method lets us remove and destroy a member, either by specifying its string key, or its current index position. The Item method lets us access a member, once again by key or index. There is one problem with Collections: If I want to know if a particular key is already in the collection I have to use the item method and trap for a run-time error. That's right, if it doesn't exist it raises an error! Not very friendly!

"But", you say, "I want more!" Ok, then add more. Build your own collection object. How?

Here's how.

Open Visual Basic. Select a new ActiveX DLL project. Rename the project. For this example, I will use MyObjectCollection. VB already added one class module, we will add a second one. Right click on the project in the Project Explorer window and choose Add – Class Module. Select New Class

module and OK. Rename the classes. The first class module will be our Collection class. For this example I will use MyObjects. Remember, it is wise to make the name of your collection class plural. The second class will be the member class. I will name it MyObject.

One more thing we should do before coding. We need to set the Instancing property for our two classes. Select MyObjects and in the properties window select Instancing. It will already be on 5 – MultiUse. For my example this will be ok. In some cases you might want to select GlobalMultiUse. Now select MyObject and in the properties window select Instancing. Here we want to make this 2 – PublicNotCreatable. What does this mean? Public means that if my program has an instance of this class, I can use all of its public members and properties. Not creatable means my program can't create an instance of this class. So, I could use it but I can't. Or can I?

Even though my program can't create an instance of MyObject, MyObjects can! So, when we implement the Add method for MyObjects, instead of requiring the client program to supply the object, MyObjects will create the object, add it to the collection and return a reference to the newly created MyObject instance. If you have worked with the TreeView or ListView controls, you will recognize this behavior in the Nodes Collection.

Now might be a good time to save our project. Lets start coding. Open the codepane for MyObjects. First, we need to declare a class module variable.

```
Dim mcolMyObjects As Collection
```

Now we add the Class_Initialize and Class_Terminate events.

```
Private Sub Class_Initialize()  
    Set mcolMyObjects = New Collection  
End Sub
```

```
Private Sub Class_Terminate()  
    Set mcolMyObjects = Nothing  
End Sub
```

Add our one standard property, Count (you can add your own special properties later.) You will notice we only have a Property Get Procedure. This makes this a read-only property. To have a property that can be changed by the client program also requires a Property Let and/or Property Set Procedure.

```
Public Property Get Count()  
    'Just pass on our internal collection's count  
    Count = mcolMyObjects.Count  
End Property
```

Now let's move on to the methods. Before we get into our normal, public methods let me explain the GetEnumerator hidden method. Here is the code.

```
Public Property Get GetEnumerator() As IUnknown  
    ' This property allows you to enumerate  
    ' this collection with the For...Each syntax  
    Set GetEnumerator = mcolMyObjects.[_GetEnumerator]  
End Property
```

As the comment says, this allows all the members of this Collection to be extracted via a For Each loop. Keys and Index numbers won't be needed or even known. Explaining IUnknown is beyond the scope of this tutorial. Methods that begin with underscores are hidden methods. Visual Basic doesn't allow methods to begin with underscores. So how are we going to make this a hidden method? Place your cursor somewhere within the procedure and select Procedure Attributes from the Tools menu. If the Name combobox doesn't say NewEnum then pull down and select it. Now, click the Advanced >> button. You will need to click the checkbox Hide this member. One more thing: In the Procedure ID: combobox you will need to enter -4. Don't bother looking for it in the pull-down list; it isn't there. This tells client programs that create instances of your Collection object that this is the procedure to call when doing a For Each. Once you make these changes, click OK.

The first public method we will do is the Add method. Remember what I already said about the behavior of our Add. We will pass through the built-in functionality of our internal Collection object. I will also add some simple error checking and handling on the client program's requested key.

```
Public Function Add(ByVal vKey As String, _
                  Optional ByVal vBefore As Long = 0, _
                  Optional ByVal vAfter As Long = 0) As MyObject
    Dim objNewMyObject As MyObject
```

```
On Error GoTo ErrorHandler:
```

```
    If Len(vKey) > 0 And Len(vKey) < 51 Then
        Set objNewMyObject = New MyObject
        If vBefore > 0 Then 'Provide same functionality as Collection
            mcolMyObjects.Add MyObject, vKey, vBefore
        ElseIf vAfter > 0 Then
            mcolMyObjects.Add MyObject, vKey, , vAfter
        Else
            mcolMyObjects.Add MyObject, vKey
        End If
        Set Add = objNewMyObject 'Pass back reference
        Set objNewMyObject = Nothing 'Clean up reference
    Else
        Set Add = Nothing 'Signal failure by returning nothing object
    End If
    Exit Function
```

```
ErrorHandler:
    Set Add = Nothing
    Err.Raise Err.Number, "MyObjectCollection.MyObjects", Err.Description
End Function
```

Now the Remove method. Once again, only basic error handling is included.

```
Public Sub Remove(ByVal vKey As Variant)
    ' Variant allows for string key or long index
On Error GoTo ErrorHandler:

    mcolMyObjects.Remove vKey
    Exit Sub
```

ErrorHandler:

```
Err.Raise Err.Number, "MyObjectCollection.MyObjects", Err.Description
End Sub
```

Finally, here is the Item method. We will do a little better here. Since I hate it when the collection raises an error on a not-found condition, I will make my class just return a Nothing object. And, so I can tell the difference between a not-found and an invalid key, I will raise an error on the invalid key. The nice thing is it is your collection; you decide how you want to handle every case.

```
Public Function Item(ByVal vKey As Variant) As MyObject
```

```
' Variant allows for string key or long index
```

```
On Error GoTo ErrorHandler:
```

```
    If Len(vKey) > 0 And Len(vKey) < 51 Then
```

```
        Set Item = mcolMyObjects.Item(vKey)
```

```
    Else
```

```
        Set Item = Nothing 'signal failure
```

```
        Err.Raise vbObjectError + 10000, _
```

```
            "MyObjectCollection.MyObjects", "Key value invalid"
```

```
    End If
```

```
Exit Function
```

ErrorHandler:

```
    If Err.Number = 5 Or Err.Number = 9 Then
```

```
        Set Item = Nothing 'Item not found
```

```
    Else
```

```
        Err.Raise Err.Number, "MyObjectCollection.MyObjects", Err.Description
```

```
    End If
```

```
End Function
```

Just for kicks we will add one more basic Method, RemoveAll. It probably should have been part of the basic Collection class anyway. I will take the easy way here. One thing to note: If the client program still holds any references to member objects they become orphans.

```
Public Sub RemoveAll()
```

```
    Class_Terminate
```

```
    Class_Initialize
```

```
End Sub
```

This completes the basic functionality of our Collection. We haven't done anything with our member object class yet. Just for example we will add a Property, Method and Event. You wouldn't normally have events with member objects but I want to see how it works. Here is all the code.

```
Private mstrMyName As String
```

```
Private intHowMany As Integer
```

```
Public Event MyEvent(ByVal vHowMany As Integer)
```

```
Private Sub Class_Initialize()
```

```
    mstrMyName = "MyObject"
```

```
    intHowMany = 0
```

```
End Sub
```

```

Public Property Get MyName() As String
    MyName = mstrMyName
End Property

Public Property Let MyName(ByVal vNewValue As String)
    mstrMyName = vNewValue
End Property

Public Function MyMethod() As Integer
    intHowMany = intHowMany + 1
    'Raise event every ten method invocations
    If intHowMany Mod 10 = 0 Then
        RaiseEvent MyEvent(intHowMany)
    End If
    MyMethod = intHowMany
End Function

```

There is one final thing to consider. How do we test and debug our new Collection. First, go to the File menu and select Add Project. Select new Standard EXE. Right click the new project in the Project Explorer window and select Set as Start Up. Now, select References from the Project Menu. You should see the MyObjectCollection class right below the last checked reference. Check it and OK. You can now declare variables of type MyObject and MyObjects.

I have put together a test project to demonstrate the functions of this new class. Try it out. I hope you have learned just how powerful Collections, and especially your own new Collection classes, can be.