Chapter 20: Collections

Section 20.1: Getting the Item Count of a Collection

The number of items in a `Collection` can be obtained by calling its `.Count` function:

Syntax:

```
.Count()
```

Sample Usage:

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
        .Add "Four"
    End With

    Debug.Print foo.Count    'Prints 4
End Sub
```

Section 20.2: Determining if a Key or Item Exists in a Collection

Keys

Unlike a Scripting.Dictionary, a `Collection` does not have a method for determining if a given key exists *or* a way to retrieve keys that are present in the `Collection`. The only method to determine if a key is present is to use the error handler:

```
Public Function KeyExistsInCollection(ByVal key As String, _
                                      ByRef container As Collection) As Boolean
    With Err
        If container Is Nothing Then .Raise 91
        On Error Resume Next
        Dim temp As Variant
        temp = container.Item(key)
        On Error GoTo 0

        If .Number = 0 Then
            KeyExistsInCollection = True
        ElseIf .Number <> 5 Then
            .Raise .Number
        End If
    End With
End Function
```

Items

The only way to determine if an item is contained in a `Collection` is to iterate over the `Collection` until the item is located. Note that because a `Collection` can contain either primitives or objects, some extra handling is needed to avoid run-time errors during the comparisons:

```
Public Function ItemExistsInCollection(ByRef target As Variant, _
                                       ByRef container As Collection) As Boolean
    Dim candidate As Variant
    Dim found As Boolean

    For Each candidate In container
        Select Case True
            Case IsObject(candidate) And IsObject(target)
                found = candidate Is target
            Case IsObject(candidate), IsObject(target)
                found = False
            Case Else
                found = (candidate = target)
        End Select
        If found Then
            ItemExistsInCollection = True
            Exit Function
        End If
    Next
End Function
```

Section 20.3: Adding Items to a Collection

Items are added to a `Collection` by calling its `.Add` method:

Syntax:

```
.Add(item, [key], [before, after])
```

Parameter	Description
item	The item to store in the `Collection`. This can be essentially any value that a variable can be assigned to, including primitive types, arrays, objects, and `Nothing`.
key	Optional. A `String` that serves as a unique identifier for retrieving items from the `Collection`. If the specified key already exists in the `Collection`, it will result in a Run-time error 457: "This key is already associated with an element of this collection".
before	Optional. An existing key (`String` value) *or* index (numeric value) to insert the item before in the `Collection`. If a value is given, the *after* parameter ***must*** be empty or a Run-time error 5: "Invalid procedure call or argument" will result. If a `String` key is passed that does not exist in the `Collection`, a Run-time error 5: "Invalid procedure call or argument" will result. If a numeric index is passed that is does not exist in the `Collection`, a Run-time error 9: "Subscript out of range" will result.
after	Optional. An existing key (`String` value) *or* index (numeric value) to insert the item after in the `Collection`. If a value is given, the *before* parameter ***must*** be empty. Errors raised are identical to the *before* parameter.

Notes:

- Keys are ***not*** case-sensitive. `.Add "Bar", "Foo"` and `.Add "Baz", "foo"` will result in a key collision.
- If neither of the optional *before* or *after* parameters are given, the item will be added after the last item in the `Collection`.
- Insertions made by specifying a *before* or *after* parameter will alter the numeric indexes of existing members to match thier new position. This means that care should be taken when making insertions in loops using numeric indexes.

Sample Usage:

```
Public Sub Example()
```

```
    Dim foo As New Collection

    With foo
        .Add "One"             'No key. This item can only be retrieved by index.
        .Add "Two", "Second"   'Key given. Can be retrieved by key or index.
        .Add "Three", , 1      'Inserted at the start of the collection.
        .Add "Four", , , 1     'Inserted at index 2.
    End With

    Dim member As Variant
    For Each member In foo
        Debug.Print member     'Prints "Three, Four, One, Two"
    Next
End Sub
```

Section 20.4: Removing Items From a Collection

Items are removed from a `Collection` by calling its `.Remove` method:

Syntax:

```
.Remove(index)
```

Parameter	Description
index	The item to remove from the `Collection`. If the value passed is a numeric type or `Variant` with a numeric sub-type, it will be interpreted as a numeric index. If the value passed is a `String` or `Variant` containing a string, it will be interpreted as the a key. If a String key is passed that does not exist in the `Collection`, a Run-time error 5: "Invalid procedure call or argument" will result. If a numeric index is passed that is does not exist in the `Collection`, a Run-time error 9: "Subscript out of range" will result.

Notes:

- Removing an item from a `Collection` will change the numeric indexes of all the items after it in the `Collection`. `For` loops that use numeric indexes and remove items should run *backwards* (`Step -1`) to prevent subscript exceptions and skipped items.
- Items should generally ***not*** be removed from a `Collection` from inside of a `For Each` loop as it can give unpredictable results.

Sample Usage:

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two", "Second"
        .Add "Three"
        .Add "Four"
    End With

    foo.Remove 1             'Removes the first item.
    foo.Remove "Second"      'Removes the item with key "Second".
    foo.Remove foo.Count     'Removes the last item.

    Dim member As Variant
    For Each member In foo
        Debug.Print member   'Prints "Three"
    Next
End Sub
```

Section 20.5: Retrieving Items From a Collection

Items can be retrieved from a `Collection` by calling the `.Item` function.

Syntax:

```
.Item(index)
```

Parameter	Description
index	The item to retrieve from the `Collection`. If the value passed is a numeric type or `Variant` with a numeric sub-type, it will be interpreted as a numeric index. If the value passed is a `String` or `Variant` containing a string, it will be interpreted as the a key. If a String key is passed that does not exist in the `Collection`, a Run-time error 5: "Invalid procedure call or argument" will result. If a numeric index is passed that is does not exist in the `Collection`, a Run-time error 9: "Subscript out of range" will result.

Notes:

- `.Item` is the default member of `Collection`. This allows flexibility in syntax as demonstrated in the sample usage below.
- Numeric indexes are 1-based.
- Keys are ***not*** case-sensitive. `.Item("Foo")` and `.Item("foo")` refer to the same key.
- The *index* parameter is ***not*** implicitly cast to a number from a `String` or visa-versa. It is entirely possible that `.Item(1)` and `.Item("1")` refer to different items of the `Collection`.

Sample Usage (Indexes):

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
        .Add "Four"
    End With

    Dim index As Long
    For index = 1 To foo.Count
        Debug.Print foo.Item(index) 'Prints One, Two, Three, Four
    Next
End Sub
```

Sample Usage (Keys):

```
Public Sub Example()
    Dim keys() As String
    keys = Split("Foo,Bar,Baz", ",")
    Dim values() As String
    values = Split("One,Two,Three", ",")

    Dim foo As New Collection
    Dim index As Long
    For index = LBound(values) To UBound(values)
        foo.Add values(index), keys(index)
    Next

    Debug.Print foo.Item("Bar") 'Prints "Two"
End Sub
```

Sample Usage (Alternate Syntax):

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One", "Foo"
        .Add "Two", "Bar"
        .Add "Three", "Baz"
    End With

    'All lines below print "Two"
    Debug.Print foo.Item("Bar")     'Explicit call syntax.
    Debug.Print foo("Bar")          'Default member call syntax.
    Debug.Print foo!Bar             'Bang syntax.
End Sub
```

Note that bang (`!`) syntax is allowed because `.Item` is the default member and can take a single `String` argument. The utility of this syntax is questionable.

Section 20.6: Clearing All Items From a Collection

The easiest way to clear all of the items from a `Collection` is to simply replace it with a new `Collection` and let the old one go out of scope:

```
Public Sub Example()
    Dim foo As New Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
    End With

    Debug.Print foo.Count   'Prints 3
    Set foo = New Collection
    Debug.Print foo.Count   'Prints 0
End Sub
```

However, if there are multiple references to the `Collection` held, this method will only give you an empty `Collection` *for the variable that is assigned*.

```
Public Sub Example()
    Dim foo As New Collection
    Dim bar As Collection

    With foo
        .Add "One"
        .Add "Two"
        .Add "Three"
    End With

    Set bar = foo
    Set foo = New Collection

    Debug.Print foo.Count   'Prints 0
    Debug.Print bar.Count   'Prints 3
End Sub
```

In this case, the easiest way to clear the contents is by looping through the number of items in the `Collection` and repeatedly remove the lowest item:

```
Public Sub ClearCollection(ByRef container As Collection)
    Dim index As Long
    For index = 1 To container.Count
        container.Remove 1
    Next
End Sub
```