PREV
Implementing fu

NEXT
Res…

Implementing a generic Iterator

This recipe will describe how to implement a mechanism that is able to execute any operation for objects of any class repeatedly, until a condition is met or until the whole queue of objects is processed. The condition can be any expression that results in a Boolean value.

Getting ready

Create a new function library (for instance, `cls.Iterator.vbs`), and associate it with your test. In this recipe, we shall use the code from previous recipe, *Implementing function pointers*.

How to do it...

Proceed with the following steps:

1. In the function library, write the following code:

```
Class Iterator
    Public Default Function Run(ByRef oCollection, _
                ByRef ptrFunction, _
                ByVal dicArgs, _
                ByVal sExitCondition)

        Dim count, items, ix, str, dicResults

        'Create a Dictionary to store the results for each iteratic
        Set dicResults = CreateObject("Scripting.Dictionary")

        'Get the collection count
        count = oCollection.Count

        'Get the object collection items
        items = oCollection.Items

        'Get the object collection keys
        keys = oCollection.Keys

        ix =  0
        Do While ix < count
            'Check if the exit condition holds true
            If Eval(sExitCondition) Then
                dicResults(keys(ix)) = "Iteration " & ix+1 & " not
                        vbNewLine & "Exit condition '" & sExitCondi
                Exit Do
            End If

            'This statement performs the process/operation on the c
            dicResults(keys(ix)) = ptrFunction(items(ix), dicArgs(k

            'Increment the counter
            ix = ix + 1
        Loop

        'Return Dictionary with the results
        Set Run = dicResults
    End Function
 End Class
```

2. In `Action`, write the following code:

```
Dim dicObjects
Dim dicArgs
Dim pFunc
Dim iter
Dim sExitCond
Dim key, keys

'Set the instances of the required objects
Set iter = New Iterator
Set dicObjects = CreateObject("Scripting.Dictionary")
Set dicArgs = CreateObject("Scripting.Dictionary")
Set pFunc = New WebEditSet

'Assign the string with the end condition
sExitCond = "Err.Number <> 0"
```

```
'Add the Test Objects to Dictionary dicObjects
dicObjects.Add "MyFirstObject", _ Browser("MyBrowser").Page("MyPage
dicObjects.Add "MySecondObject", _ Browser("MyBrowser").Page("MyPag
dicObjects.Add "MyThirdObject", _ Browser("MyBrowser").Page("MyPage

'Add the strings to be passed as arguments to Dictionary dicArgs
dicArgs.Add "MyFirstObject", "One"
dicArgs.Add "MySecondObject", "Two"
dicArgs.Add "MyThirdObject", "Three"

'Call the iterator default function (as a function pointer) with it
Call iter(dicObjects, pFunc, dicArgs, sExitCond)

'Dispose of the objects
Set dicArgs = Nothing
Set dicObjects = Nothing
Set pFunc = Nothing
Set iter = Nothing
```

Note

In our example, we do not store the value returned by the `Run` method (a dictionary) of `Iterator`. However, this can be done by assigning it to a variable using `Set`.

How it works...

The `Iterator` class has a default `Run` method, similar to what we have seen in the previous recipes. This method implements a loop that performs the operation defined by `pFunctiontion` for all objects in the collection defined by `dicObjects`, or until the exit condition defined by `sExitCondition` is reached. For each function call, it passes the corresponding argument, as defined in `dicArguments` (To handle functions accepting multiple arguments, one may simply send an array of values, instead of a single one as done here, and handle them within the target `Run` method of our custom command wrapper.).

The command wrapper `WebEditSet` (refer to the previous recipe) accepts as arguments a reference to the Test Object for which we wish to invoke the `Set` method and a string to be entered to the `WebEdit` object. So, what is actually going on here? The `Iterator` class passes to the `Run` method of `WebEditSet` the arguments it needs to perform the `Set` operation. The `Iterator` class is agnostic of the internals of the called method, except that it expects a return value. In this sense, it is absolutely generic.