Using a global dictionary for recovery

A powerful technique to implement the direct invocation of a recovery procedure makes an ingenious use of a global (public) `Dictionary` object. The method involves the use of command wrappers, function pointers, and callback functions. If you are not familiar with such obje oriented design patterns and concepts, then it is recommended that y first read Chapter 7, *Using Classes*, where some basic concepts are outlined.

The idea of using a dictionary to implement function pointers was firs raised in an article published on www.advancedqtp.com/ (http://www.advancedqtp.com/) back in 2007. A `Dictionary` object is capable storing data of any type, including objects and even other `Dictionar` objects, thus serving as a tree-like structure. It occurred to us that we might exploit this idea to store executable objects, which would run th code upon accessing their associated (hashed) keys. An executable object in VBScript can be built using the command wrapper design pattern, which encapsulates a process (function or subroutine) using a class. Another less elegant, yet effective way of building this feature is by means of storing references to functions using the `GetRef` method. So, in the event of an unexpected error, we may take advantage of such a data structure by means of mapping handling procedures to error numbers. The result is a hash table that stores the instructions of what to do in every case. It is also possible to store by the same method, procedures associated with unexpected events, which are not errors, such as the detection of a pop-up dialog. The advantages of this solution are obvious:

- It reduces the amount of code dedicated to reroute the test flow in case an error or any other unexpected event occurs.
- It provides a highly maintainable and clear way of handling exceptions and flow branching.
- It provides direct access to the event handler, instead of having to recur to ordinary function calls. All you have to do is retrieve the value associated with a key (the error or event), and the code is automatically executed.

The performance of such a mechanism outplays the ordinary recovery scenario procedures, as it is invoked **Just-in-Time** (**JIT**) without putting any burden on the machine's resources.

Getting ready

From the **File** menu, navigate to **New** | **Function Library...**, or use the *Alt* + *Shift* + *N* shortcut. Name the new function library `GlobalDic_Func.vbs`.

How to do it...

Proceed with the following steps:

1. In the function library, we will write the following code to implement the `EventHandlerManager` class, which will be loaded at the start of a run session:

```
Dim oEventHandlerManager

Function createEventHandlerManager()
    'Singleton
    If not lcase(typename(oEventHandlerManager)) = "eventHandlerMar
        Set oEventHandlerManager=new EventHandlerManager
    End If
End Function
```

```
Function disposeEventHandlerManager()
    Set oEventHandlerManager=nothing
End Function

Class EventHandlerManager
    Public m_DicEvents

    Function Run(sEvent)
        Events.item(cstr(sEvent))(sEvent)
    End Function

    Function mapHandlerToEvent(sEvent, sHandler)
        Events.Add sEvent, getHandler(sHandler)
    End Function

    Function getHandler(sHandler)
        Dim oHandler
        On error resume next
        Execute "set oHandler=new " & sHandler
        If err.number <> 0 Then
            Set oHandler=nothing
            reporter.ReportEvent micFail, typename(me) & ".getHandl
        End If

        Set getHandler=oHandler
    End Function

    Property get Events()
        set Events=m_DicEvents
    End Property

    Property let Events(oHandlerManager)
        Set m_DicEvents=oHandlerManager
    End Property

    Sub class_initialize
        Events=createObject("Scripting.Dictionary")
    End Sub
    Sub class_terminate
        Events=nothing
    End Sub
End Class
```

2. Now, add the following code (it can be in the same function library or a separate one) to implement a specific event handler:

```
Class MyHandler
    Public default Function Exec(sEvent)
        reporter.ReportEvent micDone, typename(me), "Handling event
        err.clear
        reporter.ReportEvent micDone, typename(me), "Handled event
    End Function
End Class
```

   The event handler `MyHandler` reports the results and clears the error. For each specific error or event, we will implement such a class and map it using the `oEventHandlerManager` global object.

3. Now, we will see how to map our event handler to an event. As an example, we will use error number `9`, which is division by zero:

```
createEventHandlerManager()

call oEventHandlerManager.mapHandlerToEvent("9", "MyHandler")

On error resume next

err.raise 9

oEventHandlerManager.Run(err.number)

disposeEventHandlerManager()
```

How it works...

We first create a global instance of `EventHandlerManager` in the global variable `oEventHandlerManager`. We then use the `mapHandlerToEvent` method to indicate that we wish to execute the code encapsulated in the `Exec` method of the `MyHandler` class. Then we raise an error, and finally, call the `Run` method of `EventHandlerManager` with the error number as an argument. Please notice that no `if-then-else` structure is used, so it actually can work as a kind of implicit try-catch mechanism. You are invited to try it with different errors and `EventHandler` implementations. For example, such a handler may halt the entire run session, or skip to the next Action iteration or test iteration. At last, we dispose off our `EventHandlerManager` class (this will be done only at the end of the test run).

The `EventHandlerManager` class is actually a wrapper to the dictionary. The member field `m_DicEvents` holds a reference to the dictionary, which stores key-value pairs (in this case, the keys being the error codes or other user-defined events), with the values being references to instances of the corresponding `EventHandler` classes.

As the `Exec` method of `EventHandler` (`MyHandler` in our example) is defined as the default, we do not need to write `Events.item(cstr(sEvent)).Exec(sEvent)` in the `Run` method of `EventHandlerManager`. It is sufficient to access the key by means of `Events.item(cstr(sEvent))(sEvent)`, and this triggers the default

method. In this sense, it works like a function pointer, as mentioned earlier in this chapter.

It is, of course, important to note that in order to make this mechanism work infallibly, all `EventHandler` classes must follow the same basic structure as exemplified in the previous code with `MyHandler`. This means, any such class must contain a public default function named `Exec`.