Recent

Topics

Tutorials

Highlights

Settings

Feedback (http://community.

Sign Out

Settings

10 days left in your trial.
Subscribe.

Feedback
(http://community.safaribooksonline.cor

Sign Out

Catching errors inside a functio or subroutine

In this recipe, we will learn how to implement an error trap inside a function or subroutine.

Getting ready

From the **File** menu, navigate to **New** | **Function Library...** or use t *Alt* + *Shift* + *N* shortcut. Name the new function library `ErrHandling_Func.vbs`. You can use any other name, or reuse an existing function library. Do not forget to ensure that the library is attached to the test through **Resources** (**File** | **Settings** | **Resources**

How to do it...

The technique is very simple. First, within your function, identify the or lines of code that carry the potential of raising an exception (for example, an unhandled error). For instance, we write the following sim function to perform a division operation:

```
Function DblDivideXByY(x, y)
    'Return the result of the division as a Double
    DivideXByY=CDbl(x/y)
End function
```

The problem with the preceding code is that it assumes a priori that the parameters passed to the function are valid. However, there are at least two cases in which the function would fail to execute due to a runtime error:

- `y=0`
- `x` or `y` are not numeric

While it is possible to check for the possible sources of error (in the preceding code, by using the `isnumeric` function and checking if `y` is not equal to zero), it is in general an impractical approach. Unlike syntax errors, which can be found easily (by navigating to **Design** | **Check Syntax** from the UFT home page), runtime errors in VBScript pose a challenging threat to the robustness of our scripts. For example, suppose a function A calls another function B. If the interface of function B changes, say, an additional argument is added; then during runtime, the call would result in an error number of `450` (the wrong number of arguments or invalid property assignments). However, we will not be able to know this until function A is called.

Unlike other programming languages (such as C, C++, C#, and Java), VBScript is an untyped, late-bound language. Untyped means that variables are of a generic type called variant, and they assume a specific type only through assignment. By late-bound, we mean that the correctness of a statement is checked only during runtime. **Windows Script Host** (**WSH**) parses our script code line-by-line, and throws an error only at this stage. For our test script, this would be too late. However, we can use a feature of VBScript that allows us to disable the error checking mechanism during runtime, in order to handle the exceptions in a custom way that better suits our needs. This way, we set a trap to capture the error in a specific block of code, which we expect to be troublesome.

For those knowledgeable in other programming languages, this resembles the `try-catch` structure used to handle potential exceptions:

```
Function DblDivideXByY(x, y)
    const C_FUNC_NAME="DblDivideXByY"
```

```
    'Disable automatic runtime error-handling
    On error resume next
    'Return the result of the division as a Double
    DivideXByY=CDbl(x/y)
    'Check if an error was thrown
    If err.number <> 0 Then
        reporter.ReportEvent micFail, C_FUNC_NAME, err.description
        'TODO: Your handler
        'Example for general handling - Halt the run session
        ExitTest
    End If
    'Enable automatic runtime error-handling
    '(not a must since it's restored upon exiting the function)
    On error goto 0
End Function
```

We then call the `DblDivideXByY` function with the following lines of code:

```
'This is OK
print DblDivideXByY(5, 5)
'This will throw error number 11 (Divide by zero)
print DblDivideXByY(5, 0)
'This will throw error number 13 (Type mismatch)
print DblDivideXByY(5, "wrong")
```

How it works...

The custom method we implemented takes two arguments, `x` and `y`. The first is the dividend and the latter is the divisor. First, to obtain full control over the flow, we disable VBScript's native runtimeerror-handling mechanism with `On error resume next`. Then, we attempt to actually perform the division operation. If an error of any kind occurs, it will be caught by the clause `If err.number <> 0 Then`, and our customexception-handling code will be executed. In our example, we report a failure and stop the test when an error occurs, but the specific implementation one chooses depends on the requirements. For instance, the error may occur under controlled conditions (negative test), and hence, our implementation should be more complex to cover such situations. In any case, it is recommended to leave the function as simple as possible.

See also

Refer to the *Adding a new method to a class* recipe of Chapter 4, *Method Overriding*.

© 2015 Safari.
Terms of Service / Membership Agreement / Privacy Policy