Chapter 46: Error Handling

Section 46.1: Avoiding error conditions

When a runtime error occurs, good code should handle it. The best error handling strategy is to write code that checks for error conditions and simply avoids executing code that results in a runtime error.

One key element in reducing runtime errors, is writing small procedures that *do one thing*. The fewer reasons procedures have to fail, the easier the code as a whole is to debug.

Avoiding runtime error 91 - Object or With block variable not set:

This error will be raised when an object is used before its reference is assigned. One might have a procedure that receives an object parameter:

```
Private Sub DoSomething(ByVal target As Worksheet)
    Debug.Print target.Name
End Sub
```

If `target` isn't assigned a reference, the above code will raise an error that is easily avoided by checking if the object contains an actual object reference:

```
Private Sub DoSomething(ByVal target As Worksheet)
    If target Is Nothing Then Exit Sub
    Debug.Print target.Name
End Sub
```

If `target` isn't assigned a reference, then the unassigned reference is never used, and no error occurs.

This way of early-exiting a procedure when one or more parameter isn't valid, is called a *guard clause*.

Avoiding runtime error 9 - Subscript out of range:

This error is raised when an array is accessed outside of its boundaries.

```
Private Sub DoSomething(ByVal index As Integer)
    Debug.Print ActiveWorkbook.Worksheets(index)
End Sub
```

Given an index greater than the number of worksheets in the `ActiveWorkbook`, the above code will raise a runtime error. A simple guard clause can avoid that:

```
Private Sub DoSomething(ByVal index As Integer)
    If index > ActiveWorkbook.Worksheets.Count Or index <= 0 Then Exit Sub
    Debug.Print ActiveWorkbook.Worksheets(index)
End Sub
```

Most runtime errors can be avoided by carefully verifying the values we're using *before* we use them, and branching on another execution path accordingly using a simple `If` statement - in guard clauses that makes no assumptions and validates a procedure's parameters, or even in the body of larger procedures.

Section 46.2: Custom Errors

Often when writing a specialized class, you'll want it to raise its own specific errors, and you'll want a clean way for

user/calling code to handle these custom errors. A neat way to achieve this is by defining a dedicated `Enum` type:

```
Option Explicit
Public Enum FoobarError
    Err_FooWasNotBarred = vbObjectError + 1024
    Err_BarNotInitialized
    Err_SomethingElseHappened
End Enum
```

Using the `vbObjectError` built-in constant ensures the custom error codes don't overlap with reserved/existing error codes. Only the first enum value needs to be explicitly specified, for the underlying value of each `Enum` member is `1` greater than the previous member, so the underlying value of `Err_BarNotInitialized` is implicitly `vbObjectError + 1025`.

Raising your own runtime errors

A runtime error can be raised using the `Err.Raise` statement, so the custom `Err_FooWasNotBarred` error can be raised as follows:

```
Err.Raise Err_FooWasNotBarred
```

The `Err.Raise` method can also take custom `Description` and `Source` parameters - for this reason it's a good idea to also define constants to hold each custom error's description:

```
Private Const Msg_FooWasNotBarred As String = "The foo was not barred."
Private Const Msg_BarNotInitialized As String = "The bar was not initialized."
```

And then create a dedicated private method to raise each error:

```
Private Sub OnFooWasNotBarredError(ByVal source As String)
    Err.Raise Err_FooWasNotBarred, source, Msg_FooWasNotBarred
End Sub

Private Sub OnBarNotInitializedError(ByVal source As String)
    Err.Raise Err_BarNotInitialized, source, Msg_BarNotInitialized
End Sub
```

The class' implementation can then simply call these specialized procedures to raise the error:

```
Public Sub DoSomething()
    'raises the custom 'BarNotInitialized' error with "DoSomething" as the source:
    If Me.Bar Is Nothing Then OnBarNotInitializedError "DoSomething"
    '...
End Sub
```

The client code can then handle `Err_BarNotInitialized` as it would any other error, inside its own error-handling subroutine.

Note: the legacy `Error` keyword can also be used in place of `Err.Raise`, but it's obsolete/deprecated.

Section 46.3: Resume keyword

An error-handling subroutine will either:

- run to the end of the procedure, in which case execution resumes in the calling procedure.
- or, use the `Resume` keyword to *resume* execution inside the same procedure.

The `Resume` keyword should only ever be used inside an error handling subroutine, because if VBA encounters `Resume` without being in an error state, runtime error 20 "Resume without error" is raised.

There are several ways an error-handling subroutine may use the `Resume` keyword:

- `Resume` used alone, execution continues **on the statement that caused the error**. If the error isn't *actually* handled before doing that, then the same error will be raised again, and execution might enter an infinite loop.
- `Resume Next` continues execution **on the statement immediately following** the statement that caused the error. If the error isn't *actually* handled before doing that, then execution is permitted to continue with potentially invalid data, which may result in logical errors and unexpected behavior.
- `Resume [line label]` continues execution **at the specified line label** (or line number, if you're using legacy-style line numbers). This would typically allow executing some cleanup code before cleanly exiting the procedure, such as ensuring a database connection is closed before returning to the caller.

On Error Resume Next

The `On Error` statement itself can use the `Resume` keyword to instruct the VBA runtime to effectively **ignore all errors**.

> *If the error isn't **actually handled** before doing that, then execution is permitted to continue with potentially invalid data, which may result in **logical errors and unexpected behavior**.*

The emphasis above cannot be emphasized enough. ***On Error Resume Next* effectively ignores all errors and shoves them under the carpet**. A program that blows up with a runtime error given invalid input is a better program than one that keeps running with unknown/unintended data - be it only because the bug is much more easily identifiable. `On Error Resume Next` can easily **hide bugs**.

The `On Error` statement is procedure-scoped - that's why there should *normally* be only **one**, single such `On Error` statement in a given procedure.

However *sometimes* an error condition can't quite be avoided, and jumping to an error-handling subroutine only to `Resume Next` just doesn't feel right. In this specific case, the known-to-possibly-fail statement can be **wrapped** between two `On Error` statements:

```
On Error Resume Next
[possibly-failing statement]
Err.Clear 'resets current error
On Error GoTo 0
```

The `On Error GoTo 0` instruction resets error handling in the current procedure, such that any further instruction causing a runtime error *would be unhandled within that procedure* and instead passed up the call stack until it is caught by an active error handler. If there is no active error handler in the call stack, it will be treated as an unhandled exception.

```
Public Sub Caller()
    On Error GoTo Handler

    Callee

    Exit Sub
Handler:
```

```
    Debug.Print "Error " & Err.Number & " in Caller."
End Sub

Public Sub Callee()
    On Error GoTo Handler

    Err.Raise 1     'This will be handled by the Callee handler.
    On Error GoTo 0 'After this statement, errors are passed up the stack.
    Err.Raise 2     'This will be handled by the Caller handler.

    Exit Sub
Handler:
    Debug.Print "Error " & Err.Number & " in Callee."
    Resume Next
End Sub
```

Section 46.4: On Error statement

Even with *guard clauses*, one cannot realistically *always* account for all possible error conditions that could be raised in the body of a procedure. The `On Error GoTo` statement instructs VBA to jump to a *line label* and enter "error handling mode" whenever an unexpected error occurs at runtime. After handling an error, code can *resume* back into "normal" execution using the `Resume` keyword.

Line labels denote *subroutines*: because subroutines originate from legacy BASIC code and uses `GoTo` and `GoSub` jumps and `Return` statements to jump back to the "main" routine, it's fairly easy to write hard-to-follow *spaghetti code* if things aren't rigorously structured. For this reason, it's best that:

- a procedure has **one and only one** error-handling subroutine
- the error-handling subroutine **only ever runs in an error state**

This means a procedure that handles its errors, should be structured like this:

```
Private Sub DoSomething()
    On Error GoTo CleanFail

    'procedure code here

CleanExit:
    'cleanup code here
    Exit Sub

CleanFail:
    'error-handling code here
    Resume CleanExit
End Sub
```

Error Handling Strategies

Sometimes you want to handle different errors with different actions. In that case you will inspect the global `Err` object, which will contain information about the error that was raised - and act accordingly:

```
CleanExit:
    Exit Sub

CleanFail:
    Select Case Err.Number
        Case 9
            MsgBox "Specified number doesn't exist. Please try again.", vbExclamation
```

```
            Resume
        Case 91
            'woah there, this shouldn't be happening.
            Stop 'execution will break here
            Resume 'hit F8 to jump to the line that raised the error
        Case Else
            MsgBox "An unexpected error has occurred:" & vbNewLine & Err.Description, vbCritical
            Resume CleanExit
    End Select
End Sub
```

As a general guideline, consider turning on the error handling for entire subroutine or function, and handle all the errors that may occur within its scope. If you need to only handle errors in the small section section of the code -- turn error handling on and off a the same level:

```
Private Sub DoSomething(CheckValue as Long)

    If CheckValue = 0 Then
        On Error GoTo ErrorHandler    ' turn error handling on
        ' code that may result in error
        On Error GoTo 0               ' turn error handling off - same level
    End If

CleanExit:
    Exit Sub

ErrorHandler:
    ' error handling code here
    ' do not turn off error handling here
    Resume

End Sub
```

Line numbers

VBA supports legacy-style (e.g. QBASIC) line numbers. The `Erl` hidden property can be used to identify the line number that raised the last error. If you're not using line numbers, `Erl` will only ever return 0.

```
Sub DoSomething()
10 On Error GoTo 50
20 Debug.Print 42 / 0
30 Exit Sub
40
50 Debug.Print "Error raised on line " & Erl ' returns 20
End Sub
```

If you *are* using line numbers, but not consistently, then `Erl` will return *the last line number before the instruction that raised the error*.

```
Sub DoSomething()
10 On Error GoTo 50
   Debug.Print 42 / 0
30 Exit Sub

50 Debug.Print "Error raised on line " & Erl 'returns 10
End Sub
```

Keep in mind that `Erl` also only has `Integer` precision, and will silently overflow. This means that line numbers

outside of the integer range will give incorrect results:

```
Sub DoSomething()
99997 On Error GoTo 99999
99998 Debug.Print 42 / 0
99999
      Debug.Print Erl   'Prints 34462
End Sub
```

The line number isn't quite as relevant as the statement that caused the error, and numbering lines quickly becomes tedious and not quite maintenance-friendly.