Chapter 45: VBA Run-Time Errors

Code that compiles can still run into errors, at run-time. This topic lists the most common ones, their causes, and how to avoid them.

Section 45.1: Run-time error '6': Overflow

Incorrect code

```
Sub DoSomething()
    Dim row As Integer
    For row = 1 To 100000
        'do stuff
    Next
End Sub
```

Why doesn't this work?

The `Integer` data type is a 16-bit signed integer with a maximum value of 32,767; assigning it to anything larger than that will *overflow* the type and raise this error.

Correct code

```
Sub DoSomething()
    Dim row As Long
    For row = 1 To 100000
        'do stuff
    Next
End Sub
```

Why does this work?

By using a `Long` (32-bit) integer instead, we can now make a loop that iterates more than 32,767 times without overflowing the counter variable's type.

Other notes

See Data Types and Limits for more information.

Section 45.2: Run-time error '9': Subscript out of range

Incorrect code

```
Sub DoSomething()
    Dim foo(1 To 10)
    Dim i As Long
    For i = 1 To 100
        foo(i) = i
    Next
End Sub
```

Why doesn't this work?

`foo` is an array that contains 10 items. When the `i` loop counter reaches a value of 11, `foo(i)` is *out of range*. This error occurs whenever an array or collection is accessed with an index that doesn't exist in that array or collection.

Correct code

```
Sub DoSomething()
    Dim foo(1 To 10)
    Dim i As Long
    For i = LBound(foo) To UBound(foo)
        foo(i) = i
    Next
End Sub
```

Why does this work?

Use `LBound` and `UBound` functions to determine the lower and upper boundaries of an array, respectively.

Other notes

When the index is a string, e.g. `ThisWorkbook.Worksheets("I don't exist")`, this error means the supplied name doesn't exist in the queried collection.

The actual error is implementation-specific though; `Collection` will raise run-time error 5 "Invalid procedure call or argument" instead:

```
Sub RaisesRunTimeError5()
    Dim foo As New Collection
    foo.Add "foo", "foo"
    Debug.Print foo("bar")
End Sub
```

Section 45.3: Run-time error '13': Type mismatch

Incorrect code

```
Public Sub DoSomething()
    DoSomethingElse "42?"
End Sub

Private Sub DoSomethingElse(foo As Date)
'    Debug.Print MonthName(Month(foo))
End Sub
```

Why doesn't this work?

VBA is trying really hard to convert the `"42?"` argument into a `Date` value. When it fails, the call to `DoSomethingElse` cannot be executed, because VBA doesn't know what date to pass, so it raises run-time error 13 *type mismatch*, because the type of the argument doesn't match the expected type (and can't be implicitly converted either).

Correct code

```
Public Sub DoSomething()
    DoSomethingElse Now
End Sub

Private Sub DoSomethingElse(foo As Date)
'    Debug.Print MonthName(Month(foo))
End Sub
```

Why does this work?

By passing a `Date` argument to a procedure that expects a `Date` parameter, the call can succeed.

Section 45.4: Run-time error '91': Object variable or With block variable not set

Incorrect code

```
Sub DoSomething()
```

```
    Dim foo As Collection
    With foo
        .Add "ABC"
        .Add "XYZ"
    End With
End Sub
```

Why doesn't this work?

Object variables hold a *reference*, and references need to be *set* using the `Set` keyword. This error occurs whenever a member call is made on an object whose reference is `Nothing`. In this case `foo` is a `Collection` reference, but it's not initialized, so the reference contains `Nothing` - and we can't call `.Add` on `Nothing`.

Correct code

```
Sub DoSomething()
    Dim foo As Collection
    Set foo = New Collection
    With foo
        .Add "ABC"
        .Add "XYZ"
    End With
End Sub
```

Why does this work?

By assigning the object variable a valid reference using the `Set` keyword, the `.Add` calls succeed.

Other notes

Often, a function or property can return an object reference - a common example is Excel's `Range.Find` method, which returns a `Range` object:

```
Dim resultRow As Long
resultRow = SomeSheet.Cells.Find("Something").Row
```

However the function can very well return `Nothing` (if the search term isn't found), so it's likely that the chained `.Row` member call fails.

Before calling object members, verify that the reference is set with a `If Not xxxx Is Nothing` condition:

```
Dim result As Range
Set result = SomeSheet.Cells.Find("Something")

Dim resultRow As Long
If Not result Is Nothing Then resultRow = result.Row
```

Section 45.5: Run-time error '20': Resume without error

Incorrect code

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

CleanFail:
    Debug.Print Err.Number
    Resume Next
End Sub
```

Why doesn't this work?

If the `DoSomethingElse` procedure raises an error, execution jumps to the `CleanFail` line label, prints the error number, and the `Resume Next` instruction jumps back to the instruction that immediately follows the line where the error occurred, which in this case is the `Debug.Print` instruction: the error-handling subroutine is executing without an error context, and when the `Resume Next` instruction is reached, run-time error 20 is raised because there is nowhere to resume to.

Correct Code

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

    Exit Sub
CleanFail:
    Debug.Print Err.Number
    Resume Next
End Sub
```

Why does this work?

By introducing an `Exit Sub` instruction before the `CleanFail` line label, we have segregated the `CleanFail` error-handling subroutine from the rest of the procedure body - the only way to execute the error-handling subroutine is via an `On Error` jump; therefore, no execution path reaches the `Resume` instruction outside of an error context, which avoids run-time error 20.

Other notes

This is very similar to Run-time error '3': Return without GoSub; in both situations, the solution is to ensure that the *normal execution path* cannot enter a sub-routine (identified by a line label) without an explicit jump (assuming `On Error GoTo` is considered an *explicit jump*).

Section 45.6: Run-time error '3': Return without GoSub

Incorrect Code

```
Sub DoSomething()
    GoSub DoThis
DoThis:
    Debug.Print "Hi!"
    Return
End Sub
```

Why doesn't this work?

Execution enters the `DoSomething` procedure, jumps to the `DoThis` label, prints "Hi!" to the debug output, *returns* to the instruction immediately after the `GoSub` call, prints "Hi!" again, and then encounters a `Return` statement, but there's nowhere to *return* to now, because we didn't get here with a `GoSub` statement.

Correct Code

```
Sub DoSomething()
    GoSub DoThis
    Exit Sub
DoThis:
    Debug.Print "Hi!"
    Return
End Sub
```

Why does this work?

By introducing an `Exit Sub` instruction *before* the `DoThis` line label, we have segregated the `DoThis` subroutine from the rest of the procedure body - the only way to execute the `DoThis` subroutine is via the `GoSub` jump.

Other notes

`GoSub`/`Return` is deprecated, and should be avoided in favor of actual procedure calls. A procedure should not contain subroutines, other than error handlers.

This is very similar to Run-time error '20': Resume without error; in both situations, the solution is to ensure that the *normal execution path* cannot enter a sub-routine (identified by a line label) without an explicit jump (assuming `On Error GoTo` is considered an *explicit jump*).