

Budgets, business cases and more

Creating a business case for technical communicators



Communicator

The Institute of Scientific and Technical Communicators
Summer 2016

**Using methodologies:
OBASHI and Lean**

**Expanding possibilities
with MadCap Flare 12**

**Is your translated
data secure?**

**Taking care of
your health**



Creating VBA loops: part 2

Looping the loop to save repeating code. By **Mike Mee**

Introduction

In the first part of this article, I covered the most frequently used type of VBA loop, the `For ... Next` loop. In this article, I will introduce the other types of loop that you can use.

Do ... Loop

The `Do ... Loop` is useful for those situations where you do not know what the start or final values of the loop will be, so you want the code to continue until a particular event occurs. Here is the pseudo-code for a `Do ... Loop`:

Pseudo Code

```
Do
    [Code statements/functions]
    [Test for Exit]
    [Additional Code statements/functions]
Loop
```

This form of loop can, if not thoroughly tested, create an 'infinite loop' - that is, you cannot stop it from running.

There should always been a way that your loop can exit via the `Exit Do` command. Naturally, you should always save your work before testing out a new `Do ... Loop` in your code.

The following example code will loop indefinitely until the user enters the correct password:

```
Sub AskForPassword()
Dim strWord As String
Do
    strWord = InputBox("What is the correct password?")
    If strWord = "ISTC" Then Exit Do
Loop
MsgBox "You entered the correct password."
End Sub
```

Another way of preventing an infinite `Do ... Loop` is to ask a question before the loop and use the answer as part of the exit strategy.

```
Sub DoLoopWithExitTest()
Dim intCount As Integer
Dim strAnswer As String
strAnswer = InputBox("How many loops?")
intCount = 1
Do
    Debug.Print "This is loop #" & intCount
    If intCount = CInt(strAnswer) Then Exit Do
    intCount = intCount + 1
Loop
End Sub
```

Admittedly, the code above does not do a great deal as it just prints a set of numbers to the Immediate window, but it shows you how to `Exit` out of a `Do ... Loop`.

While ... Wend

The third type of loop is useful for those situations where the loop will run while the condition is true.

Pseudo Code

```
While <condition>
    [Code statements/functions]
    [Test for Exit]
Wend
```

This form of loop is, like the `Do ... Loop` combination, one that could run infinitely if you have not included an 'exit clause' in your code.

The following example will go through all of the tables in your document and delete them all. A message box will appear afterwards giving you the total of tables deleted.

```
Sub DeleteAllTables()
Dim lngTable As Long
Dim lngTabCount As Long

' Initialise counter
lngTabCount = 0

' Get a count of the tables
lngTable = ActiveDocument.Tables.Count

' Loop through each table in document
With ActiveDocument
    While lngTable <> 0
        .Tables(lngTable).Delete
        lngTabCount = lngTabCount + 1
        lngTable = lngTable - 1
    Wend
End With
MsgBox lngTabCount & " tables were removed from your document."
End Sub
```

The best way to exit this particular loop is by executing the contents of the loop if there are no tables in the document. This is what the `lngC <> 0` portion of the code does.

Here is another example of using a `While ... Wend` loop. This time, the code will go through your document and remove all of the hyperlinks. This differs slightly to how the above example removed the tables.

```
Sub RemoveAllHyperlinks()
With ActiveDocument
    While .Hyperlinks.Count > 0
        .Hyperlinks(1).Delete
    Wend
End With
End Sub
```

The reason for the deleting the first hyperlink in each iteration is that when Word deletes it, all of the other hyperlinks move up by one. The second hyperlink becomes the first, and so on. This is why I covered the reverse `For ... Next`

loop in the Spring 2016 article. Sometimes VBA forces you to go backwards, depending on what you need to do!

Note that there is no error checking in the `While ... Wend` loop examples, but you can easily include a message box to ask the user if they want to unlock each table or delete each hyperlink.

Do While loops

This is a combination of the two previous loop types described. If the result of the check is false, the loop continues, otherwise it stops.

Pseudo Code

```
Do While <condition>
    [Code statements/functions]
    [Test for Exit]
Loop
```

Here is an updated example of the earlier

`Do ... Loop` password entry code, but this time using `Do While`.

```
Sub AskForPassword2()
Dim strWord As String
Do While strWord <> "ISTC"
    strWord = InputBox("What is the
correct password?")
Loop
MsgBox "You entered the correct password."
End Sub
```

Do Until loops

This loop acts in a similar way to the `Do While` loop, but the loop continues until a condition becomes true and then it will stop.

Pseudo Code

```
Do Until <condition>
    [Code statements/functions]
    [Test for Exit]
Loop
```

Here is the same password entry example as before, but this time using `Do Until`.

```
Sub AskForPassword3()
Dim strWord As String
Do Until strWord = "ISTC"
    strWord = InputBox("What is the
correct password?")
Loop
MsgBox "You entered the correct password."
End Sub
```

For all of the password entry examples, you can change 'ISTC' to whatever you want.

For Each loops

Finally, we have the `For Each` type of loop.

Pseudo Code

```
For Each WordObjectType In Collection
    [Code statements/functions]

    [Test for Exit]
    [Additional Code statements/functions]
Next
```

Defining variables

There are three places where you can declare your variables:

1. within the procedure (also known as local variables)
2. within the module
3. at global level.

Variable defined within the procedure

The usual place to declare variables is within a procedure before you need to use the variable. You can declare a variable, using `Dim` at any time up to the first time you use the variable.

Some programmers like to place all variable declarations at the start of the procedure, but that is not necessary. However, it might ensure that your code becomes a lot more readable when you need to debug it.

Here's an example which places the `Dim` statement within the `Sub / End Sub` commands:

```
Sub DisplayName()
Dim strWord As String
    strWord = InputBox("What is the your name?")
    MsgBox "Your name is " & strWord & "."
End Sub
```

Variable defined at module level

You can declare variables at the module level using the `Private` command. Module-level variables are available to all procedures in the module. Place any module-level variables before the first procedure in the module. For example:

```
Option Explicit
Private strWord As String
Sub DisplayName()
    strWord = InputBox("What is the your name?")
    MsgBox "Your name is " & strWord & "."
End Sub
```

Variable defined at global level

The third option is to define the variable at global level using the `Public` command. This ensures that every module, and every subroutine within your code, can use this particular variable. For example:

```
Option Explicit
Public strWord As String
Sub DisplayName()
    strWord = InputBox("What is the your name?")
    MsgBox "Your name is " & strWord & "."
End Sub
```

Figure 1. Defining variables

Table 1. Types of variables

Variable Type	What can be stored in it
Boolean	True or False values.
Byte	0 to 255.
Integer	-32,768 to +32,768.
Long	Values can be from approximately -2 billion to +2 billion.
Currency	Decimal numbers with as many as 19 digits.
Single	Single-precision, floating-point numbers.
Double	Double-precision, floating-point numbers.
Date	A date or time value.
String	Text
Object	Any object, such as a Word document or a window.
Variant	A generic number or string.

Note: Single or Double variables are more useful in Excel than Word.

This type of loop is useful for going through particular Word objects, such as the complete document, or the contents thereof, including tables, images, comments and paragraphs.

For those of you reading this and wondering how I will tweak the password entry test as a `For Each` loop, I will have to disappoint you. This kind of loop is not suitable for that kind of work, so I will include some completely different examples.

In the first example, the `For Each` loop will go through all of the documents that are currently open, and add their filenames to a string called `strName`. After the loop is complete, `strName` will contain every filename. The `MsgBox` command will display all of the filenames afterwards.

```
Sub ListAllDocuments()
Dim objDoc As Object
Dim strName As String
For Each objDoc In Documents
    strName = strName & objDoc.Name &
vbCr
Next objDoc
MsgBox "Documents open:" & vbLf & vbLf &
strName
End Sub
```

This example uses `vbCr` for a carriage return and `vbLf` for a line feed.

For the second `For Each` example, the code will go through every paragraph in your document and remove the bottom border.

```
Sub RemoveAllHorizontalLines()
Dim parLines As Paragraph
With ActiveDocument
    For Each parLines In .Paragraphs
        parLines.Borders(wdBorderBottom).
LineStyle = wdLineStyleNone
    Next parLines
End With
End Sub
```

You might notice that the code does not check if the formatting is already there, it just assumes that your text in each paragraph has that formatting enabled. I will leave it up to you to add that check to the code.

The final example of a `For Each` loop is taken directly from my Word Toolbox. This code removes the footers from each page so that I can insert my table with the various fields for document title, page number and such like.

Do not worry; I am not going to drop code of that magnitude into this article, as that would be cruel. I will break down the code to a simpler level and show you how to clear both the headers and footers from the document using an outer `For Each` loop, with two `For Each` loops inside, one covering the headers, the other for the footers.

```
Sub ZapHeaderAndFooter()
Dim objSection As Section
Dim oHF As HeaderFooter

With ActiveDocument

    ' Loop through each section
    For Each objSection In .Sections
```

```
        ' Loop through the headers
        For Each oHF In objSection.Headers
            oHF.Range.Delete
        Next
        ' Loop through the footers
        For Each oHF In objSection.Footer
            oHF.Range.Delete
        Next
    Next objSection
End With
End Sub
```

The above code loops through each section in your document. As each section can have different header and footer types, two further loops are required. The first sub-loop goes through each header and deletes its contents. The second sub-loop does the same for every footer. Then the code moves onto the next section, and so on, until the end of the document.

Loop Optimisation

You can speed up your loops, especially if you are intending to use them on large documents. Some of the easiest ways to implement this are:

- use the `With` command
- turn off the screen updates
- if the document is using Track Changes, turn them off, albeit temporarily.

With Command

The `With` command has two benefits. It can make your code easier to read and it can speed it up. It follows this layout:

```
With objectExpression
    [statements]
End With
```

I have been including examples of the `With` command in a lot of my articles. Here is a simple example.

```
With ActiveDocument.Paragraphs(1).Range
    .Font.Bold = True
    .Font.Name = "Arial"
End With
```

All the above code does is change the font in the first paragraph to Arial and make it bold. The `With` command saved you from typing in `ActiveDocument.Paragraphs(1).Range` in front of both commands. Your code listing is smaller and VBA will speed up slightly because it is not having to go back up the chain to `ActiveDocument`, then to `Paragraphs(1)` and finally to `Range` before it applies the changes.

You can also nest `With` commands, using this layout (as an example):

```
With objectExpression
    [statements]
    With objectExpression2
        [statements]
    End With
End With
```

Here is an example that has an outer `With` followed by two inner ones:

```
Private Sub DoubleWiths()
With ActiveDocument.Paragraphs(1)

    With .Range.Font
        .Bold = True
        .Name = "Arial"
    End With
    .LineSpacingRule = wdLineSpaceSingle

    With .Borders(1)
        .LineStyle =
wdLineStyleDouble
        .ColorIndex = wdBlue
    End With
End With
End Sub
```

The above example looks at the first paragraph and changes the text to bold. It sets the line spacing to single then adds a blue border (double-lined) around the text.

Imagine how long-winded that example would look if you had included each `ActiveDocument.Paragraphs(1)` before each command.

Screen Display

The first is to tell Word not to update the screen. The VBA command `Application.ScreenUpdating` toggles Word's display updates on or off. Word stores the current value in a Boolean variable.

To turn the screen updates off:

```
Application.ScreenUpdating = False
```

To turn the screen updates back on:

```
Application.ScreenUpdating = True
```

If you insert this command before you start any of your loops, and switch it back on again after the loop has completed, the end user will not see anything happen until it is complete.

Track Changes

The second way is; check if Track Changes is on, switch it off, then switch it back on again after your code has finished.

The Boolean variable that will store the current setting needs defining as a global variable via the `Public` and not `Dim` command.

In my Word Toolbox, I set all of my global variables up in a separate code module. This way, any other routine can use the variable without it redefining it in each module that will use it.

My article in the Autumn 2015 issue of *Communicator*, covers the different types of variables used in VBA in more depth.

```
Public blnTrackChanges As Boolean
```

You can now store the current setting for Track Changes, by querying the

`ActiveDocument.TrackRevisions` command. You can work out if it is switched on or off in the current document, depending if the result is `True` or `False`.

In the following code example, I have not put the `Sub/End Sub` surrounds on this code; this is because you will insert it inside each routine that needs it.

```
' Store current Track Changes current
setting then switch them off
With ActiveDocument
    blnTrackChanges = .TrackRevisions
    .TrackRevisions = False
End With
```

When your macro has finished tweaking the document, you can then switch the Track Changes to its previous state.


```
' Restore Track Changes to its previous state
ActiveDocument.TrackRevisions =
blnTrackChanges
```

The document will still have Track Changes switched on, if it was on when it was loaded, without any of the changes recorded.

However, you might prefer to see all of the changes that your macro(s) do to your document, so this is a personal choice.

The End (of the loop)?

I have now covered the main types of loop that you will use within your VBA macros.

You now need to find something annoying about a particular item in one of your documents that you would normally have to clean up manually one-by-one - and automate it. Good luck, but remember to keep backups! 

Further reading

Mee M (2015) 'Variables and screen output/input' *Communicator*, Autumn 2015: 44-47

Mee M (2016) 'Creating VBA loops: part 1' *Communicator*, Spring 2016: 34-36

Microsoft 'Declaring Variables' <https://msdn.microsoft.com/en-us/library/office/gg264241.aspx?f=255&MSPPError=-2147217396> (accessed April 2016)



Mike Mee MISTC is a technical author working at CDL in Stockport.

E: mike.mee@cdl.co.uk

T: @Mug_UK

W: <http://mikestoolbox.weebly.com> - my toolkit for Word 2007-2016 (the full source code is also available on the website).