XML PRESS
http://xmlpress.net

Optimizing VBA Code

Part Two: Looping

by Robert Delwood

This is part two of "Optimizing VBA Code.". You do not need to read Part One: Variables to use the information in this article, but I hope you do. The material in both articles is derived from my book, The Secret Life of Word.

This article looks into optimizing loops and the use of objects in loops.

Use the `For Each` Loop

The `For Each` loop is a specialized loop that repeats based on the number of items in the specified collection. The practical advantage is you don't need to know how many items there are (as you would with the conventional `For Next` loop). If there are no items, the loop isn't even entered. The optimization advantage is that the loop is almost always quicker and more efficient. That is, this:

```
Dim myPara as Paragraph
For Each myPara in ActiveDocument.Paragraphs
    'Do something
Next
```

is always faster than this:

```
Dim i as Long
for i = 1 to ActiveDocument.Paragraphs.Count
    'Do something
Next
```

Keep Object References Simple

A Word document is collection of systems. Each document has to track hundreds of items such as words, sections, misspellings, links, footnotes, printers, etc. Each system is an object, which means that a document is nothing more than a series of objects. Each of those objects may contain other objects, and this nesting can go on for quite a while.

VBA statements use "dot command" syntax. For example, `ActiveDocument.Name` accesses the property `Name` in an object named `ActiveDocument`. This is called a "simple object," but things get more complex. Consider the following common example:

```
ActiveDocument.Paragraphs(1).Range.Font.Bold = False
```

This example accesses a property (`Bold`), too, but the object that contains this property is nested inside other objects, five deep to be precise (`ActiveDocument`, `Paragraphs` (the 1 in parentheses means this is the first paragraph), `Range`, and lastly `Font`). An object nested like this is called a "compound object."

The optimization point is that accessing an object, especially a compound object nested five deep, takes a considerable amount of time. If you access the same compound object more than once, you have two options to make it more efficient. The first option is to use the `With` statement. This statement specifies an object path name that will be used in a series of statements that follow. VBA will access the object just once, then use it multiple times in subsequent references without recalculation. For example:

Email: <publisher@xmlpress.net>

```
With ActiveDocument.Paragraphs(1).Range.Font
     .Bold = false
     .Name = "Verdana"
     ActiveDocument.Range.Insert After "The background shading color is"
     & .Shading.BackgroundPatternColor
End With
```

This example defines an object path (`ActiveDocument.Paragraphs(1).Range.Font`). Then, within the `With ... End With` block, you can access any property or object within that path by starting the reference with "." followed by the rest of the path rather than retyping the entire name. Of course, if you need to access other objects inside the `With` statement, you can do so using the normal syntax (like `ActiveDocument.Range.Insert` in the example above).

The other option is to use the `Set` statement to create a new object defined from the object path. For example:

```
Dim myFont as Font
Set myFont = ActiveDocument.Paragraphs(1).Range.Font
myFont.Name
```

Now you can access the `ActiveDocument.Paragraphs(1).Range.Font` object using `myFont`, and again avoid repeatedly evaluating the complete compound path.

Look Inside the Loop

Since a loops, any kind of loop, can repeat actions multiple times, it makes sense to review the code inside a loop. Here are some things to look for:

- **Keep objects as simple as possible:** Use one of the options mentioned in the previous section if you are referencing objects nested more than two deep.
- **Use as few `Dim` statements as possible:** If you don't need to create a new variable inside a loop, don't. Declare it and initialize it before starting the loop. If a statement evaluates the same way each time, move it outside the loop.
- **End the loop as soon as possible:** If there is a special end condition, consider checking for that and exiting the loop early.
- **Consider removing the `DoEvents` statement:** This statement allows the operating system to catch up on events outside of the VBA program. It also is a very time consuming call. Many of the examples in The Secret Life of Word include `DoEvents` in each loop because that allows you to set a breakpoint in the statement during debugging. If you happen to create an endless loop, and you do not include a `DoEvents` statement, your only option is to force Word to end (type CTRL-ALT-DELETE, select Word in the Applications tab, and click End). However, once you're satisfied the macro is working as you intended, consider removing the `DoEvents` statements.

Testing Simple vs. Compound Object Speeds

In case you're wondering whether any of this makes any difference, the following code demonstrates the time savings of using simple objects over compound objects. Use a relatively large document of your own and run the code shown below against it (The example code is available at http://xmlpress.net/publications/word-secrets/examples/. To load the macro,

1. Open the Word document,
2. Press **ALT-F11** (to open the IDE),
3. Double click ThisDocument in the Project Window, and
4. Copy and paste the code into ThisDocument module.

To run it, from within the document click Developer|Code|Macros, select SpeedTest, and click Run. Your values will be different but on sample 20,000 word documents the times were consistently two seconds for the simple object (case 1) and about 140 seconds for the compound objects (case 2).

```
Option Explicit

Public Sub SpeedTest()
    On Error GoTo MyErrorHandler

    Dim currentDocument As Document
    Set currentDocument = ActiveDocument

    Dim timeStart As Date
    Dim finalTime_Case1 As Long
    Dim finalTime_Case2 As Long

    ' --- Start case 1 ---
    timeStart = Now
    Dim myParagraph As Paragraph
    For Each myParagraph In currentDocument.Paragraphs
        Debug.Print myParagraph.Range.Text
        DoEvents
    Next
    finalTime_Case1 = DateDiff("s", timeStart, Now)

    ' --- Start case 2 ---
    timeStart = Now
    Dim i As Long
    For i = 1 To currentDocument.Paragraphs.Count
        Debug.Print currentDocument.Paragraphs(i).Range.Text
        DoEvents
    Next i
    finalTime_Case2 = DateDiff("s", timeStart, Now)

    ' --- Show results ---
    Debug.Print "finalTime_Case1: " & finalTime_Case1
    Debug.Print "finalTime_Case2: " & finalTime_Case2

    Exit Sub

MyErrorHandler:
    MsgBox "SpeedTest" & vbCrLf & vbCrLf & "Err = " & _
       Err.Number & vbCrLf & "Description: " & Err.Description
End Sub
```

Summary

While these are only a few of the many optimization techniques available, they can produce large speed gains.