

Conclusion and Improvements

| | | | | | |
|--------------------|--------------|---------------------|--------------|---------------------|---------------|
| Created by: | Marek Prazak | Reviewed by: | Marek Prazak | Create Date: | 19 April 2019 |
|--------------------|--------------|---------------------|--------------|---------------------|---------------|

| | | |
|-------|---|----|
| 1. | PURPOSE..... | 2 |
| 2. | DEFINITION | 2 |
| 3. | CONTENT | 2 |
| 3.1 | VBA ERROR HANDLING..... | 2 |
| 3.1.1 | VBA ERRORS | 3 |
| 3.1.2 | ON ERROR STATEMENT | 5 |
| 3.1.3 | THE ERR OBJECT..... | 9 |
| 3.2 | BEST PRACTICES..... | 9 |
| 3.2.1 | TURN OFF NON ESSENTIALS | 9 |
| 3.2.2 | READ/WRITE IN A SINGLE OPERATION..... | 10 |
| 3.2.3 | AVOID SELECTING / ACTIVATING OBJECTS..... | 11 |
| 3.2.4 | WITH... END WITH | 12 |
| 3.3 | CODE DESCRIPTION | 13 |
| 3.4 | INSTRUCTIONAL ONE PAGER..... | 13 |
| 3.5 | Review of code | 14 |
| 4. | CLOSING EXERCISE | 17 |
| 5. | ADDITIONAL KNOWLEDGE | 18 |

1. PURPOSE

- This lesson will take you through the best practices for VBA users, such as how to deal with errors. How to make your macro more efficient. How to compose simple instructional one pager. Why use code description.
- The lesson should take approximately 1 hour to finish.

2. Definition

- VBA Error Handling - Error Handling refers to code that is written to handle errors which occur when your application is running. These errors are normally caused by something outside your control like a missing file, database being unavailable, data being invalid etc.

3. CONTENT

3.1 VBA ERROR HANDLING

- If we think an error is likely to occur at some point, it is good practice to write specific code to handle the error if it occurs and deal with it.

Code description for handling errors.

| Item | Description |
|------------------------------|---|
| On Error Goto 0 | When error occurs, the code stops and displays the error. |
| On Error Goto -1 | Clears the current error setting and reverts to the default. |
| On Error Resume Next | Ignores the error and continues on. |
| On Error Goto [Label] | Goes to a specific label when an error occurs. This allows us to handle the error. |
| Err Object | When an error occurs the error information is stored here. |
| Err.Number | The number of the error. (Only useful if you need to check a specific error occurred.) |
| Err.Description | Contains the error text. |
| Err.Source | You can populate this when you use Err.Raise. |
| Err.Raise | A function that allows you to generate your own error. |
| Error Function | Returns the error text from an error number. Note: This type is no longer in use, we chosen to add this, in case that you'd encounter it in the future. |
| Error Statement | Simulates an error. Use Err.Raise instead. |

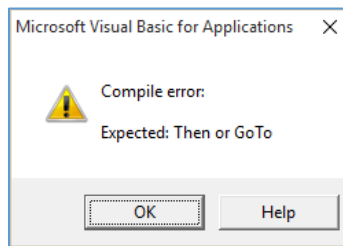
Note: This type is no longer in use, we chosen to add this, in case that you'd encounter it in the future.

3.1.1 VBA ERRORS

- There are three types of errors in VBA

A. Syntax

- When you type a line and press return, VBA will evaluate the syntax and if it is not correct it will display an error message.
-For example in case where you type **If** and forget the **Then** keyword, VBA will display the following error message and highlight the incorrect line



- Syntax errors relate to one line only. They occur when the syntax of one line is incorrect.

Note: You can turn off the Syntax error dialog by going to Tools->Options and checking off "Auto Syntax Check". The error line will still appear highlighted, if there is an error but the dialog will not appear.

B. Compilation

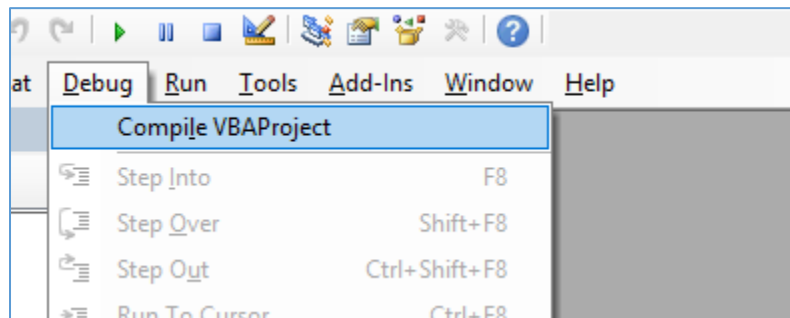
- **Compilation errors** occur over more than one line. The syntax is correct on a single line but is incorrect when all the project code is taken into account.

Examples of compilation errors:

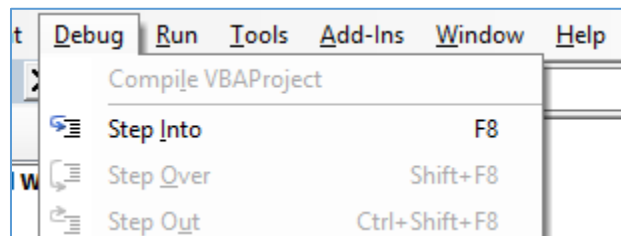
- If statement without corresponding **End If** statement
- Variables not declared (**Option Explicit** must be present at the top of the module)
- Calling a Sub or Function that does not exist

-Using Debug->Compile

- To find compilation errors, we use Debug->Compile VBA Project from the Visual Basic menu.



- When you select Debug->Compile, VBA displays the first error it comes across.
- When this error is fixed, you can run Compile again and VBA will then find the next error.
- Debug->Compile will also include syntax errors in its search which is very useful.
- If there are no errors left and you run Debug->Compile, it may appear that nothing happened. However, "Compile" will be grayed out in the Debug menu. This means your application has no compilation errors at the current time.



- You should **always use Debug->Compile before you run your code**. This ensures that your code has no compilation errors when you run it.
- If you do not run Debug->Compile then VBA may find compile errors when it runs. These should not be confused with Runtime errors.

C. Runtime

- Runtime errors occur when your application is running. They are normally outside of your control but can be caused by errors in your code.
- For example, imagine your application reads from an external workbook. If this file gets deleted, then VBA will display an error when your code tries to open it.

-Expected/Unexpected Errors

- If we think a runtime error could occur we put code in place to handle it. For example, we would normally put code in place to deal with a file not being found.
- The following code checks if the file exists before it tries to open it. If the file does not exist, a user friendly message is displayed and the code exits the sub.

```
Sub OpenFile()  
  
    Dim sFile As String  
    sFile = "C:\docs\data.xlsx"  
  
    ' Use Dir to check if file exists  
    If Dir(sFile) = "" Then  
        ' if file does not exist display message  
        MsgBox "Could not find the file " & sFile  
        Exit Sub  
    End If  
  
    ' Code will only reach here if file exists  
    Workbooks.Open sFile  
  
End Sub
```

- When we think an error is likely to occur at some point, it is good practice to add code to handle the situation. We normally refer to these errors as **expected errors**.
- If we don't have specific code to handle an error it is considered an unexpected error. We use the VBA error handling statements to handle the **unexpected errors**.

3.1.2ON ERROR STATEMENT

- The VBAs On Error statement is used for error handling. This statement performs some action when an error occurs during runtime.
- There are four different ways to use this statement

1. On Error GoTo 0

- When an error occurs, the code stops at the line with the error and displays a message. *(This is the default behavior of VBA. In other words, if you don't use **On Error** then this is the behavior you will see.)*
- The application requires user intervention with the code before it can continue. This could be fixing the error or restarting the application. In this scenario no error handling takes place.

2. On Error Resume Next

- tells VBA to **ignore the error** and continue on. No error message is displayed. *(There are specific occasions when this is useful, but most of the time **you should avoid using it.**)*
- If we add **Resume Next** to our example Sub, VBA will ignore the divide by zero error.

```
Sub UsingResumeNext ()  
  
    On Error Resume Next  
  
    Dim x As Long, y As Long  
  
    x = 6  
    y = 6 / 0  
    x = 7  
  
End Sub
```

- It is **not** a good idea to do this. If you **ignore the error**, the **behavior can be unpredictable**. The problem is that you **aren't aware** that **something went wrong**, because you have suppressed the error.
- The code below is an example of where using **Resume Next** is valid.

```
Sub SendMail()  
  
    On Error Resume Next  
  
    ' Requires Reference:  
    ' Microsoft Outlook 15.0 Object Library  
    Dim Outlook As Outlook.Application  
    Set Outlook = New Outlook.Application  
  
    If Outlook Is Nothing Then  
        MsgBox "Cannot create Microsoft Outlook session." _  
            & " The email will not be sent."  
        Exit Sub  
    End If  
  
End Sub
```

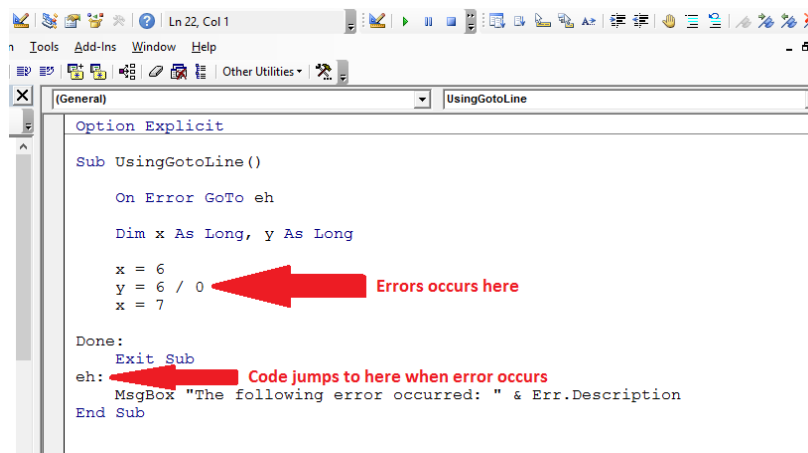
- In this code we are checking to see if Microsoft Outlook is available on a computer. All we want to know is if it is available or not. We are not interested in the specific error.
- In the code above, we **continue on** if there is an **error**. Then in the next line we check the value of the Outlook variable. If **there had been an error**, the value of this variable will be set to **Nothing**.
- This is an example of when **Resume** could be **useful**. The point is that even though we use **Resume** we are still **checking for the error**.

3. On Error GoTo [label]

- the code moves to a specific line or label. No error message is displayed. This is the one we use for error handling.

```
Sub UsingGotoLine()  
  
    On Error GoTo eh  
  
    Dim x As Long, y As Long  
  
    x = 6  
    y = 6 / 0  
    x = 7  
  
Done:  
    Exit Sub  
eh:  
    MsgBox "The following error occurred: " & Err.Description  
End Sub
```

- The screenshot below shows what happens when an error occurs



- VBA jumps to the **eh** label because we specified this in the **On Error Goto** line.

Note 1: The label we use in the `On...GoTo` statement, **must be in the current Sub/Function**. If not, you will get a compilation error.

Note 2: When an error occurs, when using `On Error GoTo [label]`, the error handling returns to the default behavior i.e. The code will stop on the line with the error and display the error message.

Note 3: Only **one End sub** can exist in sub. We use **Exit sub** as a substitute.

4. On Error GoTo -1

- This statement is used to clear the current error rather than setting a particular behavior. You might see this being used in combination with **On Error Resume Next** where you want to reset the error that you've accumulated before going through additional error check.

3.1.3 THE ERR OBJECT

- When an error occurs you can view details of the error using the **Err object**.
- When a runtime error occurs, VBA automatically fills the **Err object** with details.
 - Below line displays error details

```
Debug.Print "Error number: " & Err.Number _  
            & " " & Err.Description
```

"Error Number: 13 Type Mismatch"

- The **Err.Description** provides details of the error that occurs.
- The **Err.Number** is the ID number of the error e.g. the error number for "Type Mismatch" is number 13.
- The number of errors is quite large and remembering all is impossible. In case that you want to get familiar with the errors you can use link in additional knowledge section, but we would suggest to dive deep into the error description, when you actually encounter such error.

3.2 BEST PRACTICES

- It is possible that you might have already noticed in your daily routine, some excel files or macros take surprisingly long time to run operations that you want from them (Print 200 emails, calculate and copy data from multiple sheets with formulas, etc.). Such things happen, as VBA is performing every single step you've written in it and in many cases also showing you the steps.
- To speed up your macros and make it more reliable, we will show you few tips.

3.2.1 TURN OFF NON ESSENTIALS

- Speeds up the macro as you won't see every single step happen on your screen.
- Feel free to visit **lesson 7** to refresh on this code.

3.2.2 READ/WRITE IN A SINGLE OPERATION

- Instead of looping through cells one at a time and getting or setting a value, do the same operation over the whole range in one line, using an array variable to store values as needed.

A. Looping

- this code loops 60000 times which will take few min to perform, such solution is not very efficient especially if you want to use it few times in your macro. It might be sustainable in case where you have only few rows.

```
Dim DataRange As Range
Dim Irow As Long
Dim Icol As Integer
Dim MyVar As Double

Set DataRange = Range("A1:C10000")

For Irow = 1 To 10000
    For Icol = 1 To 3
        'Read values from the Excel grid 30K times
        MyVar = DataRange(Irow, Icol)

        If MyVar > 0 Then
            'Change the value
            MyVar = MyVar * MyVar
            'Write values back into the Excel grid 30K times
            DataRange(Irow, Icol) = MyVar
        End If
    Next Icol
Next Irow
```

B. Using data range to split the time by half

```
Dim DataRange As Variant
Dim Irow As Long
Dim Icol As Integer
Dim MyVar As Double

'read all the values at once from the Excel grid, put into an array
DataRange = Range("A1:C10000").Value

For Irow = 1 To 10000

    For Icol = 1 To 3

        MyVar = DataRange(Irow, Icol)
        If MyVar > 0 Then
            'Change the values in the array
            MyVar = MyVar * MyVar
            DataRange(Irow, Icol) = MyVar
        End If

    Next Icol

Next Irow

'writes all the results back to the range at once
Range("A1:C10000").Value = DataRange
```

- C. Another solution to this would be to use **Power Query** as explained in **lesson 10**. Which would bring down the time from minutes into seconds.

3.2.3 AVOID SELECTING / ACTIVATING OBJECTS

- This next optimization minimizes how frequently Excel has to respond to the selection changing in the workbook by minimizing the selection changing as much as possible.
- It is considered as best practice to **avoid Select all together**. There are many different ways how to avoid select that we encourage you to use. Just to name a few, **with/end with**(below), **referencing objects**, **sheets** and as already mentioned by using **power query** avoiding the need of select altogether

A. Select

- The code below selects each single shape and then inputs the string.

```
For i = 0 To ActiveSheet.Shapes.Count  
    ActiveSheet.Shapes(i).Select  
    Selection.Text = "Hello"  
Next i
```

B. Without select

- This method is much faster as there is no need to select anything, the object is only referenced.

```
For i = 0 To ActiveSheet.Shapes.Count  
    ActiveSheet.Shapes(i).TextEffect.Text = "Hello"  
Next i
```

3.2.4 WITH... END WITH

- This statement allows you to write shorter code by referring to an object only once instead of using it with each property, which will make your code more efficient.
 - As an example, suppose we want to change various aspects of the ActiveCell. We want to change the font name, the font size, the boldness, the italics, etc.

```
ActiveCell.Font.Bold = True  
ActiveCell.Font.Color = vbBlue  
ActiveCell.Font.Name = "Arial"  
ActiveCell.Font.Size = 22  
ActiveCell.Font.Italic = True
```

- But notice the repetition here. We've used **ActiveCell.Font** five times. By using a With Statement, we can just type the **ActiveCell.Font** once.

```
With ActiveCell.Font  
    .Bold = True  
    .Color = vbBlue  
    .Name = "Arial"  
    .Size = 22  
    .Italic = True  
End With
```

- In example above it does not seem like big deal as using copy paste five times would do the trick. But imagine having code with 200 rows where you have constantly write *thisworkbook.activeworkbook.range(".....* such thing will unnecessarily slow you down, in addition it will make the code slower.
- **With Statements** are quite intuitive, just **remember**: if you're typing the same object over and over, you might do better to use a **With ... End With statement**.

3.3 CODE DESCRIPTION

- As you might have already noticed in previous lesson by using **apostrophe** in your code that section will not be considered part of the code. Which is useful if you want to **disable part of the code** but more importantly it is used to create **comments**.

```
'below is actual code  
Dim myname As String
```

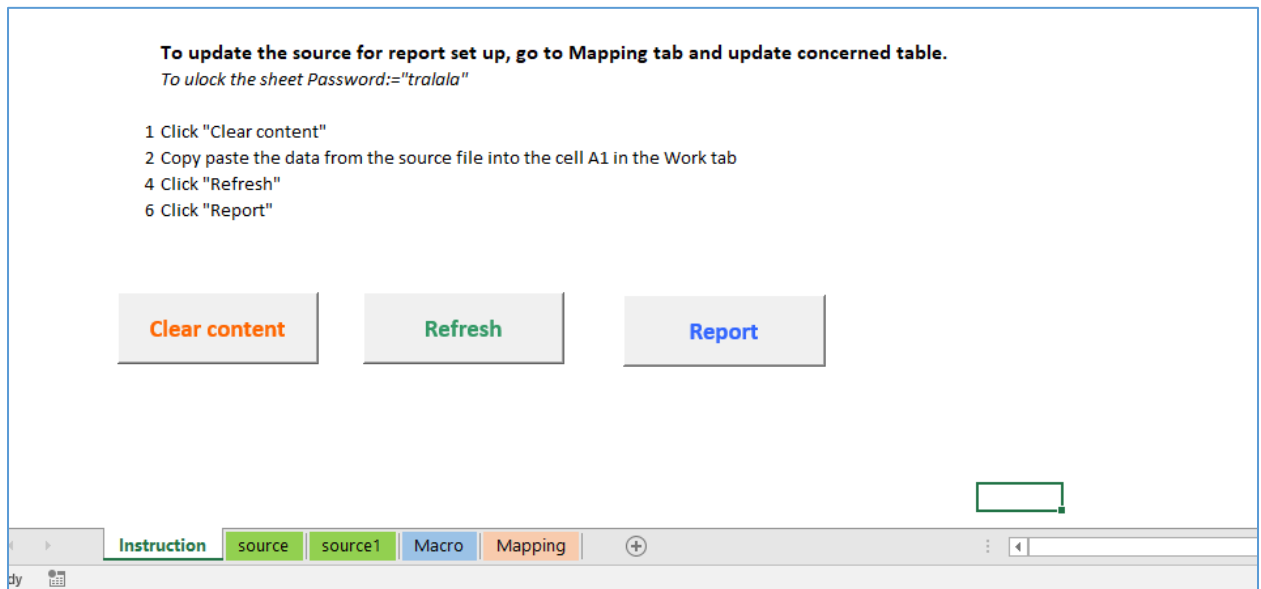
```
'below code is disabled  
'Dim myname As String
```

- **Why should use comment your code?**
 - It is considered as **best practice** to comment your code with short description, as to make the coding **easier for yourself** (some code sequences can be long and complex) and at the same time you'll **save time** to anyone who comes after you as they won't have to spend so much time **understanding** your code.
- Try to keep the comments **short** and **to the point**

3.4 INSTRUCTIONAL ONE PAGER

- It is recommended to create instruction page to your macro.
- As is with the description of the code the instruction should be **short** and **to the point** so anyone can use the macro **without long** and **extensive practice**.

Example of one pager below:



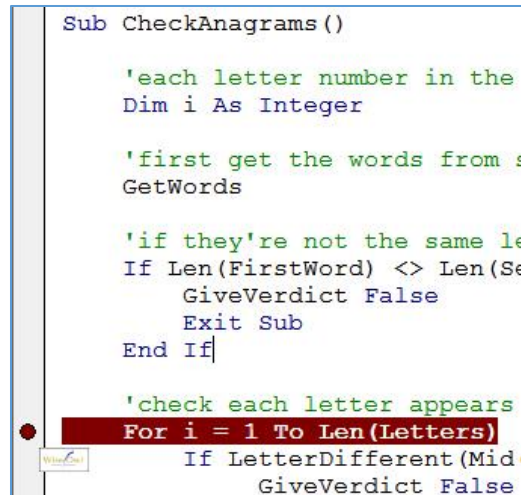
- When creating macro, it is good to keep **the main buttons in the instruction page** as compared to having each sheet with refresh, send, etc. This will prevent accidental run of macro and at the same time it will not limit the admins from using other parts of the excel (tracker, etc.)
- Don't forget that this is **only basic instruction** of the macro and how to make it work and **not an SOP** of the process or code.
- Even when one pager is created, **SOP has to be updated** with detailed description of process and macro (done by admins).

3.5 REVIEW OF CODE

- Regardless whether the code is yours or someone else's, when the need arises for you to review code or find error below you can find few pointers on how to proceed.
 - **Run the code to see any errors.**
 - Try to run the macro according to instructions.
 - Try to run the macro and use nonsense data or incorrect data (words in date section, etc.), don't fill required cells, etc.
 - **Run the code linebyline**
 - Open the VBA editor and find the start of the macro. Press **F8** to do one line of the code.
 - If you've encountered an error or your macro is giving incorrect results, you can identify where the issue is.

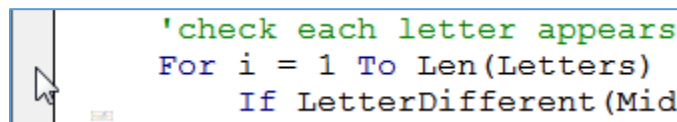
▪ Setting Breakpoint

- Often in a complicated or long macro you will want to avoid stepping through it line by line. To do this you can set a **breakpoint**



```
Sub CheckAnagrams()  
    'each letter number in the  
    Dim i As Integer  
  
    'first get the words from s  
    GetWords  
  
    'if they're not the same le  
    If Len(FirstWord) <> Len(Se  
        GiveVerdict False  
        Exit Sub  
    End If  
  
    'check each letter appears  
    For i = 1 To Len(Letters)  
        If LetterDifferent(Mid  
            GiveVerdict False
```

- When you run the macro it will go until this point, then you can use **F8** to do line by line.
- You can set or remove a breakpoint by clicking in the margin to the left of it.

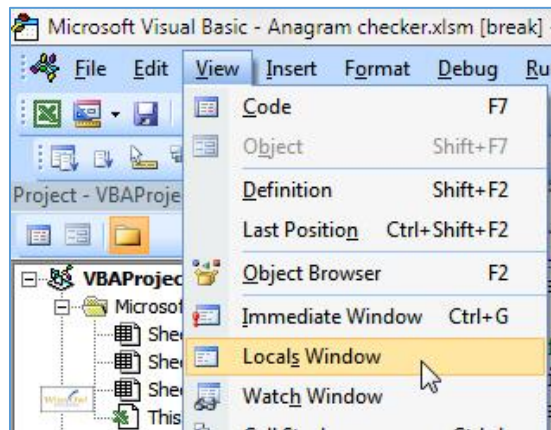


```
    'check each letter appears  
    For i = 1 To Len(Letters)  
        If LetterDifferent(Mid
```

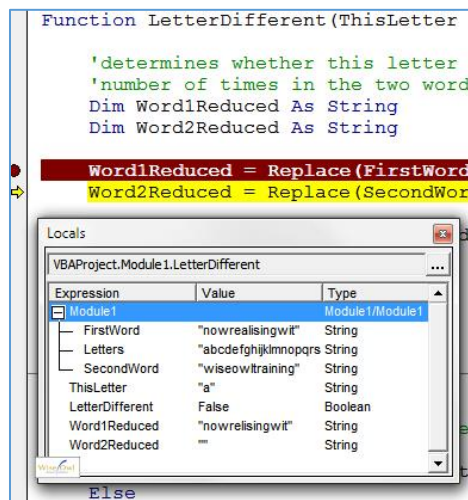
Note: A surprisingly useful shortcut key is **SHIFT + CTRL + F9**, which removes all of the breakpoints that you've set.

▪ The Locals Window

- The *Locals* window isn't particularly well named, as it allows you to **see the value of local and global variables**. It might be better called the **Variables window**.



- Here's an example of the **Locals window**:



- If you combine this with F8 and breakpoints you'll easily find if your variables contain any values.

4. *CLOSING EXERCISE*

As purpose of this lesson was to give you additional information and tips on how to work with VBA, there is not one exercise, instead below you can find link to 52 optional exercises, on which you can practice what you've learned so far.

- **Optional excesses** <https://www.wiseowl.co.uk/vba-macros/exercises/excel-vba/>

5. Additional knowledge

More on Error handling - <https://excelmacromastery.com/vba-error-handling/>

List of Error description - <http://www.vba-market.com/list-of-vba-error-codes/>

Improving performance - [https://docs.microsoft.com/en-us/previous-versions/office/developer/office-2007/aa730921\(v=office.12\)](https://docs.microsoft.com/en-us/previous-versions/office/developer/office-2007/aa730921(v=office.12))

Utilizing Array/Range - <http://dailydoseofexcel.com/archives/2006/12/04/writing-to-a-range-using-vba/>

Working with breakpoints - <https://www.wiseowl.co.uk/blog/s196/breakpoints.htm>