Learning Excel VBA

About Variables

Prepared By Daniel Lamarche
ComboProjects

ComboProjects
Understanding does matter

About Variables

By Daniel Lamarche (Last update February 2017).

The term *variables* often send shivers in the back of many learning VBA in Excel. Maybe the reason is that it is *your* responsibility to create them, to choose the proper data type, understand their *initial* value, assigning values to them and keep track of what value is currently stored in them. That is a lot for a newbie!

This document will clarify all the above in various context. After you consider the material below you will master the uses of variables in your code. Good news is that there is nothing really difficult about them.

This document will discuss the use of *data* variables and introduce *object* variables.

What is a variable?

To see why variables are part of the ABC of programming let us go back a couple of years (more for some of us) when you were in school and the math teacher would utter the dreaded words:

Let x equals 5

As the teacher said that, the following was written on the board: x = 5.

Why do we call the letter x a variable? Because a couple of minutes later, x would be equal to a completely different value! Suddenly x is equal to 8! So the value of x is … variable!

An important point here

If we are to fully grasp how variables behave in VBA we need to clarify an important detail: The use of the equal character. For instance when we read “Let x equal to 5” we read it from left to right. In reality this is *not* what is happening in the arithmetical sense.

In programming as in arithmetic the equality of two things should not be considered from left to right but from right to left! Let us see why and this will clarify a couple of problems beginners have, especially when dealing with loops.

The expression: y = 8 really says “Store the value of 8 in the variable y”. Of course we read it from left to right but the equation is *resolved* from right to left. The following expression will make this clearer.

Suppose the variable y is assigned the value 8. Consider the following expression:

x = y + 2

Reading the above expression from left to right makes it more difficult to understand what is going on right? What if we resolve the right side of the equal *first* then store the result in the variable on the left side of the equality we get: y + 2 equals 10 so the equation becomes:

x = 10

Thus the value 10 is stored in the variable x.

Another example

When we agree to deal with resolving variables the way described above then it becomes easy to deal with expression like:

x = x + 1

As a VBA instructor I very often see people confused with the above equation. Some will ask “How can x be equal to x + 1? And they are right!

If we consider this from left to right then it is a bit of a puzzle. But let us consider it now from right to left. Suppose x is equal to 5 let’s resolve the expression x = x + 1.

From right to left:

x + 1 = 6

so the formula becomes:

x = 6

This simple expression increased the value of x by 1. In a loop we call this expression a *Counter*.

What type of variables are available

The VBA language comes with a fairly rich variety of *data type*. Let us see the most common ones and their initial value.

Data Type	Byte Used	Range
Byte	2	0 to 255
Integer	2	-32,768 to +32,767
Long	4	-2,147,483,648 to 2,147,483,647
Single	4	-3.402E+38 to -1.401E-45 for negative values 1.401E-45 to 3.402E+38 for positive values.
Double	8	-1.797E+308 to -4.940E-324 for negative values 4.940E-324 to 1.797E+308 for positive values.
Currency	8	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Boolean	2	-1 to 0 (-1 = True and 0 = False)
String	String Length	1 to 65,400 characters
Date	8	January 1, 100 to December 31, 9999
Object	4	Storing addresses that refer to Excel objects (worksheet, workbook, chart, etc.)

Just as a reminder an integer is a whole number; any number that has no decimal.

As you can guess Byte and Integer data types are perfect for small to medium size numbers. The Long data type covers the largest range of integer type numbers.

On the other hand Single and Double are used with number having decimal. If you use large numbers having lots of decimal use the Single data type. For very large numbers with a very large number of decimals and a great quantity of processing use Double. Be aware that the largest data type will require more processing on the part of the PC’s processor.

What can I store in a variable?

As shown in the table above, you can use variables to store *data*. Examples of data are numbers, dates and text strings. For sake of simplicity we will call these variables *data* variables.

However variables are often used to store another type of information. We will call them *objects*. A bit of an obscure word but it’s nothing difficult. In Excel, objects are *things* the user deals with daily when working in Excel. Examples are Workbooks, Worksheets, Charts and Pivot Tables.

The examples above are only a couple of examples of obvious *things* a user interacts with in Excel but there are more! Considers a couple more examples:

Users work with ranges of various sizes. In Excel even a single cell is a range. So the Range objects are common objects accessible to your VBA code. Columns, Rows, Selection, Page Breaks are also objects that are accessible in VBA code.

We will look into this important topic later in this document.

When to use variables

It is impossible for the macro recorder to create and use variables. Variables are only created by the programmers.

So whenever you need to store a number, some text or dates that will differ every time you need to run your procedure, store it in a variable. That same variable will be used for thousands of possible values the procedure will be fed with.

Give me a couple of examples

For example a procedure that will return the age of an asset or a person will require at the very least the date the asset was purchase or the birth date of a person. That date would be safely stored in the variable used for that purpose.

Another example would be a function that returns the commission paid to an employee based on the amount sold for a period and the number of years the employee worked for the company. In this example to do its job the procedure must be fed that information. The function would have, at the very least, two variables storing the amount sold and the number of years.

Declaring Variables

The term declaring variable refers to the way the VBA programming language informs (or makes available) to a procedure, variables you create. Declaring a variable involve specifying its data type.

To do this we use the term Dim. Below is a couple of example:

```
Dim sngCommission As Single
Dim bytYearsDifference As Byte
Dim bolInWeekend As Boolean
Dim strEmpFName As String
```

The term 'Dim' comes from the word *dimension*. A dimension is a *measure* of some sort. In VBA, each data type uses a different measure of computer's memory. Programmers in the 70's decided to use the term Dim to determine the length (or data type) of memory that the variable will require.

Notice that the 4 example above all start with a three characters *prefix*? This is not necessary; however it is common practice to prefix all variables with a prefix to remind you later in the code of the data type of a variable. Not everyone will use this approach and you may even hear that this habit is useless! Nevertheless in this document (and all PDF's available in this site) we will prefix all variables with a three characters prefix in lowercase.

It should be clear by now that the sole purpose of using variables is that every time the Sub of Function is used, it runs with different values. Variables will retain these values for the duration of the procedure.

Arguments are variables

Below is an example of a function. In the brackets next to the function name is the argument or parameter (intCel). The argument is indeed a variable and need to be declared with its data type.

```
Function Cel2Fah(intCel As Integer) As Integer
' Converts a Celsius to Fahrenheit. Uses whole numbers only.

    Cel2Fah = 9 / 5 * intCel + 32
End Function
```

Even the value returned by the function is qualified by specifying its data type. Here the parameter and the values returned are integers.

Failing to specify the data type

If you do not specify the data type when declaring a variable will cause VBA to assign a one itself. The default data type is Variant. VBA processes these data types slower because it must determine the most accurate data type for the assigned value. Depending to who you talk to, some will say that it's a bad choice and will slow down the processing.

Make sure you *always* assign a data type to all arguments, variables and returned values from functions. The data type should be proportional to the range of values that will be stored in it.

Declaring more than one variable in the same line

If your procedure will require a number of variables, you can declare more than one variable in the same line. In the example below three variables are declared separated by a comma:

```
Dim datDOB As Date, bytInStock As Byte, strCategory As String
```

Using this approach is perfectly acceptable. Note that you need only to use the keyword Dim once but every variable are typed!

Using your variables in the code

Once a variable had been declared and assigned a data type, you will start to use that variable. The first thing is to assign a value to the variable. Obviously you assign value to a variable using the equality character (the equal character). For example:

```
datDOB = Range("A1")        ' Data come from cell A1
bytInStock = 12             ' Number assigned
strCategory = "Fruits"      ' String assigned
```

The value assigned can be the result of an expression. For example:

```
' The concatenation is stored in a variable
strFullName = strFName & " " & strLName
```

Another example could be:

```
strFirstChar = Left(strPartNo, 1)
```

In the above example the first character of the value stored in the variable strPartNo is stored in the variable strFirstChar.

By the way

You might encounter the assignment of a value to a variable using the term Let. For example the last expression above could look like the following:

```
Let strFirstChar = Left(strPartNo, 1)
```

Nowadays programmers stop using the term Let this way. You can omit it because it is implied by VBA when using data variables.

Object Variables

The concept of object variables is akin to the one of data variables. As briefly discussed above, the important difference is that an object variable points to an Excel *thing*, not to some data. You will encounter them very often if you visit Web sites discussing even basic Excel VBA techniques.

To declare an object variable use the following syntax:

```
Dim var As ObjectType
```

Once it is declared you assign an object to it using the keyword Set as shown below:

```
Set var = Object
```

With object variable you *must* use the keyword Set. You remember that with data variables, VBA assumes the optional keyword Let.

Why use object variables?

You use object variables for a similar reason that you use data variables. With data variables the information will change. With object variables the procedure will use different object of a specific type.

Here are a couple of basic examples:

```
Dim wkbStores As Workbook
Dim wksOrders As Worksheet
Dim rngStaff As Range
```

You can see that some principles are the same as with data variables.

- The use of the keyword Dim
- The three character prefix reminding you of the type of object
- The variable are assigned a data type

Assigning an object to a variable

The only important difference between assigning data to a data variable and assigning an object to an object variable is the use of the keyword Set. The term Set is mandatory when tying an object variable to the object. Below are 3 examples:

```
Set wkbStores = Workbooks("Finance.xlxs")
Set wksOrders = Worksheets("January Sales")
Set rngStaff = Range("A1").CurrentRegion
```

A couple examples

In a new workbook saved as a Macro-Enabled Workbook, type 5 or 6 suburb in column A starting with cell A1 and make sure the cells containing the suburbs are selected

In a module type the following small Sub:

```
Sub ListSuburbs()
' Print each suburb in the Immediate Window.
    Dim rng As Range
```

```
    For Each rng In Selection
        Debug.Print rng.Value
    Next rng
End Sub
```

Make sure the Immediate Window is open (Ctrl+G). In the Immediate window type the Sub name (ListSuburbs) and hit the Enter key. The suburbs are printed in the Immediate window.

Because of the type of loop we used here, there is no need to use the Set keyword. The loop automatically assigns the value of *each* selected cell to the variable rng.

Create a couple of blank sheets in your workbook and type the following Sub:

```
Sub ListSheets()
    Dim wks As Worksheet

    For Each wks In Worksheets
        Debug.Print wks.Name
    Next
End Sub
```

When you run this sub, the Immediate window will list the names of all the worksheets in the current Workbook.

Storing a selection into an object variable

The next example is a common example of a selection stored in a variable. In this example we have a column of entries starting in cell A1 but between uses the number of rows differs. One day you have 12 rows and another day there will be 39.

The Sub will simulate the user making A1 the active cell and pressing Ctrl+Shihf+↓ to select from A1 to the last cell down.

```
Sub SelectColumn()
' Select the entries from cell A1 to the last entry down.
    Dim rng As Range

    Set rng = Range("A1", Range("A1").End(xlDown))
    Debug.Print rng.Count  ' Print the number of cells selected.
End Sub
```

This sub selects all cells in column A starting in A1 and stores that selection in the object variable rng.

A fascinating aspect of this approach is that the active cell didn't budge! It can be anywhere and the sub will not require the active cell to move at all. At the line `Debug.Print` you could have put any VBA instructions. You have access to everything you want to do with each cells in that selection. Cool? Indeed!

Storing a worksheet into an object variable

If we can store a selection into an object variable, we can as easily store a worksheet or even a workbook! It does not have to be the active sheet or even the active workbook.

In this next example we are going to write a couple of entries in a worksheet named Regions *without* even moving to that worksheet! In any module type the following simple Sub:

```
Sub WriteToAnotherSheet()
' Write the names of 5 regions in the Regions worksheet.
    Dim wks As Worksheet
```

```
    Dim rng As Range

    Set wks = Worksheets("Regions")
    Set rng = wks.Range("A1:A5")

    rng.Cells(1, 1) = "North"     ' The arguments for Cells are Cells(rows, cols)
    rng.Cells(2, 1) = "South"
    rng.Cells(3, 1) = "East"
    rng.Cells(4, 1) = "West"
    rng.Cells(5, 1) = "Central"
End Sub
```

Make sure you do have a sheet named *Regions* and ensure it is *not* the active sheet otherwise the trick is not fun anymore!

Click anywhere in your sub and hit the F8 key (Step In) until the yellow bar goes past the End Sub statement. It should be about 10 times. Then go to the sheet Regions at see that the regions are written exactly where you specified. Note that it didn't matter where the active cell was located!

Did you pick up how we assigned the range to the variable rng? By starting the assignation with wks VBA informs Excel that the range that follows is in that other worksheet!

Can you believe that you could have done the same thing with another workbook! Of course that workbook must be open and the sheet you want to use must exist.

About the Cells property

It took me some times to fully appreciate the Cells property. A while ago I would have typed wks.Range("A1") then wks. Range("A2") etc. But when a loop is required referring to each cell in a range can become tedious. Because the Cells property uses numbers only, it is easier to put a variable as the row or column (or both) arguments.

If you are familiar with both approach (The Range object and the Cells property) you are likely to make the correct choice for any particular requirement.

Last example

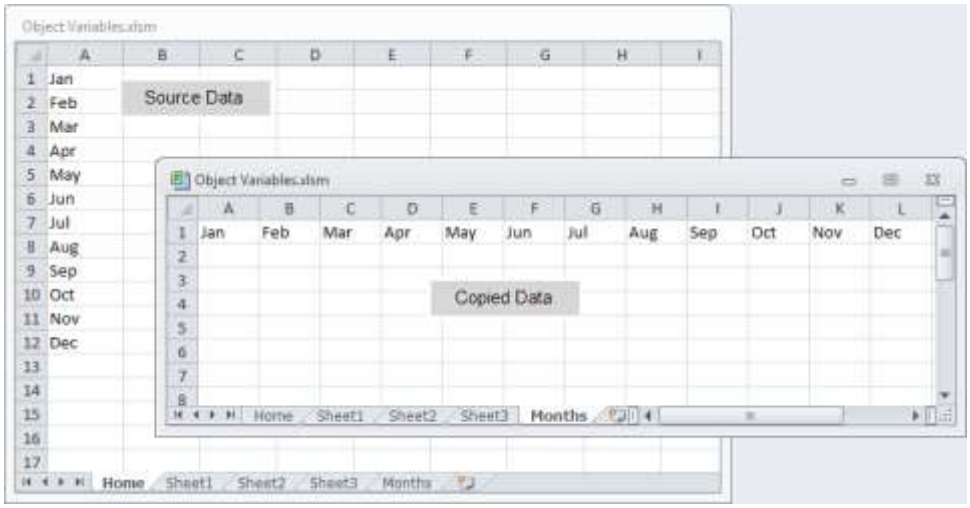
In the last example, we will see another way to duplicate a vertical range in the first row of another worksheet. Of course it is not something that you would use in the real world but again the purpose is to review 3 things:

- Use object variables
- Work with data *without* moving the active cell
- The code will write to a worksheet that is *not* the active worksheet

As you can see on the next page the entries in the Home tab (the Source Data) are duplicated in the Months worksheet (the Copied Data) but horizontally. We want to do this without moving the active cell and without going to the Month worksheet!

We need two object variables pointing to both sheets, one for the output range in the Month worksheet and one to store how many entries are in column A in the Home sheet. Please create a workbook similar to the one the next page. There can be any number of sheets as long as the sheets **Home** and **Months** exist.

Create a list on months in column A of the Home worksheet. There can be any number of consecutive entries in column A.



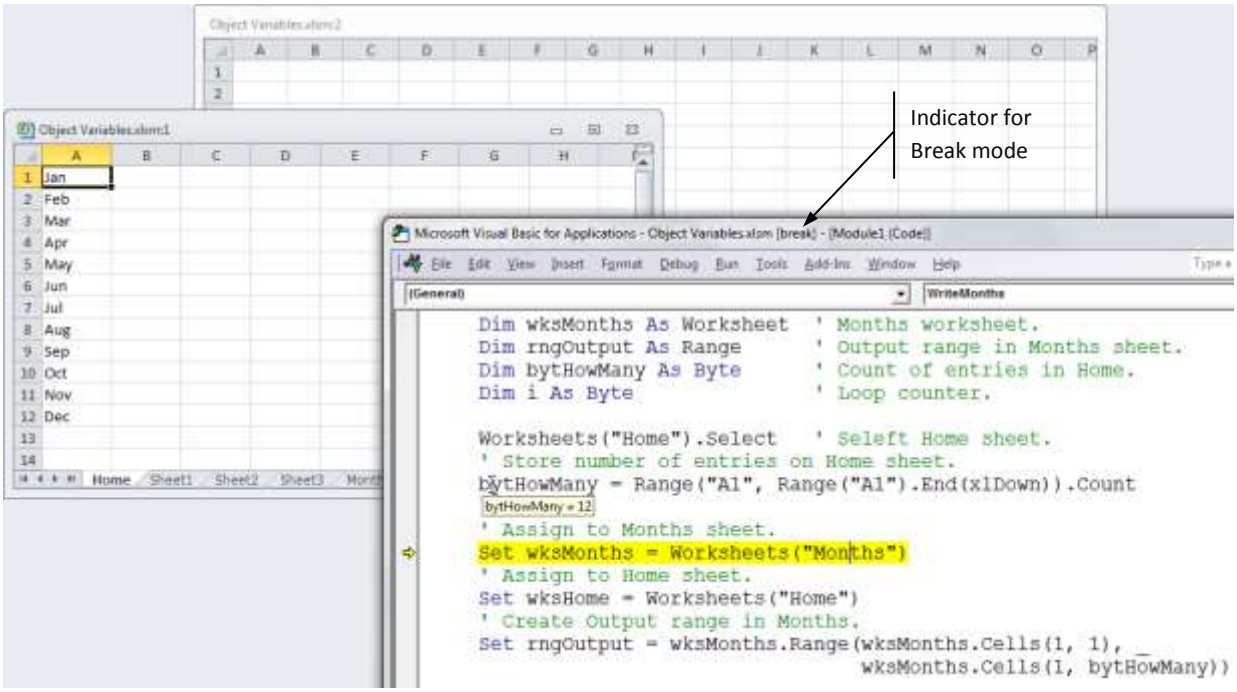
In a module type the following Sub. You can ignore the comments if you want.

```
Sub WriteMonths()
' Write the months in the Home sheet into the Month worksheet.

    Dim wksHome As Worksheet     ' Home worksheet.
    Dim wksMonths As Worksheet   ' Months worksheet.
    Dim rngOutput As Range       ' Output range in Months sheet.
    Dim bytHowMany As Byte       ' Count of entries in Home.
    Dim i As Byte                ' Loop counter.

    Worksheets("Home").Select    ' Select Home sheet.
    ' Store number of entries in column A of the Home sheet.
    bytHowMany = Range("A1", Range("A1").End(xlDown)).Count

    ' Assign to Months sheet.
    Set wksMonths = Worksheets("Months")
    ' Assign to Home sheet.
    Set wksHome = Worksheets("Home")
    ' Create Output range in Months.
    Set rngOutput = wksMonths.Range(wksMonths.Cells(1, 1), _
                                    wksMonths.Cells(1, bytHowMany))

    ' Loop through all entries in Home sheet and write them in Months output range.
    For i = 1 To bytHowMany
        rngOutput.Cells(1, i) = wksHome.Cells(i, 1)
    Next i
End Sub
```

Notice the variable bytHowMany? It is assigned to the count of cells in the range starting from cell A1 to the bottom of the selection. This expression was discussed on page 6. Knowing the number of entries will allow us to control the loop at the end of the Sub.

Once done click anywhere in the procedure the hit the F8 key a couple of times to Step In the code. When you hit the F8 key your procedure is said to be in *break mode*. Once you passed the line where the variable bytHowMany is assigned the number of entries in column A, move your mouse over the variable bytHowMany to see its current value.

Things would look clearer if you can arrange your screen as shown below.



Note in the screenshot above that in Break mode you can move the mouse pointer over a data variable and see its value at that moment!

If you want to see the actual range assigned to a range variable, move in the Immediate Window (Ctrl+G) and type the name of the range variable followed with `.Select` to select it in Excel. For example, one you have passed the line starting with `Set rngOutput` move to the Months worksheet and type the following in the Immediate Window:

```
rngOutput.Select
```

The first few cells in row 1 of that worksheet will be highlighted depending on how many entries are in column A. Continue to hit the F8 key until you reached to end of the Sub.

In the For loop you note that each cell belonging to the horizontal Output range gets the value of the corresponding entry in column A of the Home sheet.

Using a For Each – Next Loop

Another popular technique to cycle through a range of cells is the For Each – Next loop. It is well discussed in the *About Loop* document available on the Web site. We're not saying that the Cells property is not as good as a For Each – Next loop discussed below, it's just a different approach.

To use that type of loop we need two object variables: The range itself and the individual cells in the range. Note that this type of loop *only* works with object variables and arrays, not data variables.

Let us look at the sample code below:

```
Dim rngValues As Range      ' A selection of cells in a sheet.
Dim rng As Range            ' Individual cells in the selection.
```

Then in your code you would use something like the following:

```
For Each rng in rngValues
    ' Do something with individual rng
Next rng
```

The advantage of this technique is you don't have to worry about incrementing any value since rng will take on each value in the selected range automatically and do whatever it is told to do in the loop. It is also worth mentionning that the loop will automatically stop once it reaches the last entry.

Let redo the exercise on the top page 8 using a For – Next loop.

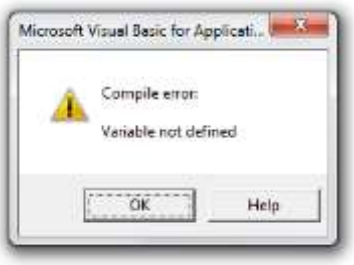
```
Sub WriteMonths2()
' Write the months in the Home sheet into the Month worksheet.

    Dim wksMonths As Worksheet  ' Months output worksheet.
    Dim rngInput As Range       ' Input range in Home sheet.
    Dim rng As Range            ' Individual cells in selection.
    Dim i As Byte               ' Controls which cell in used in sheet Months.

    Worksheets("Home").Select   ' Select Home sheet.
    ' Store the range in Home sheet in variable rng.
    Set rngInput = Range("A1", Range("A1").End(xlDown))

    ' Assign to Months sheet.
    Set wksMonths = Worksheets("Months")

    ' Loop through all entries in Home and write them in Months.
    For Each rng In rngInput
        ' The arguments for Offset are Offset(rows, cols)
        wksMonths.Range("A1").Offset(0, i) = rng
        i = i + 1
    Next rng
End Sub
```

Using a For Each – Next loop is a very efficient of going through each selected cell in selections of any size and shape.

About Option Explicit

To finish this article let just briefly discuss the statement Option Explicit at the top or every module. I have heard from some developers that this option is not strictly necessary. Well, maybe that is why it is an … option.

On the other hand I am convinced that is very important. Every module should have this option at the top and let it do its contribution to our coding.



The term *explicit* simply means that every variable must be (explicitly) declared in a procedure (Sub or Function) before it is used. If a variable is not declared, you get a compile error specifying that the highlighted word is a variable that was not declared.

In real life though, it most likely means that you've made a typo. In my humble opinion, any feature that alerts me that I have made a mistake in typing a variable is welcome.

You don't even have to type Option Explicit in the *declaration section* of your module (in other words at the top). There is an option in the VBE (Visual Basic Editor) that is disabled by default and you simply need to enable it once and for all.

Follow these steps:

- In the VBE choose **Options** from the **Tools** menu.
- Enable the second option: **Require Variable Declaration** and click OK

You will never have to worry that a typo in a variable's name makes a mess in your code.

Conclusion

It is obvious that variables are essential in pretty much all pieces of code you will write. Using them wisely and efficiently will make your code run smoother. Please remember also to *document* you code using comments as shown in the last few examples above.

There are many Web pages discussing this topic. Simply search for "Excel VBA variable examples" or "Excel VBA object variables"

Daniel Lamarche
ComboProjects.com.au