VBA Cheat-Sheet and Tutorial

Charlie Nuttelman

University of Colorado at Boulder

VBA References to Information in Excel

VBA refers to information in Excel in several ways. These include the Range, Selection, and ActiveCell object, the Cells property, and the Offset property. Column and Rows can also be utilized. Here are some examples:

- `Range("D22")` → refers to cell D22
- `Range("A1:B5")` → refers to the range A1:B5
- `Range("Pressure")` → refers to the named range "Pressure"
- `Range("D:D")` → refers to entire column D
- `Range("3:3")` → refers to entire row 3
- `Range("A1:B5").Cells(2,2)` → refers to the 2nd row, 2nd column of range A1:B5
- `Range("A1:B5").Range("B2")` → also refers to the 2nd row, 2nd column of range A1:B5
- `Range("D22").Offset(3,1)` → refers to the cell that is 3 rows down, 1 column to the right of cell D22
- `Cells(4,5)` → refers to the 4th row, 5th column, or simply cell E4
- `Columns("D:F")` → refers to columns D, E, and F
- `Columns(2)` → refers to column B
- `Rows(2)` → refers to row 2
- `Rows("2:3")` → refers to rows 2 and 3
- `Selection.Cells(3,4)` → refers to the cell that is in row 3, column 4 of the current selection
- `ActiveCell` → refers to the active cell
- `ActiveCell.Offset(-1,1)` → refers to the cell that is 1 row up and one column to the right of the active cell
- `ActiveCell.Offset(-1,1).Activate` → activates the cell that is 1 row up and one column to the right of the active cell. Note that .Select and .Activate should be avoided, there are normally other ways to place information into cells rather than selecting or activating them first.

Moving Data Between Excel & VBA

There are several ways to move data to and from VBA/Excel.

<u>Importing data from Excel:</u> Let's say we wanted to define a local variable to VBA, x, in terms of values in an Excel spreadsheet. There are a couple ways we can do this and the "VBA References to Information in Excel" examples (above) can oftentimes be implemented to do this. Some examples:

- `x = Cells(2,2).Value` → assigns the value in **B2** to x (note that .Value is optional)
- `x = ActiveCell.Value` → assigns the value in the active cell to **x**
- `x = Range("A1:B5").Cells(2,2)` → this would assign to variable **x** the value in the 2nd row, 2nd column of range A1:B5
- `x = Range("A1:B5").Value` → this would assign to variable **x** the range **A1:B5**; NOTE: in this case, **x** must be declared as a variant type since **x** is an array here

Exporting data to Excel: Moving data from VBA to Excel is essentially the opposite of importing data. Some examples assuming we have already defined a local VBA variable **y**:

- `Cells(2,2).Value = y` → places the value **y** into **B2**
- `ActiveCell.Value = y` → places **y** into the active cell
- `Range("A1:B5").Cells(2,2) = y` → places **y** into the 2nd row, 2nd column of range **A1:B5**

Other ways to modify the contents of cells: Here are some other examples in which we can utilize VBA to modify the contents of cells:

- Range("A1:B5").Value = 23 → places 23 in each cell of range A1:B5
- Selection = 10 → places a 10 in each cell of the current selection
- Range("D22") = 2 * a → multiplies local variable a by 2 and places in cell D22

For Each: The **For Each** statement in VBA is a nice way to quickly make a change to the contents of an array in Excel. The first example below will add 5 to each value in the range **A1:C3**. The second example will change the value in a selection to 9999 if and only if it is equal to 5. This is a nice way to enable conditional statements in an array. Importantly, c must be declared as a variant type.

```
Sub ForEach()
For Each c In Worksheets("Sheet1").Range("A1:C3")
c.Value = c.Value + 5
Next c
End Sub

Sub ForEachIf()
For Each c In Selection
    If c.Value = 5 Then
        c.Value = 9999
    End If
Next c
End Sub
```

The second sub here will make the following transformation:

2	7	4
8	5	6
3	9	5

2	7	4
8	9999	6
3	9	9999

Importing Arrays from Excel Into VBA – Fixed size & place

Oftentimes it is necessary to import data in an array format from Excel into VBA. Maybe you want to bring in the data and do a calculation on it, etc. When the size and spreadsheet location are fixed (e.g., cells **C4:F8** are always used and size nor location never change), you should use one of the approaches outlined here.

Direct Import of an Array: Arrays can be directly imported into VBA from Excel using Range objects. Importantly, the variable array in VBA must be declared as a variant type. In the following example, if we wanted to import **A1:C3** into VBA, the variable **a** would have to be declared as a variant.

```
Sub GetArray1()
a = Range("A1:C3").Value
End Sub
```

Also, this approach is not tremendously useful because in order to do calculation on each component of array a we could first need to extract each component of the array and re-define as a single or double type. For example, the following does NOT WORK and we cannot just create a new array b by multiplying array a by 2:

```
Sub GetArray1()
a = Range("A1:C3").Value
b = 2 * a
End Sub
```

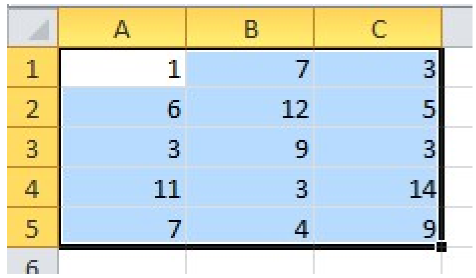
(This gives us a type mismatch error)

Nevertheless, if we are interested in using single values of an imported array, the method described above works well. For example, if we wanted to extract the value in cell (2,2) (row 2, column 2 of array a) then we could easily do this and assign it to value c:

```
Sub GetArray1()
a = Range("A1:C3").Value
c = a(2, 2)
End Sub
```

We can also use other references including but not limited to **Selection** (current selection) and **Range("Temp")** [if Temp is a named array in Excel, for example], and include the **Offset** property if we wish.

Iteration Import: Nested **For… Next** loops can be used nicely to import arrays from Excel and be stored as local arrays in VBA. Please see the ***For… Next Loops*** section later in this tutorial for more information about count-controlled iteration. There are different ways to do this, but one way is shown here. Let's say we have an array in cells **A1:C5** in Excel:

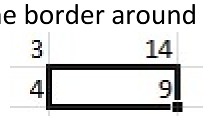


We wish to bring this selection into VBA for a particular use or transformation. Note that this region is *selected* and is not a fixed size region. The first step is to **Dim** our array A in VBA (**Dim A()**). At this point, the parentheses are empty because the size is unknown. Next, we count the rows and count the columns of our selection and **ReDim** our array (**ReDim A(nRows, nCols)**, for example). Finally, we set up a **For… Next** loop within a **For… Next** loop; the outer loop iterates through the rows and the inner loop iterates through the columns. Here is the code in VBA:

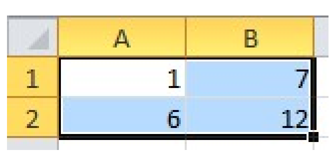
```
Dim A() as Double
Dim j as Integer, k as Integer, nRows as Integer, nCols as Integer
nRows = Selection.Rows.Count
nCols = Selection.Columns.Count
ReDim A(nRows, nCols) as Double
For j = 1 to nRows
    For k = 1 to nCols
        A(j, k) = Selection.Cells(j,k).Value
    Next k
Next j
```

Changing the Active Cell

The active cell in Excel is the cell that has the border around it. Here, the cell with 9 in it is the active cell:



A selection also has an active cell. By default, the active cell is the upper left cell (but this can be changed). In this example, the cell with 1 in it is the active cell of the selection:



What if we want to change the active cell? We use the **ActiveCell.Offset(<rows>, <columns>).Select** command. This will select as the new active cell the cell that is **<rows>** number of rows and **<columns>** number of columns away from the previous active cell.

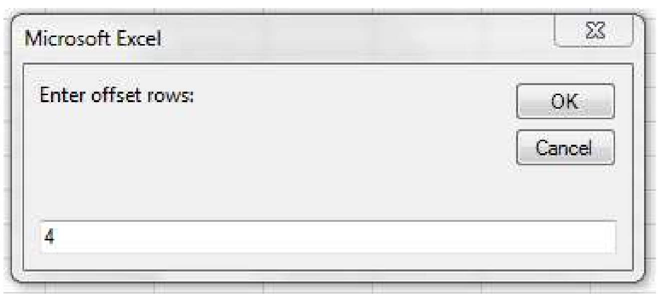
If I start with:

	A	B	C
1	1	7	3
2	6	12	5
3	3	9	3
4	11	3	14
5	7	4	9

and run the VBA code here:

```
Sub MoveActiveCell()
  Dim Rows As Integer, Cols As Integer
  Rows = InputBox("Enter offset rows: ")
  Cols = InputBox("Enter offset columns: ")
  ActiveCell.Offset(Rows, Cols).Select
End Sub
```

it will ask me for two inputs:



then the result of my input above is:

	A	B	C
1	1	7	3
2	6	12	5
3	3	9	3
4	11	3	14
5	7	4	9

You can also use **Activate** to change the active cell. The main difference between **Selection** and **ActiveCell** is that **Selection** can refer to a highlighted/selected range of cells and **ActiveCell** refers to the single active cell, which can be part of a selection.

This example will also work for activating a new cell:

```
ActiveCell.Offset(2,1).Activate
```

The cell that is 2 rows down and 1 column to the right will now be the active cell.

Note that oftentimes it is unnecessary to activate or select a new active cell and these procedures can slow down subroutines. For example, to place the value of 5 in a cell that is down 2 rows and over 2 columns, one might use:

```
ActiveCell.Offset(2,2).Activate
ActiveCell = 5
```

It is much faster and more efficient just to use:

```
ActiveCell.Offset(2,2) = 5
```

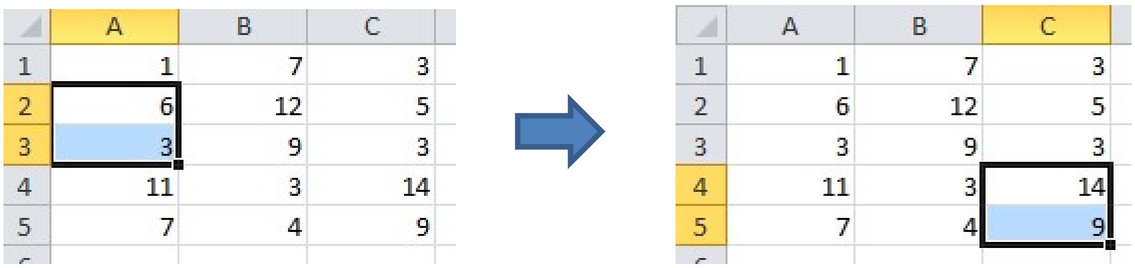
Offsetting the Selection

What if I have a selection (a bunch of cells in Excel) and I want to offset the selection? We can use the **Selection.Offset(<rows>, <columns>).Select** command:

If I implement the following code in VBA:

```
Sub MoveSelection()
  Dim Rows As Integer, Cols As Integer
  Rows = InputBox("Enter offset rows: ")
  Cols = InputBox("Enter offset columns: ")
  Selection.Offset(Rows, Cols).Select
End Sub
```

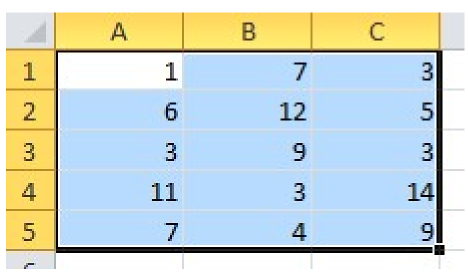
and enter into the two input boxes 2 and 2, the following change is made:



As above, it is oftentimes unnecessary to select different cells, depending upon what you are trying to do, and the .Select command can slow things down.

Exporting Arrays from VBA to Excel

After we create or modify an array in VBA, it is nice to display this array in Excel. Let's use the matrix A from "Importing Arrays from Excel Into VBA" (see above) and assume we have imported it correctly. We now want to add 5 to each element of A and then export back to Excel. (For an easy way to do this without iteration, please see the ***For Each*** section above.) Note again that we are using a *selection* in Excel, not a range or named array. We are again going to use a **For… Next** loop within a **For… Next** loop. Please see the ***For… Next Loops*** section later in this tutorial for more information about count-controlled iteration. In the inner loop, we will add 5 to each element of A and we will export that value to the spreadsheet. There are several ways to do this, but a suitable method is explained here. The selection we start with is the same as above:



First, we need to offset the active cell. We wish to start this in, say, cell E1. To offset the active cell to begin exporting from VBA back to Excel, we need to offset by 0 rows and 4 (3 + 1) columns in this case. In general, we will offset by (nCols + 1) columns. The code for this is shown here (don't forget to **Dim** m and n!):

```
ActiveCell.Offset(0, nCols + 1).Select
For m = 1 to nRows
  For n = 1 to nCols
    A(m, n) = A(m, n) + 5
    Selection.Cells(m, n).Value = A(m, n)
  Next n
Next m
```

When we run the code (added onto the code above for importing the array elements), we end up with the following output in Excel in columns E through G:

E	F	G
6	12	8
11	17	10
8	14	8
16	8	19
12	9	14

Programming Structures in VBA

One-Way If ("If-Then")

There are several different structures that use the **If… Then** statements. First, there is the one-way **If**. This is quite simple. If a condition is met, then you do something; otherwise, you don't do anything:

```
If y  > 0 Then
    func = 5
End If
```

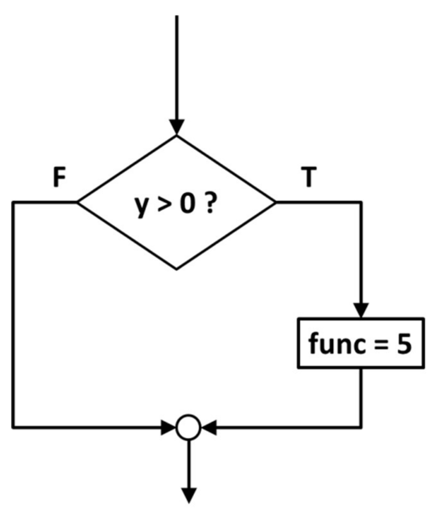
Note that if y is less than zero we don't do anything. This is why it is known as a *one-way* **If**.

Also note that we could also write these one-way **If** statements as one-liners; if we do this, we don't need an "**End If**" at the end:

```
If y > 0 Then func = 5
```

This is equivalent to the 3 lines of code above.

A flowchart for this process is shown here:



Two-Way If ("If-Then-Else")

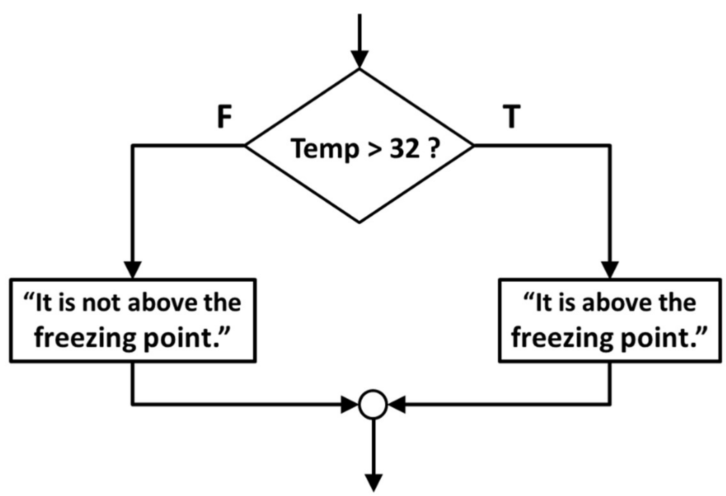
A two-way **If** is similar to the one-way **If** but does something if the first condition is not met. In other words, if the condition is met then one thing happens (as in the one-way **If** above) but if that condition is not met then something else happens – and this something else will always happen if the original condition is not met (in contrast, if there is an **ElseIf** statement then that something else that happens is contingent on satisfying a secondary condition). To indicate the two-way nature and to indicate what should occur if the condition is not met, the **Else** statement is included.

Here is an example:

```
If Temp > 32 Then
    MsgBox ("It is above the freezing point.")
Else
    MsgBox("It is not above the freezing point.")
End If
```

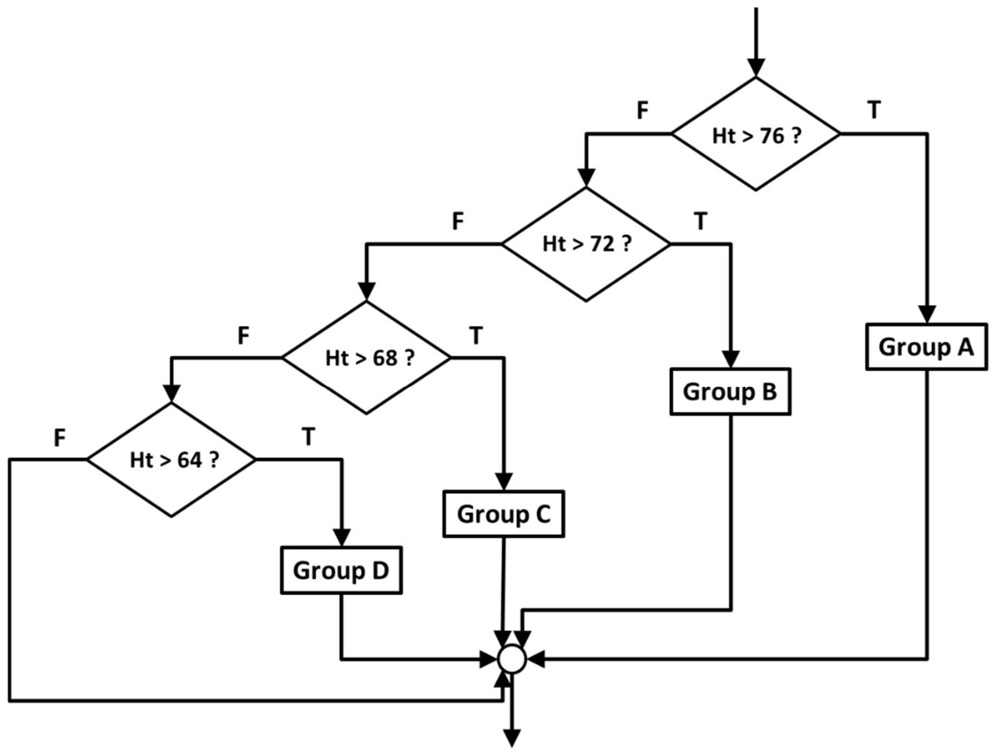
Note that this is a TWO-WAY **If** because now something happens if the conditional is true (**Temp > 32**) and something *also* happens if the conditional is false. This is NOT a multi-alternative **If** because there is only one conditional statement (**Temp > 32**). The **Else** element is what I like to refer to as a "catch-all" (i.e., if the conditional is not met, then whatever is in the **Else** part is carried out).

This is a flowchart for the two-way **If** statement example above:



Multi-Alternative If ("If-Then-ElseIf")

I like to think of multi-alternative **If** statements as selective "parsing". You basically start with *all* available options (for example, a range for a variable x of $-\infty \leq x \leq \infty$) and then you parse off different options from there. It is like starting with all the students in the class and asking "Everyone who is taller than 6'4" please write an 'A' on your forehead", and then they step to the front of the room (already assigned). Then you ask "Everyone who is taller than 6'0" please write a 'B' on your forehead", and they step out of the way. Notice that in this second group, you don't have to ask for who is shorter than 6'4" because they've already been parsed out in the first group. So, we could set up the following flow chart for this process (height in inches), separating the class into a bunch of different height categories:



This would be coded in VBA as follows:

```
If Ht > 76 Then
    Group = "A"
ElseIf Ht > 72 Then
    Group = "B"
ElseIf Ht > 68 Then
    Group = "C"
ElseIf Ht > 64 Then
    Group = "D"
End If
```

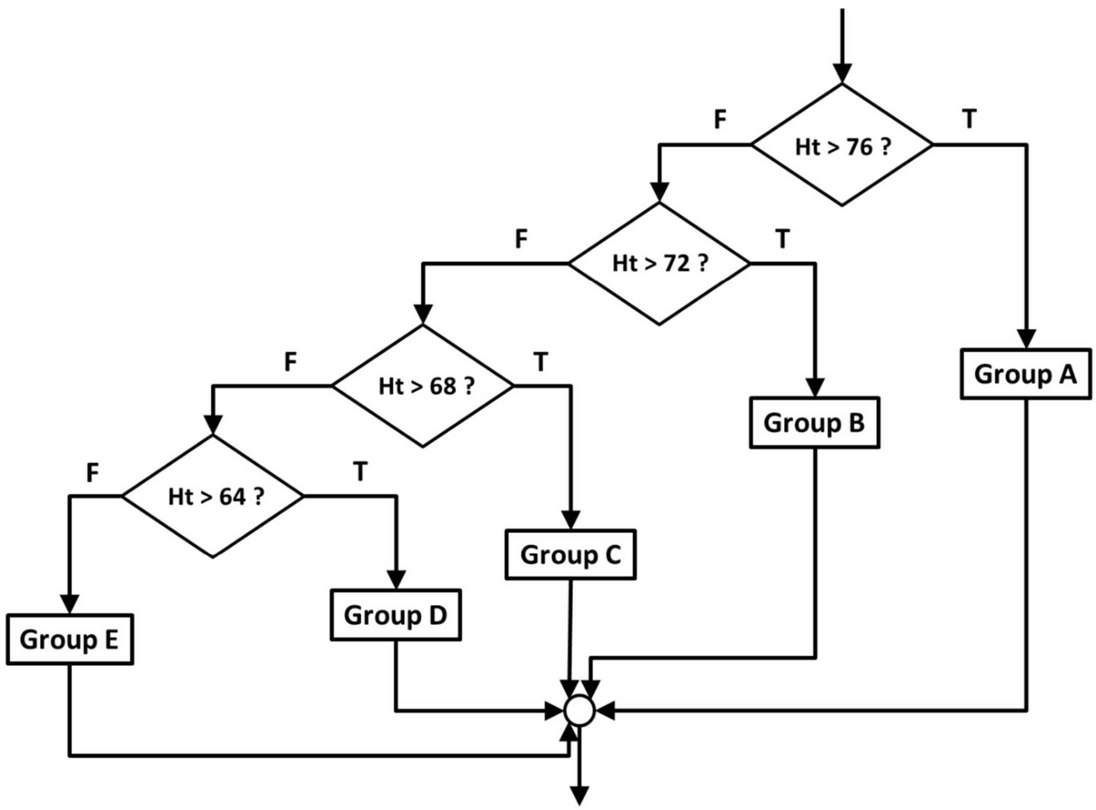
This would assign a **group** corresponding to an entered **Ht**. But what if you are shorter than 64"?! Nothing happens, you don't get a group (and letter on your forehead!). The **Else** and **ElseIf** statements are all conditional statements. That is, in the example above **Ht** must be greater than a certain value in order for group to be assigned.

In order to add a "catch-all" to the code above, we would add in the **Else** statement, like in the **If-Then-Else** above. The **Else** statement is optional – without it we don't do anything if none of the conditions in the **ElseIf** lines are met; with it we do something else. In other words, if none of the other conditions are met, then we do *something* (in this case, we assign students shorter than 64" to Group E).

Here is the VBA code with the “catch-all” **Else** statement:

```
If Ht > 76 Then
    Group = "A"
ElseIf Ht > 72 Then
    Group = "B"
ElseIf Ht > 68 Then
    Group = "C"
ElseIf Ht > 64 Then
    Group = "D"
Else
    Group = "E"
End If
```

And the corresponding flowchart is shown here:



GoTo …

When all else fails and you can’t seem to get your selection scheme down right, consider the **GoTo** statement. This is essentially a “choose your own adventure” or “beam me up, Scotty” procedure. An example of the **GoTo** is shown on the next page:

```
Sub GoToExample()
Dim x, JumpOut As String
x = InputBox("Please enter your name:")
If x <> "Charlie" Then GoTo JumpOut
    MsgBox ("You have the same name as the instructor!")
    Exit Sub
JumpOut:
    MsgBox ("You do not have the same name as the instructor.")
End Sub
```

The sub asks the user for his/her name. If the name is not equal to Charlie (not equal to in VBA is "<>") then the **GoTo** statement warps to the **JumpOut** spot, bypassing the first message box. Upon continuing at the **JumpOut** "bookmark", a second message box appears if the user's name is not Charlie. You will also not that both x and **JumpOut** are declared as strings and the **Exit Sub** statement will bypass the second message box if the user's name is Charlie. Oftentimes, the **GoTo** statement is combined with **On Error**. If an error occurs in the sub, the **On Error** (placed at the beginning of the sub) combined with **GoTo** will allow for error protection. An example is shown here:

```
Sub SquareRoot()
Dim x As Single, Bookmark As String
On Error GoTo Bookmark
x = InputBox("Please enter a number, I will take the square root")
MsgBox ("The square root of your number is: ") & Sqr(x)
Exit Sub
Bookmark:
MsgBox ("An error occurred - number you entered was negative, please
try again")
End Sub
```

If the user inputs a negative number, an error will occur on the 5th line (can't take the square root of a negative number). If this happens (error), then the code jumps to Bookmark. If the number is positive (or zero) then the message box displays the square root of the number. The subsequent Exit Sub statement then allows bypassing of the second error message box.

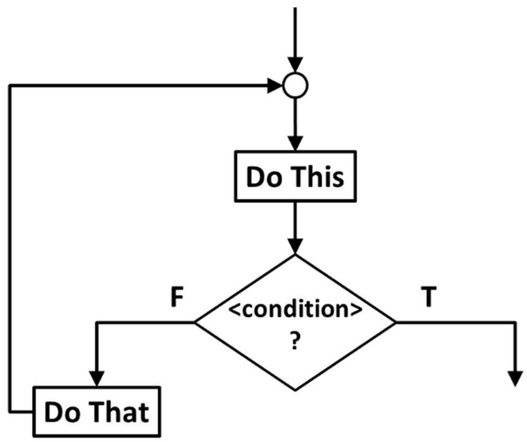
Repetition Structures

Do loops are performed when you want to keep iterating until a condition is met. For example, you are at a bar and you want the phone number of a particular girl (or boy). You want to keep asking her/him for her/his number until she gives it to you. You will not leave until the condition is met (this is similar to input validation that we have talked about – the program will not continue unless the input is a number, for example). Just like you and your persistence, the program won't take "NO" for an answer (by the way, I don't recommend this approach while at the bar). The loop keeps going unless some condition is met. You will ask over and over and over until she/he gives you her/his phone number. This could be all night (100 or more times!) or on the first try. **For.. Next** loops are performed when you want to iterate/repeat something a *fixed* number of times. Applied to the example above, you are going to ask the girl/boy for her/his phone number *exactly* 10 times. If she/he gives you her/his number after try 1, you will still ask her/him another 9 times – the number of iterations is fixed.

General Do Loop

Do loops must be used with caution. If there is no way to exit the **Do** loop, then the program will be caught in an eternal loop with no way to get out - a dog chasing his or her tail forever.

In a general **Do** loop, there must be a way to *exit* the loop. The loop is exited when a condition (**If <condition> Then Exit Do**) is met. For example, when a tolerance value of 0.01 is met then the loop is done. Or the loop is exited once a number is input (keeps looping until a number is input). A general **Do** loop looks something like this:



and the coding for this in VBA would be:

```
Do
    <Do This>
    If <condition> Then
        Exit Do
    Else
        <Do That>
    End If
Loop
```

Note that **<Do This>** and **<Do That>** are just blocks that correspond to "pre-test" (something you do before the test of **<condition>**) and "post-test" (something you do after the test of **<condition>**) coding. **<Do This>** and **<Do That>** are optional.

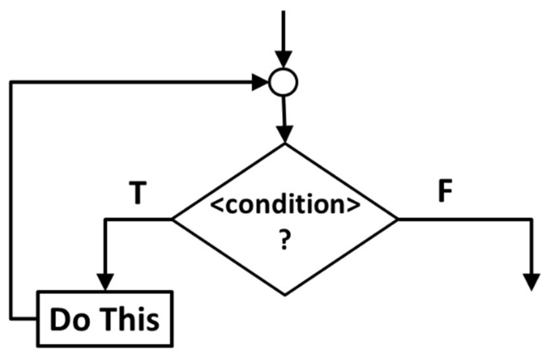
An example of a general **Do** loop is shown here:

```
x = 20
Do
    If x < 1 Then Exit Do
    x = x / 2
Loop
```

As the code progresses, the value of x will change (20, 10, 5, 2.5, 1.25, 0.625) and since 0.625 is less than 1 it will exit the loop after 5 iterations. Note in this example that there is no pre-test code but there is code before the loop (just not within the loop). Also note that since there is no **ElseIf** we can use a one-liner for the **If** command. Finally, the post-test code basically divides the old value of x by 2 before doing the loop again. The loop will continue to halve the value of x until it is less than 0.1; at this point, the loop will be exited. What would happen if the "x = x / 2" line was left out? That's right, there would be an eternal loop because there is no way to get out (x would never ever be less than 0.1 since it would forever be 20 – no way to change the value of x – the **Exit Do** command would never execute).

Do While Loop

The **Do While** loop is set up to do something while a condition is true. This is also known as a "pre-test" loop because the test is done at the beginning of the loop. In fact, if the condition is met going into the loop then the loop won't even execute a single time! The **Do While** loop is a special case of the general **Do** loop and in fact just has the **If** statement essentially built into the first line of code (in other words, the **If… End If** are wrapped into the **Do While** statement, basically to simplify things because including the **If** statement in a **Do** loop is so common). It looks something like this in flowchart form (compare to the general **Do** loop – the same except there is no pre-test code):



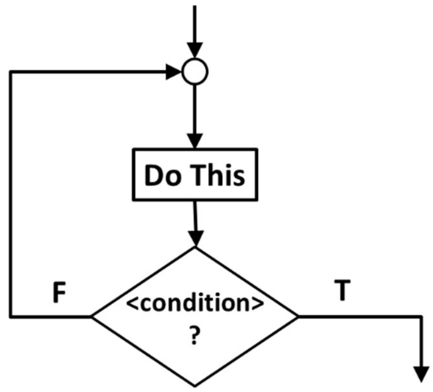
Note that the loop is continued on **True** and exited on **False**, which is different than the general **Do** loop. The code will do something while the condition is true. Adapting our previous general **Do** loop example, this is what we might code in VBA:

```
x = 20
Do While x > 1
    x = x / 2
Loop
```

The VBA will loop through this and x will change (20, 10, 5, 2.5, 1.25, 0.625 – 5 times) and the value of x will end up as 0.625. There is no **Exit Do** and there is no need to include **If… Then** statements since these are inherent in the **Do While** statement. In the flowchart above, **<condition>** refers to "is x > 20 ?" and "**Do This**" refers to "x = x / 2".

Do… Until Loop

Another special case of the general **Do** loop, the **Do… Until** inherently includes an **If… Then** statement in its closing line. **Do… Until** loops look like the following:



and they begin with the first line of **Do** and close with the last line **Loop Until <condition>**. In these regards, the **Do… Until** loop is referred to as a "post-test" loop since the first test is done at the completion of a loop. **Do… Until** loops will *always* run at least once. Like the general **Do** loop (different from the **Do While** loop), they are exited on a condition being true.

The VBA code for our example we've been following might look like:

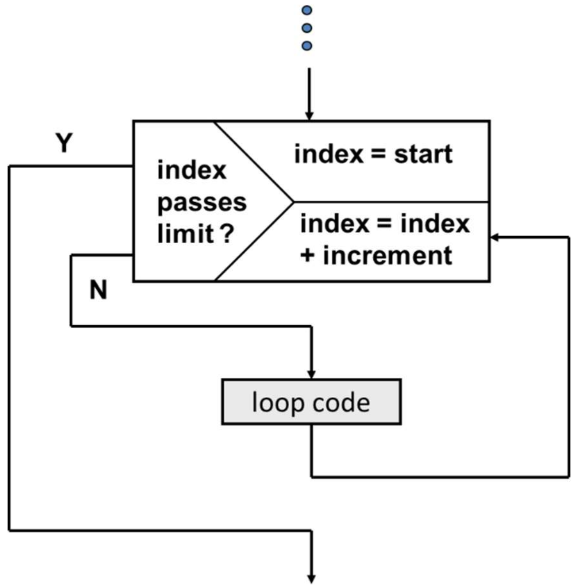
```
x = 20
Do
    x = x / 2
Loop Until x < 1
```

How many times will this loop execute? In the first iterations, x will exit as 10. In the second iteration, x will exit as 5. Then 2.5 in the 3rd iteration, 1.25 in the 4th iteration, 0.625 in the 5th and final iteration.

For… Next Loops (Count-Controlled Iteration)

For… Next loops are used when you have a *fixed* number of iterations (or you might determine the number of iterations from the user or elsewhere, but once you enter the **For… Next** loop the number of iterations is known and is fixed). **For… Next** loops are generally used to iterate through data in a controlled fashion or when you need to do something a specified number of times.

There are several ways to set up the flowchart element for the **For… Next** loop but the most popular is shown below:



We enter the **For… Next** loop through the top of the box. At this point, the index (the number we are iterating through) is set to its start value. Next, we migrate to the left part of the box and there is an embedded decision made here – is the current value of the index greater than our limit? If so, we are done with the **For… Next** loop. If not, then we move along to the **loop code**. Oftentimes, the **loop code** will do something that involves the current index value. After the **loop code**, we go back to the lower right corner of the box and increment our index (increment is commonly 1 but can be whatever you choose). The general syntax for the **For… Next** loop in VBA looks like:

```
For <index> = <start> to <limit> {Step <increment>}
    <loop code>
Next <index>
```

An example might be

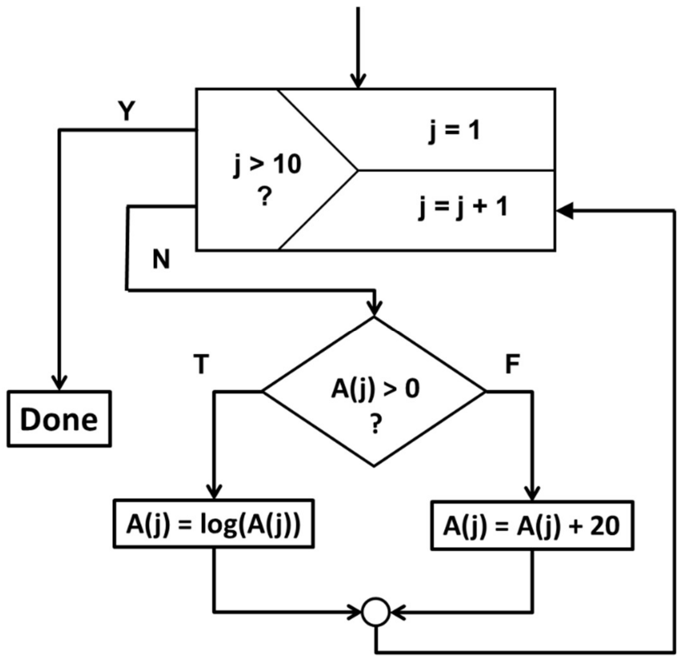
```
For j = 1 to 10
   Sum = Sum + j
Next j
```

Or another example might be (can you figure out what this might do?):

```
For k = 18 to 0 Step -2
    A((20 – k) / 2) = k
Next k
```

where we are iterating through the index k starting with 18 and going down by 2 (k = 18, 16, 14, … , 0).

Shown below is another example. In this example, we are iterating through the index j from 1 to 10 and during each iteration we are taking the natural log of the number if it is positive and we are adding 20 to each number if it is negative or equal to zero:



Note that in each iteration, we have a decision (two-way **If**) to make. This will often be the case. The VBA code for this example is:

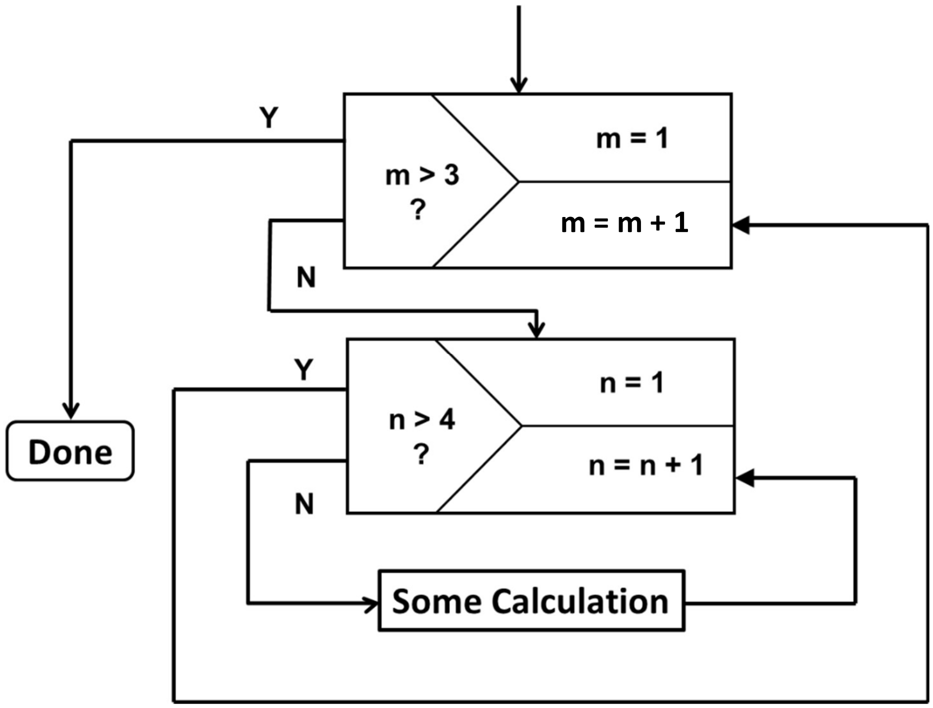
```
For j = 1 to 10
    If A(j) > 0 Then
        A(j) = Log(A(j))
    Else
        A(j) = A(j) + 20
    End If
Next j
```

Nested For… Next Loops

Oftentimes it is important to iterate through a 2-dimensional array (for example, an array with 3 rows and 4 columns). This is done by embedding (or nesting) **For… Next** loops. In the following example, the **m** index (outer loop) refers to the row index and the **n** index (inner loop) refers to the column index. Here, we are iterating through each column of Row 1, then each column of Row 2, etc. We are performing some calculation for each of these array elements of array A.

```
For m = 1 to 3
    For n = 1 to 4
        <Some calculation>
    Next n
Next m
```

A flowchart for this process looks something like this:



Exit For

Sometimes we wish to iterate through an array or vector, for example, until a certain value is found or constraint is met and we want to stop iteration once that condition is met. We can use the **Exit For** in conjunction with an **If… Then** statement to limit the amount of iteration that is performed. As a simple example, let's combine this with the **For Each** statement and we can search through an array until the first incidence of a constraint is met. We'll illustrate this with one of the **For Each** examples shown earlier on p. 2. You will see in the code below that we have added just a single line of code. We've added **Exit For** inside the **If…Then** statement. Once the value in a cell in the selection is equal to 5, we set the value of that cell to 9999 and then we exit the **For** loop. We do not continue and you will notice that when we run this, only the first 5 is replaced with a 9999.

```
Sub ForEachIf()
For Each c In Selection
    If c.Value = 5 Then
        c.Value = 9999
        Exit For
    End If
Next c
End Sub
```

When this sub is run on a selection, the following occurs:

2	7	4
8	5	6
3	9	5

→

2	7	4
8	9999	6
3	9	5

Hope you find this tutorial helpful!