Towards A Deeper Understanding of Visual Basic for Applications

Draft, S. Malpezzi, April 26, 1999

Introduction

Visual Basic for Applications (VBA) is a computer language. Like any language, English, French, German, or Korean, there is a lot to learn at the start. Like any other language, there is a grammar, and vocabulary.

When learning languages, conversational or computer, there are two general approaches one can take. One is to jump in and immerse one's self in conversation, or in writing a program. In time, how the pieces fit together will become clear through example. The second approach is to focus on a formal study of the structure of the language, the grammar, as well as learning vocabulary.

So far in this course we have taken the former approach. Now let's take the second approach, and examine what we are up to a little more formally.

The Grammar of VBA

VBA has three main parts of speech: objects, properties, and methods. There are some others: VBA also has operators, functions, and statements.

One reason to get familiar with the grammar is that the online help, error messages when debugging, and most Microsoft books refer to these concepts endlessly.

I should mention at the outset that some keywords can signify more than one part of speech, depending on context. For example, there is a Name object and a Name property in VBA. Also, be aware that small differences in spelling matter. There's a difference between Row and Rows, for example. With that said, let's examine each of the main types.

Object: Objects are nouns or things. Objects are specified with *references*. Objects can be grouped into *collections*. Objects are often hierarchical.

<u>Example:</u>

```
Workbooks("653_1.XLS").Sheets("Proj_2").Range("B1")
```

The preceding is a hierarchical object, namely a particular cell, in a particular worksheet, in a particular workbook. The objects we use the most include:

Name
Range
Workbook
Worksheet

Property: Properties are attributes of objects; "adjectives." Examples of useful properties include:

ActiveWorkbook
ActiveSheet
ActiveCell
Application
Font
Selection
Value

Method: Methods are verbs or actions to be taken. For example, to change a property, delete an object, add, count, or assign a value to a variable, we would use a method. Useful examples include:

Activate
Add
Clearcontents
Copy
Cut
Delete
Insert
Paste
PasteSpecial
Range
Select
Columns
Rows

<u>Methods</u> often take <u>arguments</u>.

VBA has both *named arguments* vs. *positional arguments*. For an example of positional arguments, consider the syntax of the **Add** method when applied to the Sheets collection (i.e. to create a new worksheet or chart sheet):

Sheets.Add(before,after,count,type), where:

before=where sheet appears (required)

after=where sheet appears (optional)
count=number of sheets to add (optional, default=1)
type=type of sheet to add (i.e. worksheet, chart, etc.; optional, default=worksheet)

Example:
```
     Sub Add1()
          Sheets.Add Sheets.("Sheet2"), , ,xlModule)
     End Sub
```
adds a sheet before Sheet2

Operators: The most commonly used operators are the familiar arithmetic operators, +, -, *, / and ^.

Other useful operators are comparison operators: <, >, =, <=, >=, <>
(the last three are less than or equal to, greater than or equal to, and not equal to).

There are logical operators, and, or, not, that return "true" or "false": Consider the following examples:

```
A = 1
B = 2
C = 3
Test1 = A < B   'Returns True
Test2 = C < B   'Returns False
Test2 = A < B And C < B   'Returns False
Test4 = A < B Or  C < B   'Returns True
```

A less common, but very useful operator, is &, the concatenation operator. Concatenation merges two character strings:

```
Myhero = "Fred" & " Mertz"   'Myhero contains "Fred Mertz"
```

Functions: VBA, like Excel, has some functions that carry out common operations. However, it's easy to get confused, because VBA are logically separate from Excel (application) functions. For example, to take a square root, the Excel function is =sqrt(), while the VBA function is sqr().

Generally, when running VBA and Excel at the same time (what we usually do) both VBA and application functions are available to us. To use an Excel function, specify it using Application.Function. For example, here are two ways to take the square root of 4 in VBA:

```
y = sqr(4)
z = Application.Sqrt(4)
```

Useful functions include:

Abs
InputBox
MsgBox
Int
Rnd
Sqr

Statements: Statements control what VBA does, but don't, strictly speaking, *directly* act on the worksheet. Every VBA program starts with a statement, Sub, and ends with one, End Sub. Statements also include our familiar control structures:

Do While
Do Until
End
For Next
If Then Else

See the handout on "Making Decisions for details. Also, our favorite silly command favorite, Beep, is a statement.

The Visual Basic Editor

There is no getting around it: the Visual Basic Editor can be a pain in the neck. That is why I photocopied introductions to the VBA editor from a couple of other books and placed them on reserve.

If you are using an older version like Excel 5.0, you won't find the Visual Basic Editor as described in those handouts, and as we have used in class. Instead, you will be writing and editing macros in a separate sheet within the workbook. This special "module sheet" in Excel 5.0 looks like a worksheet , except that it does not have any cells. If you find yourself working in Excel 5.0 (or an earlier version), I suggest you consider upgrading to Excel 97. I am not interested selling more software for Microsoft, but there are inevitable problems going back and forth between Excel 5.0 and Excel 97. For most users it is worth it to spend a few dollars, get the new product, and be compatible with the majority of other users, as well as the schools, classroom and lab setups.

Three small but important things to remember in the editor:

(1) Comments are indicated by a single quote:

```
'This is a comment.
```

Use comments freely to outline and annotate your work.

(2) When you want to extend a logical line of code on to two or more physical lines, insert an underscore character at the end of the first physical line:

```
Y = X1 + X2 + X3 _
       + X4 + X5
```

(3) You can run programs from within the VBA editor by choosing "run" from the menu at the top. When debugging, and you're ready to retry after fixing something, hit the "reset" button first.

Help!

When reading through the VBA Language Reference, or looking at the online help, it is important to understand the document conventions.

Words or characters you are expected to type are in **bold**. Bold words with the initial letter **Capitalized** indicate a language specific term: a property, a method, an object name, or another VBA key word. Something in *italics* indicates placeholders for information you supply. If the italics are ***bolded***, these placeholders are for arguments that can be either positional or named arguments. Items [inside of square brackets are optional], (that is, they are optional according to the syntax. You may need the thing in the brackets for the particular problem you are addressing). If you see text inside braces with a vertical bar in between them, which indicates you have to choose between two or more items, for example {this|that}. Small capital letters are used for the name of keys on the keyboard, such are ENTER and CONTROL. A plus sign between key names means hit both at the same time. For example, CTRL+V means to hold down the control key while pressing the V key. Finally, light courier (`the kind that comes out of an old typewriter`) is the font used for example code.

Here are some examples:

object.**Rows**(***index***)

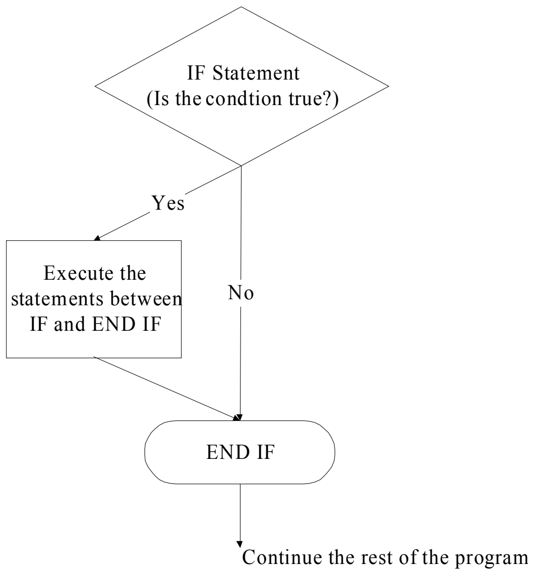
Since *object* is italicized you know that it represents some property that you will supply (for example a worksheet, or a named range). Since **Rows** is bolded and starts with a capital letter we know that that is a VBA key word (in this case the name of the rows method). The word ***index*** tells you that you need to supply some key information, in the case the name of the row. The fact that index is bolded tell you that you can either use a positional or named argument.

Control Structure

Our hand out on “making decisions” explains various methods of controlling program operations. It is worthwhile to step back and think a little more clearly about exactly what control structure or making decisions is all about. Control structure is something that makes decisions about which parts of the program it runs (if at all).

A program without any control structure (and many simple programs don’t have any) simply start at the beginning, and run to the end without stopping. But the reason we love computers is they are so good at making decisions for us, at least simple decision based on things like whether statements are true or false or one number is larger than another. We also love computers because they can repeat steps over and over again that we would find boring and tedious, but they never get tired. Running twenty cases, or twenty thousand cases, through a proforma model is a good example of this.

The Logic of IF/THEN Control

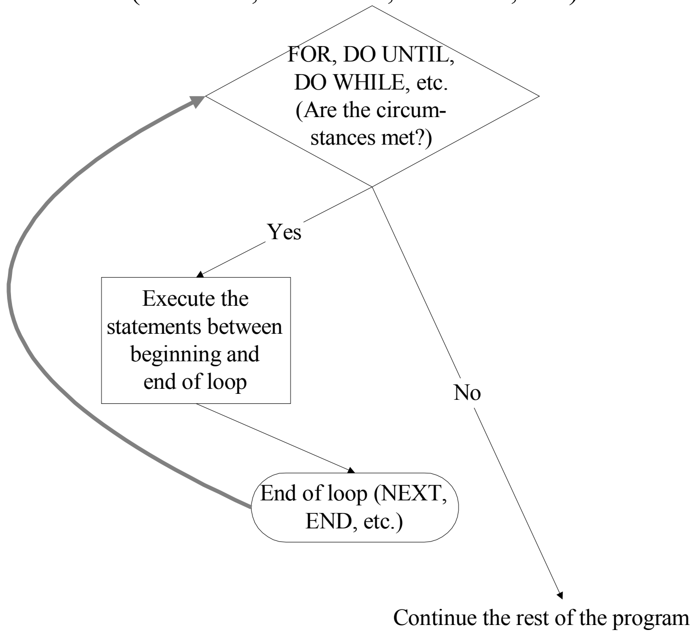

IF Statement
(Is the condtion true?)
Yes
No
Execute the statements between IF and END IF
END IF
Continue the rest of the program

Figures 1 and 2 will help understand how control structures work. The first figure shows how an If/Then statement works. Whenever the program comes to an If/Then statement it evaluates some conditions, such as “is adjusted gross income greater than $100,000?” If the statement is true, then we can calculate, for example, the amount of past losses that current tax laws permit us to deduct. We'd put these statements after the If/Then statement, and signal that they're done with an End statement. If the condition is false we can skip the statements between If/Then and End.

The preceding familiar example highlights the fact that If Then structures are available within formulas in Excel spreadsheets as well as VBA code. The other general kind of program control is available in VBA, but is not generally available as a function in Excel. That is looping, or deciding how many times to repeat a given amount of code.

Looping is accomplished with programming constructs like For-Next, Do While, Do Until, and so on. We have been working extensively with For/Next. The others are variations on the theme, once you understand one, figuring out the others is pretty straightforward. Figure 2 shows how looping works in general. When the program comes to the beginning of the loop, we

The Logic of Looping Control Structures
(For/Next, Do While, Do Until, etc.)


FOR, DO UNTIL, DO WHILE, etc. (Are the circum-stances met?)
Yes
Execute the statements between beginning and end of loop
No
End of loop (NEXT, END, etc.)
Continue the rest of the program

have some control structure like for next or do while that will tell use implicitly how many times to loop. For Next is the simplest kind:

```
For X = 1 to 5
   'Whatever you want to do in here
Next X
```

tells us to loop five times, for example. The corresponding Next statement at the end of the loop, when we add one to the index variable X, is then used to test whether we are ready to move on.

Common Tricks and Traps

Fixed versus Relative Addresses

Don't forget the importance differences between fixed and relative addresses. This is important when you create formulas and spreadsheets, as we all know. It is even more important to get this right when running the macro recorder. With use relative references turned on, operations like Copy and Paste will be recorded as relative addresses (so many cells to the left, so many cells down, and so on). With the use relative references turned off, the action will be generally recorded as taking place in a particular cell. The majority of the time – not always- when we are doing things like copying and pasting we will want to use some sort of relative address. But we have already seen an example of when we wanted an absolute address: every single line of input in our Performa model needs to be copied into the same set of cells that comprise the buffer.

Named Ranges

It is important to be cleared on the advantages of named ranges. The advantages that are so powerful in spreadsheets are even more important in visual basic code. If in our VBA code we refer to specific cell addresses, any time we move things around in the parent spreadsheet we need to go back and fix the VBA code. This is tedious at best, and often the cause of programming errors. If we use named ranges in the spreadsheet, and in VBA, to refer to cells, then generally the movement of positions will be take care of automatically.

Become familiar with the main methods of naming ranges. As we have shown in class, begin by highlighting a range of cells with the mouse (or my favorite, use the F8 key to anchor the cursor and then move the cursor about to select a block of cells. I find using the mouse difficult on any range that is larger than a screen full of data).

Once you have highlighted such a range, go to the name box at the top of the spreadsheet (to the left of the screen but below the menu bars). That will generally contain a cell reference for the top left cell of the range you have highlighted. Click on that cell, and that entry will turn blue. Delete it, and replace it with text comprising the name of the range. Hit enter. You are done.

Notice that naming ranges is a general operation that can be applied to any block of cells. It is often applied to a single cell, or a contiguous block of cells. You can also select and name discontiguous blocks of cells (use the control key in conjunction with the mouse, see your introductory Excel manual for step by step instructions). Once you have marked several blocks of cells you can give them a name in the same way.

Another way to name cells is to use the Insert Name Create command. Generally this saves a lot of time if, as we prefer, we type the names of the cells to the left or above the ranges that *contain* the named ranges. Be very clear: when we name cells things like "XCASES" the cell that contains the name is not the named cell. That cell contains the name, but the named cell is below or to the right (generally), where we have created it.

Lay out the cells you need named, then type in the names as text either to the left or above the cells in question. Mark the range containing both the labels and the cells that you want to name. Click on Insert Names Create, and you will see a dialog box that will have four buttons: create names in top row, left column, bottom row, right column. Excel is very good at guessing where in a highlighted range the names of the cells are and which cells you wan to name. On the assumption that it is guessed correctly (and it usually does) just click okay. If it hasn't guessed correctly, click the appropriate box and then click ok. Then you are done.

Anytime you want to review which cells are named, what cells exactly a named range refers to, or delete some you don't use anymore go to Insert Name Define and follow the instructions therein. It is a little aggravating, but I haven't found an easy way yet to delete a whole *series* of named ranges at once, which I sometimes need to do when cleaning up a complicated spreadsheet. To date I have only been able to rename them one at a time.

What's VBA and What is Excel?

A common problem when starting out is figuring out what is a reserved term in Visual Basic, like a VBA property or statement, and what is something you dream up, like a variable name. A related problem is, when you enter something, is it an Excel concept or a VBA concept? There are several tip-offs as you are beginning.

One tip-off is that something you put within parenthesis and quotes is usually the name of something - a cell reference or a named range for example – that is in the spreadsheet that you will be working with. For example: Worksheets ("Sheet1") is a reference to a worksheet with a particular name, Sheet 1. The name can be any name you like, Fred or Ethel, as long as when you run the macro Excel and Visual Basic can find it.

Another issue that arises is variable names within a VBA program. These are perhaps a little bit less obvious because we do not put them in quotes or inside parenthesis. After you begin to get familiar with VBA syntax and some vocabulary, you are quickly able to separate variable names from VBA key words. For example,

```
For X = 1 to 5
```

you have to use the word "For" and no other to represent the statement that starts the loop; but the index variable, X, can have any name you choose.

To reduce ambiguity between Excel and VBA constructs, the designers have provided a few other hints.

:= vs. = When you see = in VBA code, that's a VBA equals sign. When you want to set something equal to something in Excel *within* VBA you use :=

For example, in the code

```
X = InputBox("Insert the Value of X Now!")
```

is a VBA construct, so we use =, but in the code

```
Selection.PasteSpecial Paste:=xlValues
```

you are telling Excel to PasteSpecial Values, which is an Excel concept. Hence the use of := instead of =.

Application.*function* is another way to signal you're using an Excel function, rather than VBA. If you wrote:

```
Y = Sqrt(4)
```

VBA would return an error, because Sqrt is an Excel function, not VBA (Sqr is the name of the square root function in VBA).

If you wanted to use Excel's square root function (or one of the many, many functions that Excel has but VBA doesn't), you'd write it as:

```
Y = Application.Sqrt(4)
```

Note that for this to work, Excel has to be open when you're running the macro, so that VBA can find the right application function.

Some pre-named variables contain *built in constants* that are used to turn things "on" and "off" in a VBA program. For example, the special variable xlValues is a constant that takes the value 1 if if you want to PasteSpecial values, and 0 if you are using some other PasteSpecial (like transpose) but "turning off" values (i.e. are pasting the contents of the cell, text, formula, or number, rather than the result of evaluating the cell).

These generally have the form xl*ZZZZ* or vb*ZZZZ* for Excel built-in constants and VBA built-in constants, respectively. Note that xl is a lower case letter X, and a lower case letter L. It's not an X and a one.

What is a Variable?

Think of a variable in VBA as something akin to a cell within a worksheet. In both cases, the cell, or the variable, *contains* something. In general what we are doing in a spreadsheet is evaluating the numbers or formula with in a cell, and returning a number at the time we make use of it. We can distinguish between a cell, the contents of the cell, and the results of evaluating the contents of a cell.

Same thing with a variable in VBA. The simplest way to create variables in Visual Basic is through assignment statements. For example `X = 5` assigns the value 5 to the variable X. Of course once you have assigned the value to the variable X, you can also put X on the right hand side as in: `Y = X`

How does Visual Basic know when something you type is a variable name? From context. In some programming languages like FORTRAN, at least older versions, you needed to "declare" the variables you were going to use later at the beginning of the program so that FORTRAN knew to reserve those names for your variables. While there may be some clarity in doing so, later language writers have decide that is saves a lot of time if languages like VBA simply figure out what variables are declared by context. That is, when you write a line like `X = 5`, VBA knows you must mean to create a variable names X and put the number 5 in it. So it creates the variable "on the fly" instead of at the beginning of the program when variables are declared, as in FORTRAN.

Learning How to Do It on Your Own

It is all very well to figure out how to write macros that follow what we've done in class, or one that you have been shown by a coworker. But this is just your introduction to a lifetime of happily writing macros using VBA. How do you come up with "new stuff?"

One way is to look for structural similarities between new problems and old problems, even though they may not be obvious. For example, we usually think of stress testing a mortgage backed security as something that is very different from calculating the rate of return for investment in a proforma. But think about the macro we have just written in class. It runs a whole set of properties through a proforma. If we have a mortgage backed security that is based on, say, 100 mortgages, and we want to run each mortgage through a little test and accumulate the results, doesn't the problem have essentially the same structure?

Our database is different: it is mortgages instead of properties. The variables (columns) in the database might represent things like the loan amount, the loan to value for the mortgage, the coupon rate, the term to maturity, and other characteristics of the mortgages. In place of the cash flow model we have been using in this example, we'd write an Excel program that would apply a stress test. That model would be quite different that the proforma, but the overall structure of the problem would be the same.

In fact we could take the macro we have written for the proforma and use it as a starting point to do our stress test. We will take a look at case like this next week, just briefly, to highlight the similarities and differences of such problems and to help you think about the general issue of how to apply an old solution to a new problem.

The Macro Recorder

The macro recorder is pretty good at writing code to undertake simple operations. It is not very good, in fact it is practically useless, for controlling program flow. What do I mean by controlling program flow? Deciding when to undertake a step, and when to repeat it. How often to repeat it when necessary (using For Next, Do While, and similar constructs). On the other hand, any program can be broken down into a series of discrete tasks, assembled in a logical and orderly fashion, and repeated where necessary. The analogy to writing a paper is that the macro recorder helps you write a particular "paragraph." It does not do a very good job a giving you an outline for the whole paper/ program, or fitting the pieces together. For that you have to use your brain.

Online Help

Visual Basic Help is invoked from the Visual Basic Editor. Generally you won't get Visual Basic Help when you are in the Excel window, or vice versa. (In Excel 5.0 and previous versions that is not the case.)

Usually when I first pull up the help screen for Visual Basic I find a bunch of jargon which is hard to make much sense of (unless you have worked with Visual Basic a lot). *My strong advice is to always click on **examples** whenever you are in help.* (That is the highlighted key word in green at the top of help).

As an example, remember our discussion of how I found how to refer to a given row of a range, for our macro. I started off by searching terms that had the word "row" in it. Look at the help screen you get when you get to the VBA help for the "Rows Property." It is a little hard to make much sense of. But when I click on the example, bells start to go off in my head: I get text that says:

```
Worksheets("Sheet 1").rows(3).delete.
```

Now we are talking. I *know* what delete is. It is an action- a method in visual basic's jargon. The thing to the left of delete is the thing that is deleted: obviously the third row of worksheet sheet one. My next leap is to consider that if I can refer to the third row of a worksheet, it wouldn't be at all surprising that I could also refer to the third row of a range. A quick test where I create a small range in a worksheet, and call it Fred confirms this. The command worksheets ("Sheet 1").range ("Fred").rows(3).delete does in fact delete the third row of Fred.

Now the next leap is that if I can delete the third row I can probably do other things to it like copy and paste it or any other action I want to take (method I want to apply in VBA-speak).

Manuals

Another way I get ideas is to flip through reference books. Reference works like Microsoft (1996) are notoriously hard to read from cover to cover and not very good for teaching you the basics of VBA. But once you have a general sense of what you are doing, flipping through the book from time to time and getting familiar with the kinds of method and properties that exist can be surprisingly useful.

Reviewing Others' Programs

Another way to learn new things is to look at previous examples. I like to get books like Carlberg (1995) and just look through them, looking for interesting examples and problems that may have some relevance to my work. Rarely do I find something I can use out of the can, but I often get ideas and learn something about better structures or programming style at the same time. Looking at other people's programs to learn how to write better VBA code is like reading first-rate authors to help become a better writer.

Selected References

Introductory

Boonin, Elisabeth. *Using Excel Visual Basic for Applications*. Indianapolis: Que Corporation, 1996. (I like this book a lot as an intro. It is written for the VBA that shipped with Excel 5.0, so some things have changed; except for the advent of the Visual Basic Editor, not enough to make much of a difference).

Cummings, Steve. *VBA For Dummies*. IDG Books, 1998.

Walkenbach, John. *Excel 97 Programming for Windows for Dummies*. IDG Books, 1997.

Intermediate VBA

Microsoft Press. *Microsoft Excel/Visual Basic Programmer's Guide*. 1995 (look for a newer edition).

Microsoft Press. *Microsoft Excel/Visual Basic Reference*. Second Edition, 1995 (look for a newer edition. I use this reference a lot, it's very handy to flip through after you're comfortable with the concepts in this handout.)

Examples of Business Applications, Intermediate

Adkins, Kathleen. *Building Business Spreadsheets Using Excel*. Adkins and Matchett, 1998.

Carlberg, Conrad. *Business Analysis with Excel*. Que, 1995.

Examples of Business Applications, Advanced

Benninga, Simon and Benjamin Czaczkes. *Financial Modeling*. MIT Press, 1997. Some mistakes to watch out for! But great for the financially minded.

Mayes, Timothy R. and Todd M. Shark. *Financial Analysis with Microsoft Excel*. HBJ, 1997.