# Part C: Excel Collection Objects and Objects

Download This Tutorial >>
Click Here!

**In Part  you will learn about:**

**-** Objects

- VBA Collection Objects

- The Application Object

- Workbook and Window Objects

- The Worksheet Object

- The Range Object

## 1) What is object?

It really is a pretty simple thing to understand. You can think of objects as separate computer programs with specific (and often common) functions that are available for repeated use in your programs.

Objects are dynamic in that they can be easily manipulated in code with the various parameters used to define them.

For example, consider a car. A car can be described by its size, color, and brand (among other things). For example, it might be a SUV, sports car, convertible and etc. The color, size, and brand are all adjectives that describe the car. Thus, they are all *properties* of the car.

A car can also perform various actions like move straight or turn. Moving and turning are action verbs that tell you what tasks the car can do. Moving and turning are *methods* of the car. Finally, the car is built out of other objects such as a frame, wheels, steering, and tyres.

These objects, in turn, have their own properties and methods. For example, a car wheel is of a certain diameter, is built out of aluminum or titanium alloys, and it turns or rolls. The diameter and type of material are properties of the wheel object, and to turn or roll would be two of its methods. So you see, there is sort of a hierarchy to the objects in your car and the car object itself sits at the top of the hierarchy.

I could take it further. For example, a wheel is built from a tyre, rim, and spoke objects. The tyres are built from organic polymers, and so on, and so on. The description continues until eventually you will get to the objects at the very bottom of the hierarchy.

In Excel, I bet that you're already familiar with many of its objects. For example, there are Workbook objects, Worksheet objects, Range objects, Chart objects, and many more. These objects are the building blocks to construct a program with VBA.

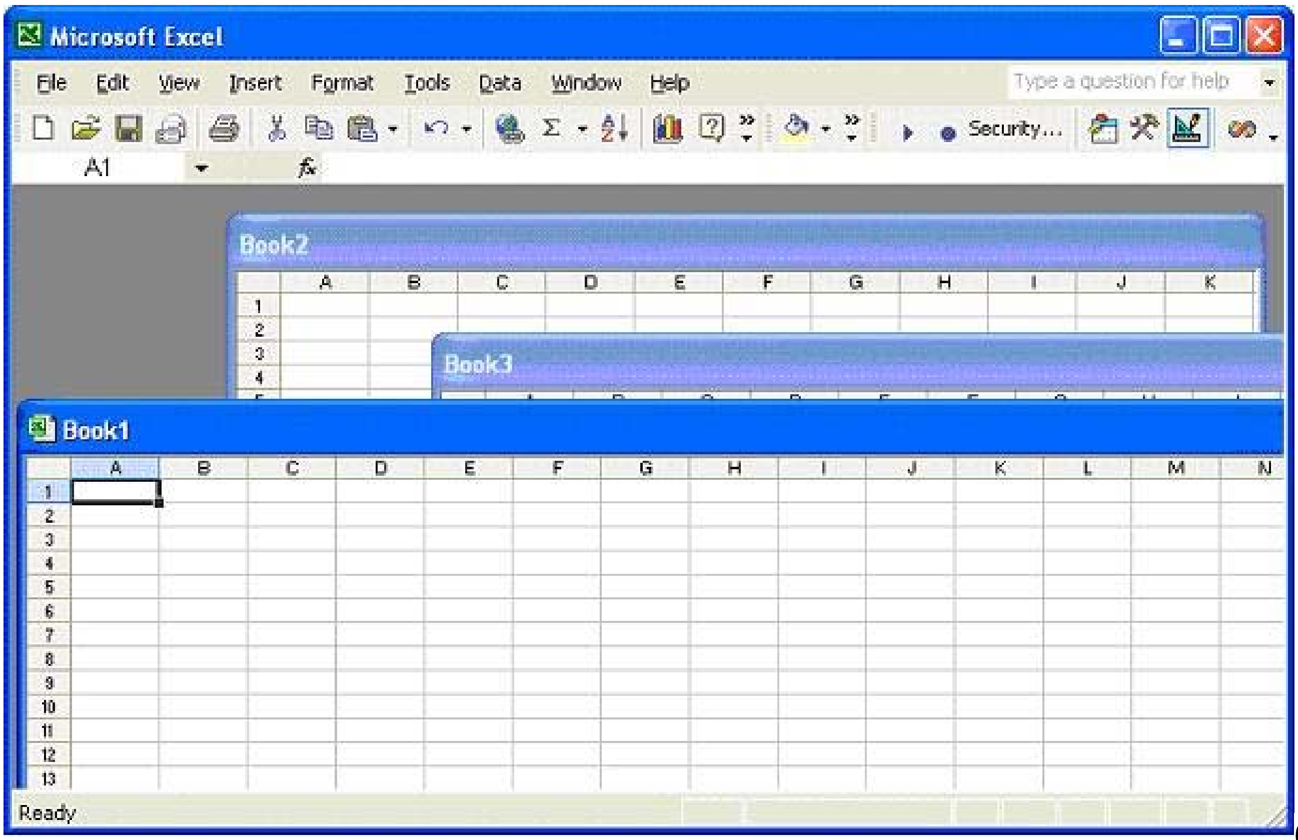
The rest of this Part C, I will show you how to use a few of Excel's objects, and in particular, some of its top-level objects.

### a) VBA Collection Objects

To understand what is collection objects in VBA, just think of a collection of cars. The car objects in your car collection can be different sizes, colors, and brand, but they are all cars. So, VBA collection objects means that it is a group or collection of the same object types

For example, any Workbook object belongs to a Workbooks collection object. The Workbooks collection object contains all open Workbook objects. The Excel window shown in Figure 3.1 contains three open Workbook objects (Book1, Book2, and Book3).

Collection objects enable you to work with objects as a group rather than just working with a single object.



(Figure 3.1)

## Top-Level Excel Objects

**The hierarchy of Excel Objects**

| |
|---|
| Application |
| Workbooks |
| Workbook |
| Sheets |
| Worksheet |
| Range |

There are too many objects in Excel to cover them all. So in this Part, I'll only explain to you the objects shown in the Figure above. My goal is to get you comfortable navigating through the object model and learning how to use the objects on your own.

**b) The Application Object**

At the top of the hierarchy is Application object. It represents the entirety of the Excel application. As the top-level object it is unique and thus, seldom needs to be addressed in code.

However, there are a few occasions when you must use the Application object's qualifier in code.

For example, the Width and Height properties used to set the size of the application window must be reference explicitly in the VBA code. Other example will be the DisplayFormulaBar property used to show or hide the formula bar.

*Application.Width = 800*

*Application.Height = 550*

*Application.DisplayFormulaBar = False*

You need to use the Application object qualifier, most of the time, to set properties pertaining to the appearance of the Excel window, such as shown above, or the overall behavior of Excel as shown below.

*Application.Calculation = xlManual*

*Application.EditDirectlyInCell = False*

*Application.DefaultFilePath = "D:\My Documents"*

You need to use the Application object qualifier with the very helpful ScreenUpdating property.

*Application.ScreenUpdating = False*

Another one will be the WorksheetFunction property.

*Range("C3") = Application.WorksheetFunction.Sum(Range("B1:B10"))*

However if you just need to set properties of lower-level objects, then the Application object qualifier is not needed.

*ActiveCell.Formula = "=SUM(B1:B10)"*

## c) The Workbook and Window Objects

You may be unfamiliar with the Window object. Window objects refer to instances of windows within either the same workbook, or the application. Within the Excel application, the Windows collection object contains all Window objects currently opened; this includes all Workbook objects and copies of any Workbook objects. The Window objects are indexed according to their layering. For example, in Figure 3.1, you could retrieve Book2 with the following code:

Application.Windows(2).Activate

because Book2 is the center window in a total of three Window objects. After Book2 is retrieved and thus brought to the top layer its index would change to 1 when using the Windows collection object. This is different from accessing Book2 using the Workbooks collection object.

Let’s take a closer look at the Workbooks collection Object via an example. There are only a few properties and methods of the Workbooks collection object and their functions are straightforward. Add the following procedure to a standard module in a workbook.

```
Public Sub exampleWorkbooks()

Dim j As Integer

For j = 1 To 3

Workbooks.Add      ‘add a new workbook

Next j

Workbooks(Workbooks.Count).Activate      ‘activate book3

End Sub
```

In you execute this procedure by selecting exampleWorkbooks from the Macro menu in Excel, you will immediately see three new workbooks opened in Excel. After that it will activate Book3

They are relatively straightforward to use, and you have already seen a couple of them (the Add() method, Count property and the Activate() method). You may find the Open() and Close() methods and Item property useful as well. Some of these members will be addressed later, albeit with different objects. You will find that many of the collection objects share the same properties and methods. This is not unusual, but be aware that depending on the object you use, the parameters that are either available or required for these members may vary.

Consider the following VBA procedure illustrating the use of the Close() method of the Workbook object. The code can be placed in a standard or object module.

```
Public Sub CloseWorkbooks()

Workbooks(Workbooks.Count).Close SaveChanges:=False

Workbooks(1).Close SaveChanges:=False

End Sub
```

This procedure will close the first and last workbooks opened in Excel without prompting the user to save changes.

In the example above, the Close() method of the Workbook object is used, not the Close() method of the Workbooks collection object. If you want to close all workbooks simultaneously, then use the Close() method of the Workbooks collection object.

The code below will close the Workbooks collection object.

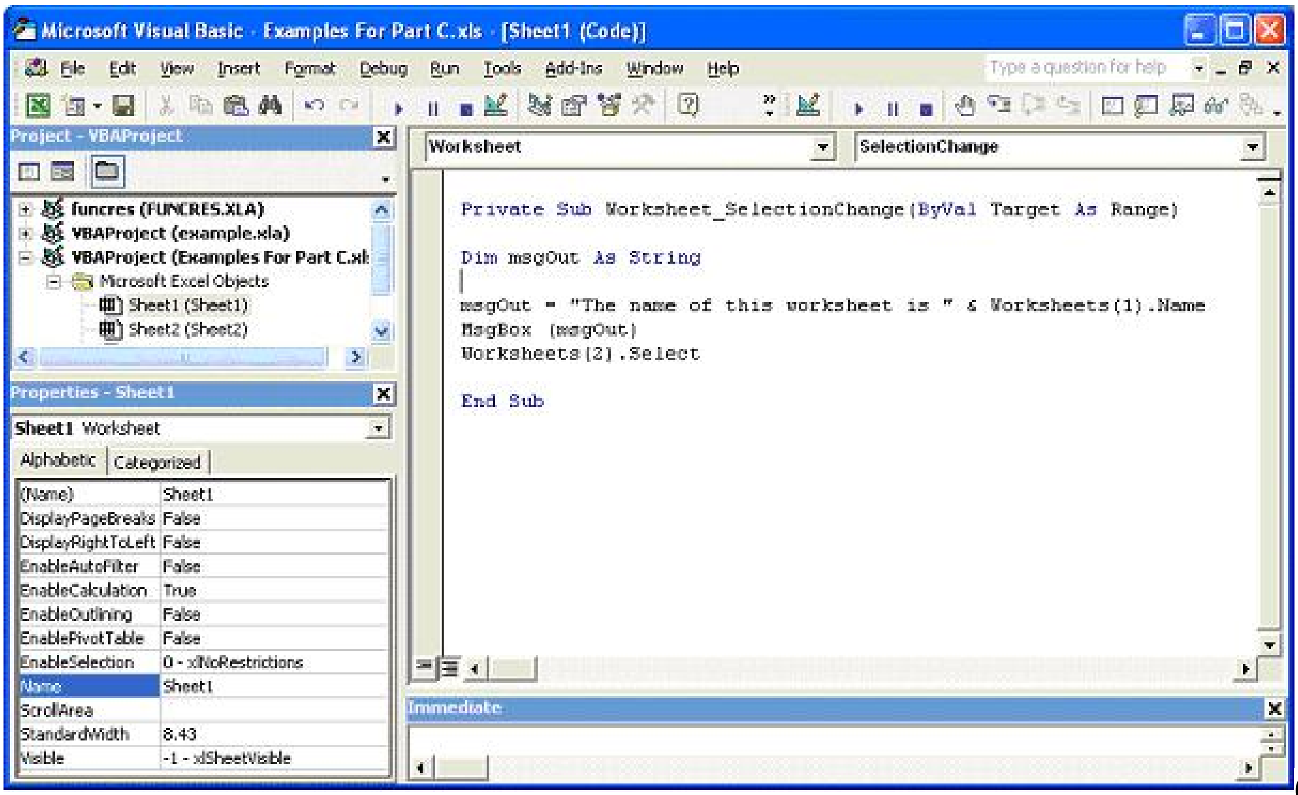
```
Workbooks.Close
```

In this case, there are no optional arguments allowed, so the user will be prompted to save the currently selected workbook.

**d) The Worksheet Object**

The Worksheet object falls just under the Workbook object in Excel's object hierarchy. To investigate some of the events of the Worksheet object, the following code has been added to the SelectionChange() event procedure of Sheet1 in a workbook. (Figure 3.3)

```
Private Sub Worksheet_SelectionChange(EyVal Target As Range)

Dim msgOut As String

msgOut = "The name of this worksheet is " & Worksheets(1).Name

MsgBox (msgOut)

Worksheets(2).Select

End Sub
```


Microsoft Visual Basic - Examples For Part C.xls - [Sheet1 (Code)]

| | |
|---|---|
| (Name) | Sheet1 |
| DisplayPageBreaks | False |
| DisplayRightToLeft | False |
| EnableAutoFilter | False |
| EnableCalculation | True |
| EnableOutlining | False |
| EnablePivotTable | False |
| EnableSelection | 0 - xlNoRestrictions |
| Name | Sheet1 |
| ScrollArea | |
| StandardWidth | 8.43 |
| Visible | -1 - xlSheetVisible |

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)

Dim msgOut As String

msgOut = "The name of this worksheet is " & Worksheets(1).Name
MsgBox (msgOut)
Worksheets(2).Select

End Sub
```

(Figure 3.3)

The SelectionChange() event procedure was is found in the object module of a worksheet.

The SelectionChange() event procedure is triggered whenever the user changes the current selection in the worksheet. The Target argument passed to the SelectionChange() event procedure is a range that represents the cells selected by the user. I will discuss the Range object shortly; for right now, ignore it because the current example does not use the passed argument.

The code in the SelectionChange() event procedure is straightforward. First, a string variable is created and assigned a value (“The name of this worksheet is”) that is then concatenated with the name of the worksheet obtained from the Name property of the Worksheet object. The full object path is not used to return the name of the worksheet, as this code will only be executed when the user changes the selection in the first worksheet of the Worksheets collection object (Sheet1). Therefore, the object path travels through the current Workbook object.

This is why index numbers can be used with the Worksheets property of the Workbook object without having to worry about returning the wrong sheet. After displaying the concatenated string in a message box, the Select() method of the Worksheet object is used to select the second worksheet in the Worksheets collection object.

Next, code is added to the Worksheet_Activate() event procedure of Sheet2. The Worksheet Activate() event procedure is triggered when a worksheet is first selected by the user or, in this case, by selecting the worksheet using program code (Worksheets(2).Select). The code is essentially the same as the previous example.

```
Private Sub Worksheet_Activate()

Dim msgOutput As String

msgOutput = “This worksheet is “ & Worksheets(2).Name

MsgBox (msgOutput)

End Sub
```

The Worksheet_Activate() event procedure is not triggered when a workbook is first opened, so it is not a good place for initialization routines intended to run as soon as a workbook is opened. These procedures should be placed in the Workbook_Open() event procedure.

The Sheets collection object is nearly identical to the Worksheets collection object and the two objects can often be used interchangeably (as is the case in the previous two examples). The difference between these two objects is that the Sheets collection object will also contain any chart sheets open in the active workbook. So, if you expect chart sheets to be open in the workbook of interest, you should access worksheets using the Sheets collection object; otherwise, either collection object will suffice.

### i) Methods and Properties of the Worksheet

Select a sheet in the VBA Project window (Figure 3.3) and you can see 11 properties of the worksheet in the Properties window of the VBE, properties for which you can set a default value to begin with and that you can modify through the VBA procedure whenever you wish.

There are 3 properties of a worksheet that you will use frequently:

- the name (name within parentheses),
- the name (without parentheses) like the caption appearing on the sheet's tab in Excel
- the visible property

To change the caption you can either do it in the property window of the VBE or in Excel by right clicking on the tab then selecting "Rename". Programmatically you can change the caption of a sheet with the following code:

```
Worksheets("Sheet1").Name= "Good"
```

The "Visible" property can take 3 different values. The first two are True or False meaning the a certain sheet is or is nor visible that it is either hidden or not.

```
Worksheets("Good ").Visible= True
Worksheets("Good 1").Visible= False
```

Remember that formulas in cells are calculated even if the sheet is hidden but before you can do anything programmatically on the sheet you must unhide it:

```
Worksheets("Good ").Visible = True
Worksheets("Good "). Select

Range(“A1”).Value = 10
```

The third value that the property "Visible" can take is very interesting. A sheet can be very hidden " *Worksheets("Good ").Visible = xlVeryHidden*". In this state not only the sheet is hidden but you can't see its name when in Excel you go to "Format/Sheets/Unhide". The value xlVeryHidden can only be changed programmatically. That means that only users that have access to the VBA code can unhide this sheet. If your code is protected by a password only users with the password can access the code and modify the "xlVeryHidden" value. You can use this value of the property "Visible" to hide confidential information like credit card details and personal info or to hide parameters that you don't want modified by the user.

```
Worksheets("Good ").Visible = True
Worksheets("Good "). Select
```

```
Range("A1").Value = 10
```

And hide it again…

```
Worksheets("Good ").Visible = xlVeryHidden
```

Remember also that formulas on other sheets referring to cells of a hidden or very hidden sheet work even if the sheet is hidden or very hidden.

You might want to delete sheets. Here is the code to do so:

```
Worksheets("Good ").Delete
```

You might also want to add one sheet. If you use the following code VBA will add a new sheet before the active worksheet.

```
WorkSheets.Add
```

If you want to be more precise as to the where and the how many you will use either of the following procedures:

Inserting one sheet after the sheet which caption is "Good" the code is like this

```
Sub exercise2()
   Sheets.Add before:=Sheets("Good")
End Sub
```

Inserting three sheets after the sheet which caption is "Good":

```
Sub exercise2()
   Sheets.Add after:=Sheets("Good "), Count:=3
End Sub
```

Inserting one sheet at the beginning of the workbook. Notice the absence of double quotes when using the rank of the sheet:

```
Sub exercise2 ()
   Sheets.Add after:=Sheets(1)
End Sub
```

And if you want to add a new sheet at the end of the workbook you need to count the sheets with **Sheets.Count** and use this value as the rank of the sheet after which you want to add the new sheet:

```
Sub exercise2 ()
   Sheets.Add After:=Sheets(Sheets.Count)
End Sub
```

## e) The Range Object

You can use the Range object represents a group of one or more contiguous cells in an Excel worksheet. It is extremely useful, as it allows us to manipulate the properties of an individual cell or collection of cells in a worksheet. You will probably find yourself using the Range object in every program you write using VBA for the Excel application.

Consider the following code examples that use properties of the Range object.

```
Range(“B1”).Value=”Column B”

Range(“B1:G1”).Columns.AutoFit

Range(“B1:C1”, “E1:F1”).Font.Bold = True
```

The Range object is one example of a VBA collection object that does not use the plural form of an existing object for its name. The Range object is a collection object in the sense that it represents a collection of cells in a worksheet, even if the collection represents only one cell.

First, note that a long object path is omitted from the examples above; thus, these lines of code will operate on the currently selected worksheet. The first line inserts the text ″Column B″ into cell B1 by setting its Value property. The Range property was used to return a Range object representing a single cell (B1) in this example. You have already seen several examples of the Value property previously. Although the Value property exists for several objects, it is the Range object for which it is most commonly used.

When you use the AutoFit() method of the Range object to adjust the width of columns B through G such that the contents of row 1 will just fit into their corresponding cells without overlapping into adjacent columns. This is equivalent to the user selecting Format, Column, AutoFit Selection from the Excel application menu.

The third and last example demonstrates setting the Bold property of the Font object to true for two distinct ranges in the active worksheet. The two ranges are B1:C1 and E1:F1. You are only allowed to return a maximum of two ranges, so adding a third range to the arguments in the parentheses would generate a run-time error.

The examples above demonstrate just a couple of formatting methods and properties belonging to the Range object (AutoFit(), Columns, and Font). If you are a regular user of Excel, then you have probably surmised that there are numerous other properties and methods related to formatting spreadsheet cells.

Recording a macro is a quick and easy way to generate the code you need without having to search the documentation for descriptions of the desired objects, properties and methods. After you have recorded the macro in a separate module, you can clean up the recorded code and then cut and paste into your program as needed.

You may have noticed that the range arguments used in the examples above (B1, B1:G1, etc.) are of the same form used with cell references in the Excel application. This identical syntax is highly convenient because of its familiarity.

### i) Using the Cells Property

The Excel Object Model does not contain a Cells object. In order to reference a specific cells you use either the Cells property or the Range property. The Cells property returns a Range object containing all (no indices used) or one (row and column indices are specified) of the cells in the active worksheet. When returning all of the cells in a worksheet, you should only use the Cells property with the Application and Worksheet objects. For example,

```
Cells(2,2).Select
```

Here you've selected the cell B2 in the worksheet.

To return a single cell from a Worksheet object you must specify an index. The index can be a single value beginning with the left uppermost cell in the worksheet (for example, Cells(5) returns cell E1) or the index can contain a reference to the row and column index (recommended) as shown below.

```
Cells(1, 4).Value=5

Cells(1, ”D”).Value =5
```

This is the familiar notation used throughout this book. Both lines of code will enter the value 5 into cell D1 of the active worksheet. You can either use numerical or string values for the column reference. You should note that the column reference comes second in both examples and is separated from the row reference by a comma. I recommend using the second example above, as there is no ambiguity in the cell reference—though on occasion it's convenient to use a numerical reference for the column index. Now consider some examples using the Cells property of the Range object.

```
Range(“C5:E7”).Cells(2, 2).Value = 100

Range(“C5:E7”).Cells(2, “A”).Value = 100
```

This code may confuse you because they appear to be trying to return two different ranges within the same line of code; however, that is not the case, but you can use these examples to more carefully illustrate how the Cells property works.

Before reading on, guess in what worksheet cell each of these lines places the value 100. If you guessed cells B2 and A2, respectively, you're wrong. Instead, the value 100 is entered in cells D6 and A6, respectively, when using the above lines of code. Why? It's because the Cells property uses references relative to the selected range. Without the reference to the Range object in each statement (Range(“C5:E7”)), the current range is the entire worksheet, thus Cells(2,2) returns the range B2; however, when the selected range is C5:E7, Cells(2,2) will return the second row from this range (row 6) and the second column (column D). Using a string in the Cells property to index the column forces the selection of that column regardless of the range selected. The row index is still relative; therefore, the second example above returns the range A6.

## ii) Method and Properties Range (Cells, Rows and Columns)

Here is some code to move around and work with the Range object.

### Selection and ActiveCell

The object **Selection** comprises what is selected. It can be a single cell, many cells, a column, a row or many of these.

For example:

```
Range("A1:A50").Select
Selection.ClearContents
```

will remove the content (values or formulas) of the cells A1 to A50..

The ActiveCell is a very important concept that you will need to remember as you start developing more complex procedures.

### Range, Select

To select a cell you will write:
*Range("B1").Select*

To select a set of contiguous cells you will write:
*Range("A1:A5").Select*

To select a set of non contiguous cells you will write:
*Range("C1,E5,F6").Select*

### Columns, Rows, Select, EntireRow, EntireColumn

To select a column you will write:
*Columns("A").Select*

To select a set of contiguous columns you will write:
*Columns("A:B").Select*

To select a set of non contiguous columns you will write:
*Range("A:A,C:C,F:F").Select*

To select a row you will write:

```
Rows("1").Select
```

You can also select the column or the row with this:
```
ActiveCell.EntireColumn.Select
ActiveCell.EntireRow.Select
Range("B1").EntireColumn.Select
Range("B1").EntireRow.Select
```

If more than one cell is selected the following code will select all rows and columns covered by the selection:
```
Selection.EntireColumn.Select
Selection.EntireRow.Select
```

**Cells, CurrentRegion**

To select all cells then
```
Cells.Select
```

**Offset**

The commonly use **Offset** method allows you to move right, left, up and down.

For example if you want to move 2 cells to the right, you will write:
```
Activecell.Offset(0,2).Select
```

If you want to move 2 cells to the left,
```
Activecell.Offset(0,-2).Select
```

If you want to move two cells down:
```
Activecell.Offset(2,0).Select
```

If you want to move two cells up:
```
Activecell.Offset(-2,0).Select
```

If you want to select one cell and three more down:
```
Range(Activecell,Activecell.Offset(3,0)).Select
Range("A1",Range("A1").Offset(3,0)).Select
```

**Column, Row, Columns, Rows, Count**

For the following lines of code notice that you need to send the result into a variable.

**myvar = Activecell.Column** will return the column number

**myvar = Activecell.Row** will return the row number

**myvar = Selection.Columns.Count** will return the number of columns in the selection

**myvar = Selection.Rows.Count** will return the number of rows in the selection

**myvar = Selection.CurrentRegion.Rows.Count** will return the number of rows in the current region of the selection

**Value**

When you want to enter a numerical value in a cell you will write:
*Range("C1").Select*
*Selection.Value = 56*

Note that you don't need to select a cell to enter a value in it, from anywhere on the sheet you can write:
*Range("C1").Value = 56*

You can even change the value of cells on another sheet with:
*Worksheets("Good").Range("C1").Value = 56*

You can also enter the same value in many cells with:
*Range("A1:B33").Value = 56*

If you want to enter a text in a cell you need to use the double quotes like:
*Range("C1").Value  = "Nancy"*

If you want to enter a text within double quotes in a cell you need to triple the double quotes like:
*Range("C1").Value  = """Peter"""*

**Formula**

To enter a formula in a cell you enter this code
*Range("A1").Select*
*Selection.Formula = "=B8+C8"*

Note the two equal signs (=) including the one within the double quotes like if you were entering it manually. Again you don't need to select a cell to enter a formula in it, from anywhere on the sheet you can write:
*Range("A1").Formula = "=B8+C8"*

If you write the following:
*Range("A1:A8").Formula = "=C8+C9"*

The formula in A1 will be =C8+C9, the formula in A2 will be =C9+C10 and so on. If you want to have the exact formula =C8+C9 in all the cells, you need to write:

*Range("A1:A8").Formula = "=$C$8+$C$9"*

## 2) Simplifying object references

*Application.Workbooks("Book1.xls").Worksheets(1).Range("A1").Value*

You don't need to fully qualify every object reference you make, like the above. Excel provides you with some shortcuts that can improve the readability (and save you some typing). For starters, the Application object is always assumed. Omitting the Application object reference shortens the example from the previous section to

*Workbooks("Book1.xls").Worksheets(1).Range("A1").Value*

If Book1.xls is the active workbook, you can omit that reference too. This bring us down to

*Worksheets(1).Range("A1").Value*

Futher, if the first worksheet is the currently active worksheet, then Excel will assume that reference and allow us to just type

*Range("A1").Value*

### a) Working with Objects

You learn in numerous examples of objects on how to set their properties and invoke their methods and events. There are a few more tools that can be very useful when working with objects i.e.the With/End With code structure.

This code structure works well to simplify code; and the object data type, which allows you to reference existing objects or even create new objects. We'll look at an example below.

**The With/End With Structure**

I always recommend the use of this structure because it makes your programs more readable. Also you will often see the With/End With structure in recorded macros. Consider the following code:

*Range("A1:D1").Select*

*Range("A1:D1").Value = 100*

*With Selection.Font*

*.Bold = True*

```
.Name = “Times New Roman”
.Size = 16
End With
With Selection
.HorizontalAlignment = xlCenter
.VerticalAlignment = xlCenter
End With
```

When executed, this code selects the range A1:D1 of the active worksheet using the Select() method of the Range object.

In this case, the Selection property of the Window object is used to return a Range object from which the Font property returns a Font object associated with the selected range. The statement could have just as easily been written without the Select() method and Selection property and entered using the Range property to return the desired Range object (for example, With Range(“A1:D1”).Font).

Once inside the structure, any property of the object can be set without having to qualify the object in each line of code. Subordinate objects and their properties can also be accessed. Each line within the structure must begin with the dot operator followed by the property or object name, then the method or assignment.

After all desired properties and/or methods have been invoked for the given object, the structure closes with End With.

You will note that a second With/End With structure is used to set the horizontal and vertical alignment of the selected range. This way your code look clean and readable.

The With/End With structure is straightforward and particularly useful when a large number of properties or methods of one object are to be addressed sequentially in a program.

## 3) The Object Data Type

Lastly, the subject on Excel objects would not be complete without a discussion of the object data type. If you find multiple instances of the same object in your program, then you can use an object variable to handle the reference rather than constantly retyping the qualifiers. Also, variables can be assigned meaningful names, making the program easier to interpret. Object variable are similar to other VBA data types in that they must be declared in code. For example,

```
Dim myObject as Object
```

declares an object variable named myObject; however, assigning a value to an object variable differs from assignments to more common data types. The Set keyword must be used to assign an object reference to a variable.

```
Set myObject = Range("B1:C15")
```

This will assign the Range object representing cells A1 through A15 to the variable myObject. Properties of the object can then be initialized in the usual way.

```
myObject.Font.Bold = False
```

This sets the values in cells B1 through C15 to be displayed in bold-face type. Or, using the Range object, the above example can be rewritten more efficiently as follow

```
Dim myRange as Excel.Range

Set myRange=Range("B1:C15")

myRange.Font.Bold = True
```

To be more efficient you can also include the library (Excel) in your declaration. Here the Range object type will be referenced at compile time and VBA will have no trouble working out references to the properties and methods of the object, as the type of object and the library to which it belongs have been explicitly declared. You will see more examples of object variable types in the as we go along.

## 4) Summary

When you do Excel programming, its all about manupulating the Excel objects. There are so many things we can do with these object and you can be as creative as you want. Like the neural network based forecasting addin develop by my company.

Previous | Back to Home | Back to TOC | Next

TOP