VBA Advanced

Sample manual - first two chapters

Wise Owl Training

Manual 1343 - 173 pages –

TABLE OF CONTENTS (1 of 6)

TABLE OF CONTENTS (2 of 6)

TABLE OF CONTENTS (3 of 6)

TABLE OF CONTENTS (4 of 6)

TABLE OF CONTENTS (5 of 6)

TABLE OF CONTENTS (6 of 6)

CHAPTER 1 - VBA RECAP

1.1 VBA Reference

This chapter provides you with a quick reference to some of the common bits of VBA that you're hopefully already familiar with.

Creating Procedures

The table below shows how to define the two most common types of procedure in VBA.

How to…	Code
Declare a subroutine	`Sub NoSpacesInProcedureNames()`
	`    'this is a comment`
	`    'applying a method to an object`
	`    Object.Method`
	`    'changing a property of an object`
	`    Object.Property = Something`
	`End Sub`
Declare a function	`Function MyFunction() As DataType`
	`    'do something useful`
	`    'then return a value`
	`    MyFunction = Something`
	`End Function`

Selecting and Activating Excel Objects

This section explains how to go to a workbook, worksheet and range of cells in Excel.

How to…	Code
Go to a workbook	`Workbooks("Book1.xlsm").Activate     'go to the named workbook`
	`Workbooks(1).Activate                'go to the 1st open workbook`
	`ThisWorkbook.Activate        'go to the workbook this code is in`
Go to a worksheet	`Worksheets("Sheet1").Select      'go to the named worksheet`
	`Worksheets(1).Select             'go to the left most worksheet`
	`Sheets("Sheet1").Select     'go to the named worksheet or chart`
	`Sheets(1).Select            'go to the left most worksheet or chart`
	`Sheet1.Select       'go to the worksheet whose codename is Sheet1`
Go to a range	`Range("A1").Select              'select cell A1`
	`Range("A1:B5").Select           'select A1 to B5`
	`Range("A1", "B5").Select        'select A1 to B5`
	`Range("A1,B5,D10").Select       'select A1 and B5 and D10`
	`Range("MyRangeName").Select     'select the named range`

Selecting an Excel Range Relatively

The techniques in the table below show how to select a range relative to another range in Excel.

How to…	Code

Move a number of rows and columns away	`ActiveCell.Offset(1, 0).Select      'move down 1 cell`
	`ActiveCell.Offset(0, 1).Select      'move right 1 cell`
	`ActiveCell.Offset(-1, 0).Select     'move up 1 cell`
	`ActiveCell.Offset(0, -1).Select     'move left 1 cell`

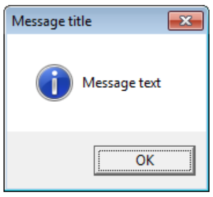
Go to the end of a list in one direction	`ActiveCell.End(xlDown).Select       'go to bottom of block (ie cell C8)`
	`ActiveCell.End(xlToRight).Select    'go to right of block (ie cell F4)`
	`ActiveCell.End(xlUp).Select         'go to top of block (ie cell C2)`
	`ActiveCell.End(xlToLeft).Select     'go to left of block (ie cell B4)`

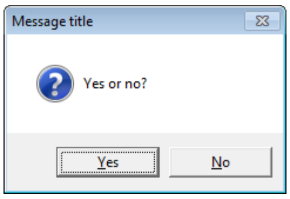
Select from one cell to the end of the list	`'select from the activecell to the bottom of the list`
	`Range(ActiveCell, ActiveCell.End(xlDown)).Select`
	
	`'select from cell A1 to the bottom of the list`
	`Range("A1", Range("A1").End(xlDown)).Select`
	
	`'select from A1 to the bottom right corner of the list`
	`Range("A1", Range("A1").End(xlDown).End(xlToRight)).Select`

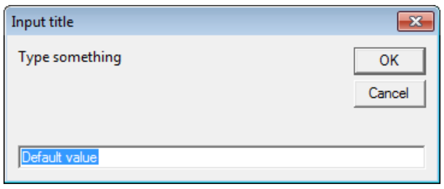
Messages and Inputs

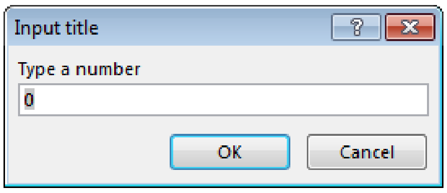
The table below shows how to display messages and ask for user input.

How to…	Code	Result

Show a message	`MsgBox _`
	`    Prompt:="Message text", _`
	`    Buttons:=vbInformation, _`
	`    Title:="Message title"`
		 Message title / Message text / OK

Ask a yes or no question	`Dim Result As VbMsgBoxResult`
	
	`Result = MsgBox( _`
	`    Prompt:="Yes or no?", _`
	`    Buttons:=vbQuestion + vbYesNo, _`
	`    Title:="Message title")`
		 Message title / Yes or no? / Yes / No

Ask for a string	`Dim Result As String`
	
	`Result = InputBox( _`
	`    Prompt:="Type something", _`
	`    Title:="Input title", _`
	`    Default:="Default value")`
		 Input title / Type something / OK / Cancel / Default value

Ask for a number in Excel	`Dim Result As Long`
	
	`Result = Application.InputBox( _`
	`    Prompt:="Type a number", _`
	`    Title:="Input title", _`
	`    Default:=0, _`
	`    Type:=1)`
		 Input title / Type a number / 0 / OK / Cancel

Declaring Variables

This section shows how to declare and assign values to variables.

How to…	Code

Force explicit variable declaration	`'add this to the top of a module`
	`Option Explicit`

Declare data type variables	`Dim SmallWholeNumber As Byte`
	`Dim MediumWholeNumber As Integer`
	`Dim BigWholeNumber As Long`
	`Dim BigDecimalNumber As Single`
	`Dim HugeDecimalNumber As Double`
	`Dim AccurateDecimalNumber As Currency`
	`Dim TrueOrFalse As Boolean`
	`Dim DateAndOrTime As Date`
	`Dim SomeText As String`
	`Dim AnyTypeOfData As Variant`

Assign values to data type variables	`SmallWholeNumber = 255`
	`MediumWholeNumber = 32767`
	`BigWholeNumber = 2147483647`
	`BigDecimalNumber = 1.234567`
	`HugeDecimalNumber = 1.23456789012345`
	`AccurateDecimalNumber = 123456789012345.6789@`
	`TrueOrFalse = True`
	`DateAndOrTime = #2/29/2016#`
	`SomeText = "any bit of text"`
	`AnyTypeOfData = "any type of value"`

Declare object variables	`'Excel objects`
	`Dim wb As Workbook`
	`Dim ws As Worksheet`
	`Dim r As Range`
	
	`'Word objects`
	`Dim doc As Document`
	`Dim p As Paragraph`
	`Dim r As Range`
	
	`'PowerPoint objects`
	`Dim pres As Presentation`
	`Dim sld As Slide`
	`Dim shp As Shape`

Set a reference in an object variable	`'Excel objects`
	`Set wb = Workbooks("Book1.xlsm")`
	`Set ws = wb.Worksheets("Sheet1")`
	`Set r = ws.Range("A1:B5")`
	
	`'Word objects`
	`Set doc = Documents.Add`
	`Set p = doc.Paragraphs(1)`
	`Set r = p.Range`
	
	`'PowerPoint objects`
	`Set pres = Presentations.Add`
	`Set sld = pres.Slides.Add(1, ppLayoutTitle)`
	`Set shp = sld.Shapes(1)`

Conditional Statements

The table below shows a variety of methods for testing conditions and performing different actions based on the result.

How to…	**Code**

Write a single-line If	`'one logical test and one action`
	`If Range("A1").Value < 0 Then Exit Sub`

Write a Block If	`'one logical test and multiple actions`
	`If Range("A1").Value < 0 Then`
	`    MsgBox "No negative numbers"`
	`    Exit Sub`
	`End If`

Include an Else clause	`'one logical test with two outcomes`
	`If Range("A1").Value < 0 Then`
	`    MsgBox "No negative numbers"`
	`    Exit Sub`
	`Else`
	`    MsgBox "Value is valid"`
	`End If`

Use ElseIf statements	`'multiple logical tests with multiple outcomes`
	`If Range("A1").Value < 0 Then`
	`    MsgBox "No negative numbers"`
	`    Exit Sub`
	`ElseIf Range("A1").Value = 0 Then`
	`    MsgBox "Must be greater than 0"`
	`    Exit Sub`
	`Else`
	`    MsgBox "Value is valid"`
	`End If`

Write a Select Case statement	`'conditions using SELECT CASE`
	`Select Case Range("A1").Value`
	`    Case Is > 0`
	`        MsgBox "A1 is positive"`
	`    Case Is = 0`
	`        MsgBox "A1 is zero"`
	`    Case Else`
	`        MsgBox "A1 is negative"`
	`End Select`

Looping

The table below shows a variety of ways to repeat a set of instructions in a loop:

How to…	Code

Loop a number of times	`Dim Counter As Long`
	
	`For Counter = 1 To 10`
	
	`    Cells(Counter, 1).Interior.ColorIndex = Counter`
	
	`Next Counter`

Loop until a condition is met	`Range("A1").Select`
	
	`Do Until ActiveCell.Value = ""`
	
	`    Debug.Print ActiveCell.Value`
	`    ActiveCell.Offset(1, 0).Select`
	
	`Loop`

Loop while a condition is true	`Range("A1").Select`
	
	`Do While ActiveCell.Value <> ""`
	
	`    Debug.Print ActiveCell.Value`
	`    ActiveCell.Offset(1, 0).Select`
	
	`Loop`

Exiting from a Loop

You can exit from a loop prematurely using the **Exit** statement. You can see how to do this in the examples shown below:

How to…	Code

Exit from a For Next loop	`Dim i As Integer`
	
	`For i = 1 To 100`
	
	`    Debug.Print Cells(i, 1).Value`
	
	`    If Cells(i, 1).Value = Cells(i - 1, 1).Value Then`
	`        Exit For`
	`    End If`
	
	`Next i`

Exit from a Do Loop	`Do Until ActiveCell.Value = ""`
	
	`    Debug.Print ActiveCell.Value`
	
	`    If ActiveCell.Value = ActiveCell.Offset(-1, 0).Value Then`
	`        Exit Do`
	`    End If`
	
	`    ActiveCell.Offset(1, 0).Select`
	`Loop`

CHAPTER 2 - OBJECT ORIENTED PROGRAMMING

2.1 Object Oriented Programming

At this point you should be comfortable with writing some common VBA instructions. This chapter helps you to work out how to do new things by explaining how the language works.

The Building Blocks of an Object Oriented Language

VBA is an example of an *object oriented* programming language. In plain English, this means that the language is made up of several characteristic building blocks, as shown in the table below:

Element	Description	Examples
Object	Any single "thing" or item that you can manipulate in VBA. Object is a deliberately vague term which could represent almost anything in an application; from physical things that you can interact with, to more abstract, invisible items. All objects are based on a *class*, which defines exactly how the object works.	A cell on a worksheet A chart on a slide A column in a chart A database connection
Collection	A collection is itself an object which you can manipulate in VBA. A collection is also a group of all of the objects of one specific type. Many VBA objects belong to a collection.	All open workbooks All shapes on a slide All data series in a chart
Method	An action that you can apply to an object. Method names are usually verbs, indicating that you're doing something to an object. When you write a subroutine or a function you are creating a custom method in the VBA project.	Select a worksheet Copy a cell Save a presentation
Property	An attribute of an object which you can often change to another value. Some properties are read-only, meaning that you can't alter them. You can write your own properties but you tend to only do this in a class module.	The value of a cell The width of a shape The count of charts

Not all VBA instructions consist solely of objects, collections, methods and properties. The table below shows some of the other elements that aren't strictly object oriented but are still important:

Element	Description	Examples
Statement	Code that doesn't necessarily perform an action but can affect what your program does.	Dim; If; Select Case; Do Until; On Error
Function	An item which returns a value or a reference to an object when you call it.	Date; Environ; Format; Instr; MsgBox
Parameter	The name of a piece of information passed to another procedure.	Prompt; Buttons; Title
Argument	The actual value that you pass to another procedure.	This could be any value
Constant	A named item which holds an underlying numeric value.	vbRed; xlDown; vbNo
Variable	A named item which stores a value when your code runs.	Almost anything you like
Operator	A symbol used in an expression to perform an operation.	+ - / * ^ &

2.2 Objects

Objects are the key building block in an object oriented language. Most VBA instructions begin by referring to the object that you want to manipulate. You can refer to objects in a variety of ways.

Referring to Objects by Name

This is perhaps the most common technique you'll use to reference an object. Start by referencing the collection to which the object belongs, as shown in the examples below:

Regardless of which class of object you're referencing, the basic syntax of the code is the same.

The name of a collection is nearly always the plural of the type of object that it contains.

The name of an object isn't case-sensitive when you use it in this way, but it is good practice to match case.

An Excel **Range** isn't technically a collection but you can use it in the same way to refer to a specific cell or cells.

```
'The basic syntax is:
'CollectionName("ObjectName")

Workbooks("Book1.xlsm").Activate

Worksheets("Sheet1").Protect

Presentations("Pres1.pptx").Close

Documents("Doc1.docx").PrintOut

Range("A1").Copy
```

Referring to Objects by Index Number

VBA indexes (assigns a number to) each item in a collection. You can use these index numbers to refer to objects, which is useful if you can work out which number refers to which object!

Again, the syntax for using this technique is consistent across all of the objects you may want to reference.

Each collection is indexed in a different way: documents are indexed in the order in which they were opened.

Worksheets, chart sheets and generic sheets are indexed by their position from left to right in the workbook.

```
'The basic syntax is:
'CollectionName(n)

'the 1st document opened in this Word session
Documents(1).Save

'the 2nd worksheet in an Excel workbook
Worksheets(2).PrintPreview

'the 1st chart sheet in an Excel workbook
Charts(1).ExportAsFixedFormat xlTypePDF

'the 3rd worksheet or chart sheet from the left
Sheets(3).Select
```

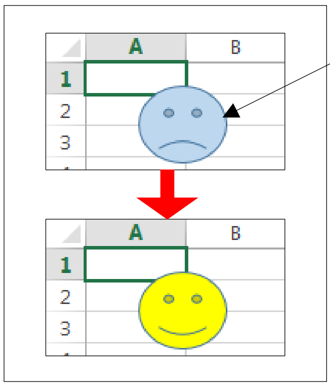
Wise Owl's Hint

*You can't use an index number with the Excel **Range** object but you can use the **Cells** property to achieve a similar result. The example below would select cell B10.*

[illegible]

Qualifying References to Objects

Some objects belong to collections which have a specific scope. The Excel **Shapes** collection, for instance, belongs to a sheet object and you can't refer to a shape without referencing the sheet.



Sadly, we can't refer to a shape in the **Shapes** collection directly because each sheet has its own separate collection of shape objects. Happily, we can refer to the sheet object first to get around this problem.

```
Sub TurnThatFrownUpsideDown()

    With Worksheets("Sheet2").Shapes("Smiley Face 1")
        .Fill.ForeColor.RGB = rgbYellow
        .Adjustments.Item(1) = 0.04653
    End With

End Sub
```

There are many other examples of objects that can only be referenced in this fashion and you can see a few of them in the table below:

Object	Code
A chart embedded on a sheet	`Worksheets("Sheet1").ChartObjects("Chart 1").Chart`
A pivot table on a worksheet	`Worksheets("Sheet3").PivotTables("PivotTable1")`
A data point in a series in a chart	`Charts("Chart1").SeriesCollection(1).Points(1)`
A shape on a slide in a presentation	`Presentations("Pres1.pptx").Slides(1).Shapes(1)`

You can qualify your references to any object in this way, even when you're not required to. This can help you to control exactly which objects your code references. For example:

Object	Code
A1 on the active sheet in the active workbook	`Range("A1")`
Cell A1 on ***Sheet1*** *in* ***Book1***	`Workbooks("Book1.xlsm").Sheets("Sheet1").Range("A1")`

Using Keywords to Reference Objects

You don't always have to refer to a collection in order to reference an object; some VBA objects don't belong to a collection. VBA has many keywords that you can use to refer to objects.

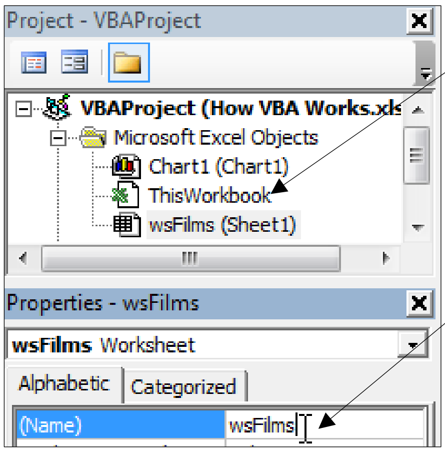
Object	Code
The active range of cells in Excel	`ActiveCell` (for a single cell) `Selection` (for multiple cells)
The active worksheet or chart	`ActiveSheet`

The active workbook, document or presentation	`ActiveWorkbook`
	`ActiveDocument`
	`ActivePresentation`

The application	`Application`

Using Object Codenames

Some objects have a *codename* as well as a name. You can usually tell if an object has a codename because it will be shown in the Project Explorer window.



If an object appears in the Project Explorer it's a good bet that it has a code name as well as a regular name.

You can use codenames in your code as a quick way to reference the object you're attempting to manipulate.

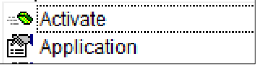
You can change the codename of an object in the Properties window to make it more meaningful.

Codenames are a very convenient way to reference objects.

```
ThisWorkbook.Save

Chart1.PrintOut

wsFilms.
```



Using Object Variables

An object variable holds a reference to an object. You can use this type of variable to make your code easier to write and understand.

You can declare an object variable to hold a reference to any class of object. In the IntelliSense list this symbol indicates that the item is a class of object.

Unlike with basic data type variables, you must use the **Set** keyword when assigning a reference to an object variable. Without it, you'll see this error:

```
Run-time error '91':

Object variable or With block variable not set
```

You can use your object variables to refer to objects and perform actions with them.

```
Dim FilmData As Range
Dim BackupSheet As Worksheet
Dim FilmChart As Chart

Set FilmData = _
    wsFilms.Range("B2").CurrentRegion
Set FilmChart = Charts.Add

FilmChart.SetSourceData Source:=FilmData

Set BackupSheet = Worksheets.Add

FilmData.Copy _
    Destination:=BackupSheet.Range("A1")

FilmChart.Location _
    Where:=xlLocationAsObject, _
    Name:=BackupSheet.Name
```

Object variables follow the same rules for scope as for data type variables. You can declare object variables at the top of a module and you can use ***Private*** *and* ***Public*** *to modify the variable's scope.*

2.3 Collections

A *Collection* is a special type of object which contains a group of all of the objects of one particular type. Many of the most common VBA objects belong to a collection.

Referring to Collections

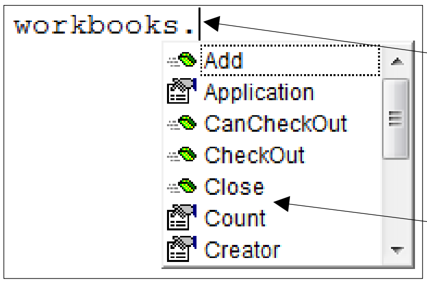
Referring to a collection object is simply a case of stating the collection's name. The table below shows examples of some of the common collections in VBA.

Collection	What it contains
`Workbooks`	All of the open Excel workbooks.
`Documents`	All of the open Word documents.
`Presentations`	All of the open PowerPoint presentations.
`Forms`	All of the running forms in an Access database.
`Worksheets`	All of the worksheets in a single workbook.
`Charts`	All of the chart sheets in a single Excel workbook.
`ChartObjects`	All of the embedded charts in a single Excel sheet.
`Slides`	All of the slides in a single PowerPoint presentation.
`Sheets`	All of the worksheets and chart sheets in a single workbook.
`Paragraphs`	All of the paragraphs in a single Word document.
`Points`	All of the data points in a single series in a chart.

You can qualify references to collections just as with other objects

Collection	What it contains
`Workbooks("Book1.xlsm").Worksheets`	All of the worksheets in **Book1**.
`Worksheets("Sheet1").ChartObjects`	All of the embedded charts in **Sheet1**.
`Worksheets("Sheet1").Shapes`	All of the drawn objects on **Sheet1**.
`Charts("Chart1").SeriesCollection`	All of the series in **Chart1**.
`Sheet1.PivotTables("PivotTable1").PivotFields`	All of the fields in a pivot table on **Sheet1**.
`Documents("Document1.docx").Paragraphs`	All of the paragraphs in **Document1**.
`Presentations("Pres1.pptx").Slides(1).Shapes`	All of the shapes on a slide in **Pres1**.

Just as with other objects, collections have a variety of methods and properties which you can use to manipulate the object.


```
workbooks.
```

To see the properties and methods of a collection, type in its name and follow it with a full stop.

This is a small selection of the methods and properties that you can apply to the **Workbooks** collection.

Adding Items to a Collection

You can add more items to many collections using the **Add** method of the collection. Each collection's **Add** method has its own list of parameters.

When you add a workbook you can optionally specify the path to a file to act as a template for the new book.

When you add a worksheet you can control where the new sheet will be inserted, as well as specifying the quantity of sheets to create.

You can control the size and position of an embedded chart when you add it to the collection using these four parameters.

```
Workbooks.Add _
    Template:="C:\Wise Owl.xltm"

Worksheets.Add _
    After:=Worksheets("Sheet1"), _
    Count:=3

Sheet1.ChartObjects.Add _
    Left:=20, _
    Top:=20, _
    Width:=200, _
    Height:=100
```

It's often useful to store a reference to the new object in a variable when you create it. This makes it easier to refer back to the object later in a procedure.

```
Dim wbBackup As Workbook

set wbBackup = _
    Workbooks.Add(
          Add([Template]) As Workbook
```

The **Workbooks.Add** method returns a reference to the workbook that is created, meaning that we can use the method to store a reference to the new workbook in an object variable.

```
Dim wbBackup As Workbook

Sheets("Sheet1").Range("B2:D8").Copy

Set wbBackup = _
    Workbooks.Add("C:\Wise Owl.xltm")

ActiveCell.PasteSpecial

wbBackup.SaveAs "C:\Film backup.xlsx"
wbBackup.Close
```

We can use the object variable later in the same procedure when we need to do something with the new workbook.

Counting Items in a Collection

It's often useful to find out how many items belong to a collection. You can use the collection's **Count** property to do this.

```
Debug.Print "Total sheets = " & Sheets.Count
Debug.Print "Worksheets = " & Worksheets.Count
Debug.Print "Charts = " & Charts.Count
```

```
Total sheets = 5
Worksheets = 3
Charts = 2
```

In a slightly more useful example, this code creates a new worksheet and positions it to the right of all the existing sheets in the workbook:

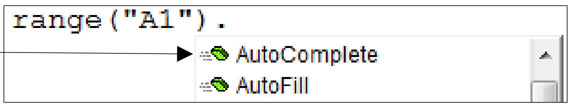
```
Worksheets.Add _
    After:=Sheets(Sheets.Count)
```

If there are already five sheets in the workbook, this code is the same as saying add a new sheet to the right of the 5th.

2.4 Methods

A *Method* is an action that you can apply to an object. You can recognise methods by the distinctive “flying green brick” symbol in the IntelliSense list.

This symbol next to a keyword in the IntelliSense list indicates that you’re looking at a method.



```
range("A1").
```

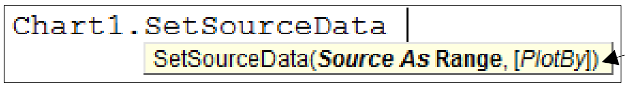
Applying Methods to Objects

Applying a method to an object is relatively straightforward: start by referring to the object, followed by a full stop and then the name of the method. The table below shows a few basic examples:

Method	What it does
`Range(“A1”).Select`	Selects the specified range object.
`Worksheets(“Sheet1”).Delete`	Deletes the specified worksheet object.
`Workbooks.Add`	Creates a new blank workbook.
`Columns(“C”).AutoFit`	Changes the width of the specified column to fit its widest entry.
`ActiveDocument.PrintOut`	Prints the active document.
`Presentations(1).Save`	Saves the first opened presentation in this PowerPoint session.

Passing Arguments to Methods

Many methods have a set of parameters to which you can pass arguments. The tooltip for a method shows you if there are any required or optional parameters.



```
Chart1.SetSourceData
```

This method has two parameters: **Source** is required; **PlotBy** is optional (it’s listed in square brackets).

```
Chart1.SetSourceData Worksheets("Sheet1").Range("B2:C7"), xlColumns
```

If we don’t pass a reference to a range of cells into the **Source** parameter, the **SetSourceData** method won’t work.

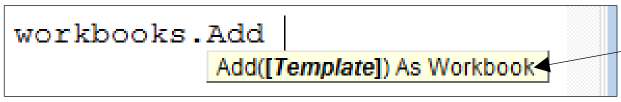
It can be useful to name a parameter when you pass an argument to it as this makes it easier to read your code.

Named parameters make your code more readable.

```
Chart1.SetSourceData _
    Source:=Worksheets("Sheet1").Range("B2:C7"), _
    PlotBy:=xlColumns
```

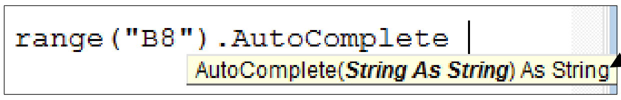
Returning Values and References from Methods

Some methods return either a value or a reference to an object. You can usually tell if a method has a return type by reading the tooltip.



```
workbooks.Add |
          Add([Template]) As Workbook
```

The **Add** method of the **Workbooks** object returns a reference to the workbook that has been created.



```
range("B8").AutoComplete |
     AutoComplete(String As String) As String
```

The **AutoComplete** method of a **Range** object returns a string representing the first matching item in the list of autocomplete entries.

To make use of the value or reference returned by a method you could choose to store it somewhere. A sensible place to do this is a variable of the appropriate type.

```
Dim FilmName As String
Dim FilmCell As Range

FilmName = InputBox("Enter a film name")

Set FilmCell = _
    Sheet1.Columns("B").Find(FilmName)

If FilmCell Is Nothing Then
    MsgBox "Nothing was found"
    Exit Sub
End If
```

The **Find** method returns a reference to the first **Range** object in which the value you're looking for was found.

We're storing the result of the **Find** method in an object variable called **FilmCell**.

We can test the contents of the variable later in the procedure to determine what to do next.

Rather than storing the reference to an object that a method returns, you can make use of it by applying further methods or properties to it instead.

```
Charts.Add.SetSourceData _
    Source:=Sheet1.Range("B2:C7")
```

The **Charts.Add** method returns a reference to a chart, so we can apply any chart method or property to it.

When to use Parentheses

The tooltip for a method always shows parentheses (round brackets) around the parameter list but you don't always use them in your code. The diagram below attempts to explain when you should!

```
Workbooks.Add Template:="C:\WiseOwl.xltm"
```

When you're simply applying a method to perform an action, as here, you don't use parentheses around your argument list.

```
Dim NewFile As Workbook

Set NewFile = _
    Workbooks.Add("C:\WiseOwl.xltm")
```

In this example we're returning the result of the **Add** method and storing it in a variable. Because we're returning a result from the method we need to use parentheses.

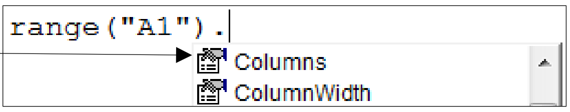
```
Workbooks.Add("C:\WiseOwl.xltm").Protect
```

You also need to use parentheses when you apply another method or property to the result of the method, as we have here.

2.5 Properties

A *Property* is an attribute of an object that you can look at and, in some cases, change to another value. You can spot properties in the IntelliSense list with their "pointy finger" icon.

This symbol next to a keyword in the IntelliSense list indicates that you're looking at a property.


```
range("A1").
```

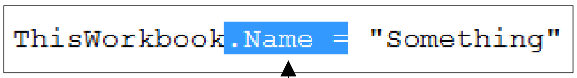
Writing to a Property

Changing the value of a property is called *writing* to it. You do this by assigning a value to the property using the = operator. You can see some examples in the table below:

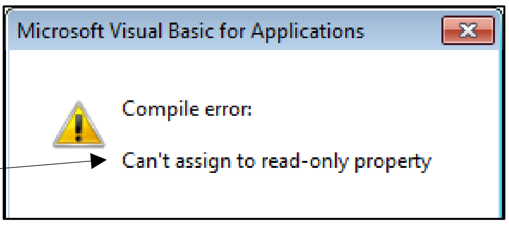
Property	**What it does**
`Range("A1").Value = 123`	Changes the information stored in the cell.
`ActiveSheet.Name = "Backup"`	Changes the name of the current sheet.
`Rows(3).Hidden = True`	Hides row 3 of the currently active worksheet.
`Columns("C").ColumnWidth = 15`	Changes the width of column C on the active sheet.
`ActiveDocument.Paragraphs(1).Alignment = _ wdAlignParagraphCenter`	Centre-aligns the text of the first paragraph in the active document.
`Presentations(1).Slides(1).Shapes(1).Width = 50`	Changes the width of a shape on a slide.

Read-Only Properties

Some properties are *read-only*, meaning you can't assign a value to them. You can't spot read-only properties in the IntelliSense list, but you'll see an error if you try to assign a value to one.


```
ThisWorkbook.Name = "Something"
```

You can't change the name of a workbook using the **Name** property, as this message so politely informs you.

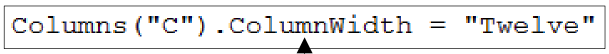

Microsoft Visual Basic for Applications

Compile error:

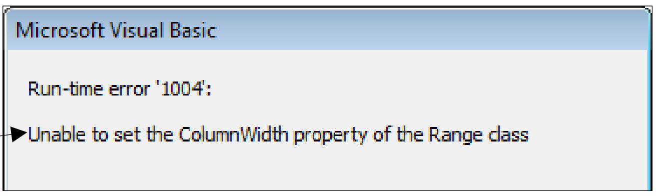
Can't assign to read-only property

Property Data Types

You should take care to assign the correct type of data to a property. In the example below, the **ColumnWidth** property can only accept a number but we're attempting to assign a string to it:


```
Columns("C").ColumnWidth = "Twelve"
```

The exact error message that you'll see will depend on which property you've tried to change. This one is fairly descriptive.


Microsoft Visual Basic

Run-time error '1004':

Unable to set the ColumnWidth property of the Range class

Reading from a Property

You *read* from a property when you look at its value. Reading a property returns either a value or a reference to an object, which you can store or make use of in some other way.

```
Dim FilmName As String
Dim FilmGross As Double

FilmName = InputBox("Enter a film name")

Columns("B").Find(FilmName).Select
FilmName = ActiveCell.Value

ActiveCell.Offset(0, 1).Select
FilmGross = ActiveCell.Value

MsgBox FilmName & " made $" & FilmGross
```

This line reads the **Value** property of a cell into a variable, storing it for later use.

Offset is a property of a cell which returns a reference to a **Range** object.

Here we apply the **Select** method to the cell returned by the **Offset** property.

Properties and Parameters

Just as with methods, many properties have a list of parameters. You can use the tooltips to find out if a property has any parameters.

```
activecell.End(
            End(Direction As XlDirection) As Range
```

The **End** property of a range has a single parameter called **Direction**, which is non-optional.

```
activecell.Address(
            Address([RowAbsolute], [ColumnAbsolute], [ReferenceStyle As XlReferenceStyle = xlA1], [External],
            [RelativeTo]) As String
```

The **Address** property of a cell has five parameters but they're all optional (each one is shown in square brackets).

You can pass arguments to the parameters of a property in the same way as for a method. The rules on whether to use parentheses are also the same.

```
ActiveCell.End(xlDown).Select
```

We're applying the **Select** method to the range returned by the **End** property so we need to use parentheses to enclose the argument list.

```
Dim CurrentCellRef As String

CurrentCellRef = _
    ActiveCell.Address( _
        RowAbsolute:=False, _
        ColumnAbsolute:=False)
```

Here we're naming two parameters of the **Address** property and returning its result to a string variable.

2.6 Getting Help in VBA

You have several choices for getting help when writing your VBA code.

The Object Browser

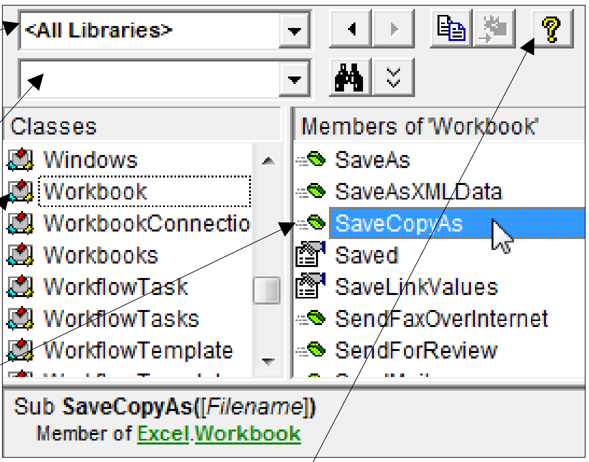
The *Object Browser* is VBA's built-in dictionary which contains definitions for each VBA keyword. To display it, choose **View | Object Browser** from the menu, or press F2 on the keyboard.

Use this list to choose which *library* you want to look in. The option shown here will give you the biggest choice of words.

You can search for a keyword by typing it here and clicking the binoculars button. The results appear in a new panel just below the search box.

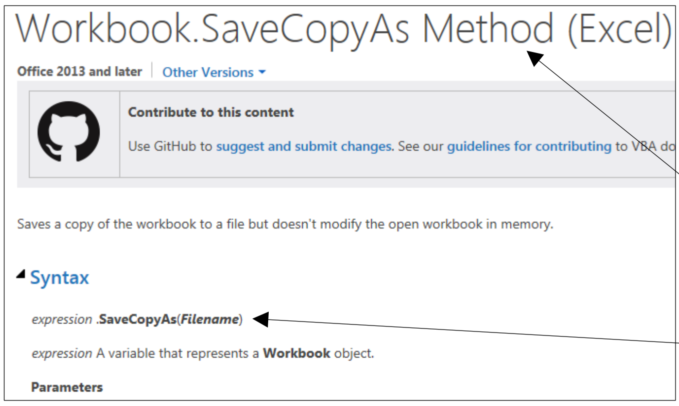
Rather than searching, it's often easier to look up keywords alphabetically, just like in a dictionary! Start by selecting the class of object you're interested in from the list on the left.

When you've chosen a class, use the list on the right to find and select the property or method you want help on.



Sub **SaveCopyAs**([*Filename*])
Member of Excel.Workbook

You'll see the syntax of the keyword at the bottom of the screen. For further help, click the question mark icon or right-click the item and choose **Help**.



Workbook.SaveCopyAs Method (Excel)

Office 2013 and later | Other Versions

Contribute to this content

Use GitHub to suggest and submit changes. See our guidelines for contributing to VBA do

Saves a copy of the workbook to a file but doesn't modify the open workbook in memory.

Syntax

expression .**SaveCopyAs**(***Filename***)

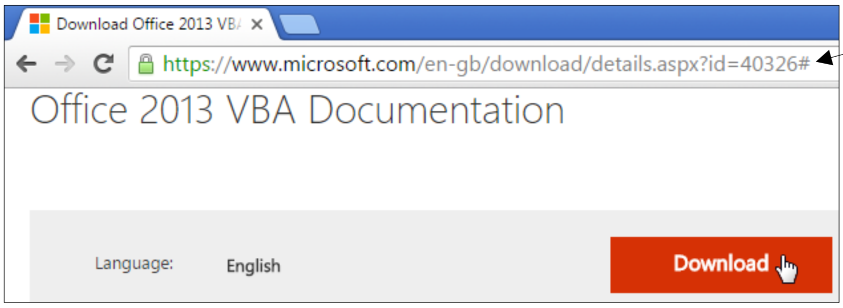
expression A variable that represents a **Workbook** object.

Parameters

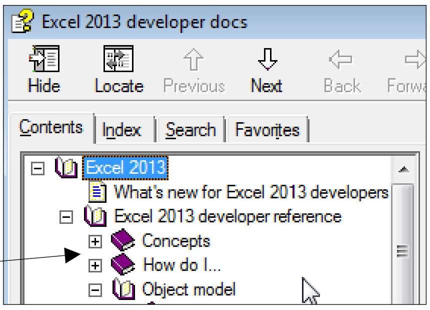
If you choose to view help on a keyword you'll be taken to a page resembling this one in your default web browser.

The page is part of Microsoft's Developer Network (MSDN) site and provides details on the keyword you've chosen to get help on.

In Office 2013 Microsoft, ironically unhelpfully, moved the VBA help files to an online system. You can still find local copies of the help files with a web search for "VBA offline help".


Download Office 2013 VB/
https://www.microsoft.com/en-gb/download/details.aspx?id=40326#
Office 2013 VBA Documentation
Language:
English
Download

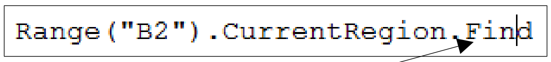
Use this page to download the local copies of the VBA help files.



Sadly, the offline help files don't integrate with the VBE. Instead, you must browse the documentation in a separate, slightly ugly application.

Context Sensitive Help

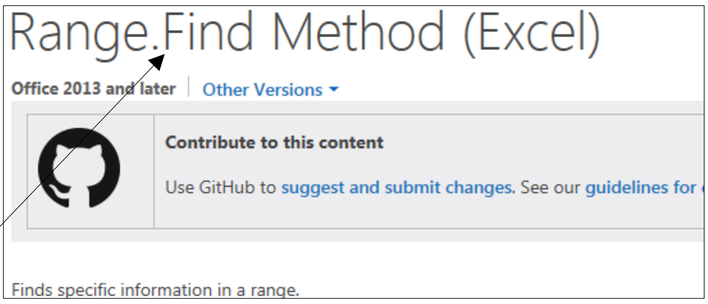
Rather than navigating through the Object Browser, you can quickly get help on a specific keyword by clicking on it in your code and pressing F1 on the keyboard.



```
Range("B2").CurrentRegion.Find
```

Position the flashing text cursor somewhere on the keyword you help with and press F1.

You should be taken to the relevant online help page, although this isn't always successful!



Range.Find Method (Excel)

Office 2013 and later | Other Versions

Contribute to this content

Use GitHub to suggest and submit changes. See our guidelines for

Finds specific information in a range.

Recording a Macro

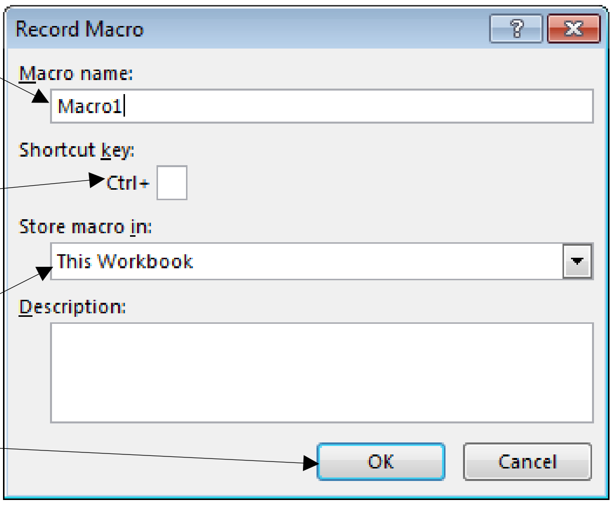
When you record a macro, the VBE automatically writes out the VBA instructions for the actions that you perform. To record a macro, choose **Developer | Record Macro** from the Excel ribbon.

You can give the macro a different name to its default one, but as we're only using this code to get help it's not really worth doing.

There's not much point in assigning a shortcut key to run the macro later either.

Storing the macro in this workbook will automatically create a new module for the recorded code.

Click OK when you want to start recording. Then you just have to perform the actions that you want Excel to write the code for.



Record Macro

Macro name: Macro1

Shortcut key: Ctrl+

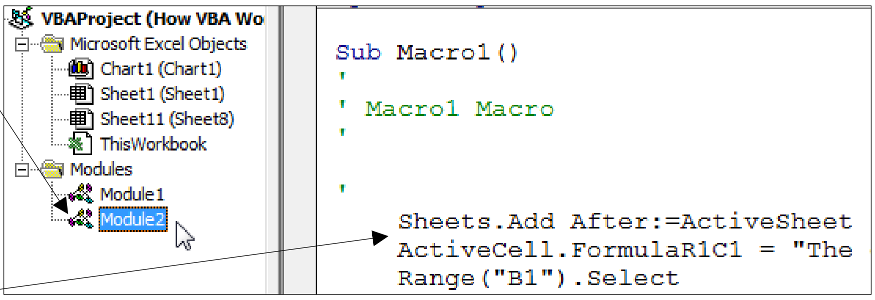
Store macro in: This Workbook

Description:

When you've finished performing actions, you can stop recording by choosing **Developer | Stop Recording** from the Excel ribbon. Now you just have to find the code you've recorded.

You should find a new module in the project you recorded the macro in.

Double-click the module to see the code in it. You should find a macro which contains code for each action you performed while recording.



```
Sub Macro1()
'
' Macro1 Macro
'

'
    Sheets.Add After:=ActiveSheet
    ActiveCell.FormulaR1C1 = "The
    Range("B1").Select
```

What we do!

		Basic training	Advanced training	Systems / consultancy
Office	Microsoft Excel	✓	✓	✓
	VBA macros	✓	✓	✓
	Office Scripts	✓		
	Microsoft Access			✓
Power BI, etc	Power BI and DAX	✓	✓	✓
	Power Apps	✓		
	Power Automate (both)	✓	✓	
SQL Server	SQL	✓	✓	✓
	Reporting Services	✓	✓	✓
	Report Builder	✓	✓	✓
	Integration Services	✓	✓	✓
	Analysis Services	✓		
Coding and AI	Visual C#	✓	✓	✓
	VB programming			✓
	AI tools	✓		
	Python	✓	✓	✓

Wise Owl Training