



# VBA Advanced

Sample manual - first two chapters



**Wise Owl**  
Training

## TABLE OF CONTENTS (1 of 6)

1	VBA RECAP	Page
1.1	VBA Reference	8
	<i>Creating Procedures</i>	8
	<i>Selecting and Activating Excel Objects</i>	8
	<i>Selecting an Excel Range Relatively</i>	9
	<i>Messages and Inputs</i>	9
	<i>Declaring Variables</i>	10
	<i>Conditional Statements</i>	11
	<i>Looping</i>	12
	<i>Exiting from a Loop</i>	12

2	OBJECT ORIENTED PROGRAMMING	Page
2.1	Object Oriented Programming	13
	<i>The Building Blocks of an Object Oriented Language</i>	13
2.2	Objects	14
	<i>Referring to Objects by Name</i>	14
	<i>Referring to Objects by Index Number</i>	14
	<i>Qualifying References to Objects</i>	15
	<i>Using Keywords to Reference Objects</i>	15
	<i>Using Object Codenames</i>	16
	<i>Using Object Variables</i>	16
2.3	Collections	17
	<i>Referring to Collections</i>	17
	<i>Adding Items to a Collection</i>	18
	<i>Counting Items in a Collection</i>	18
2.4	Methods	19
	<i>Applying Methods to Objects</i>	19
	<i>Passing Arguments to Methods</i>	19
	<i>Returning Values and References from Methods</i>	20
	<i>When to use Parentheses</i>	20
2.5	Properties	21
	<i>Writing to a Property</i>	21
	<i>Read-Only Properties</i>	21
	<i>Property Data Types</i>	21
	<i>Reading from a Property</i>	22
	<i>Properties and Parameters</i>	22
2.6	Getting Help in VBA	23
	<i>The Object Browser</i>	23
	<i>Context Sensitive Help</i>	24
	<i>Recording a Macro</i>	24

3	FOR EACH LOOPS	Page
3.1	Looping Through Collections	25
	<i>The For Each Loop</i>	25
	<i>A Basic Example</i>	25
3.2	Looping Over Excel Worksheets, Charts and Sheets	26
	<i>Protecting all Worksheets</i>	26
	<i>Excluding Objects from a Loop</i>	26
	<i>Looping Through Chart Sheets</i>	27
	<i>Looping Through All Sheets</i>	27
	<i>Looping Through Objects on a Sheet</i>	27
3.3	Looping Over the Excel Workbooks Collection	28
	<i>Processing all Open Workbooks</i>	28
3.4	Looping Over a Collection of Excel Range Objects	29
	<i>Specifying the Range to Loop Over</i>	29
	<i>Looping Through a Column of Data</i>	29
3.5	Nesting For Each Loops	30
	<i>Looping Over Shapes on All Worksheets</i>	30
	<i>Looping Through Sheets in All Open Workbooks</i>	30
3.6	Looping Over Collections in Word	31
	<i>Looping Over Documents</i>	31
	<i>Looping Over Paragraphs</i>	31
	<i>Looping Over Sentences, Words and Characters</i>	32
3.7	Looping Over Collections in PowerPoint	33
	<i>Looping Over Presentations</i>	33
	<i>Looping Over Slides</i>	33
	<i>Looping Over Objects on a Slide</i>	33
3.8	Looping Over Collections in Access	34
	<i>Looping Over Controls on a Form</i>	34
	<i>Looping Over Access Database Objects</i>	35
3.9	Looping Over Collections in Outlook	36
	<i>Looping Over Inbox Items</i>	36
	<i>Looping Over Email Attachments</i>	36

## TABLE OF CONTENTS (2 of 6)

4	CONTROLLING OTHER APPLICATIONS	Page	5	CONNECTING TO DATABASES	Page
4.1	Referencing Object Libraries	37	5.1	ActiveX Data Objects	54
	<i>Setting a Reference to an Object Library</i>	37		<i>A Brief Version History</i>	54
	<i>The Default References</i>	38		<i>Referencing the ADO Library</i>	54
	<i>References and the Object Browser</i>	38	5.2	Connecting to an External Database	55
	<i>Microsoft Office Version Numbers</i>	38		<i>Setting the Connection String</i>	55
4.2	An Example for Word	39	5.3	Creating Connections in Access	56
	<i>Setting a Reference to the Word Object Library</i>	39		<i>Referencing the CurrentProject's Connection</i>	56
	<i>Declaring a Variable for Word</i>	39	5.4	ADO Recordsets	57
	<i>Creating a New Instance of Word</i>	40		<i>Creating a Recordset</i>	57
	<i>Auto-Instancing Variables</i>	40		<i>Setting the Source of the Recordset</i>	57
	<i>Showing and Activating Word</i>	41		<i>Setting the Lock Type</i>	58
	<i>Creating a New Document</i>	41		<i>Setting the Cursor Type</i>	58
	<i>Writing and Formatting Text in Word</i>	42		<i>Opening and Closing a Recordset</i>	59
	<i>Copying from Excel to Word</i>	42		<i>Copying Data into Excel</i>	59
	<i>Saving the Document and Closing Word</i>	43	5.5	Moving in a Recordset	60
	<i>The Complete Example</i>	43		<i>Moving the Cursor</i>	60
4.3	An Example for PowerPoint	44		<i>Reaching the End of a Recordset</i>	60
	<i>Setting a Reference to the PowerPoint Object Library</i>	44		<i>Looping Over a Recordset</i>	61
	<i>Opening PowerPoint and Creating a Presentation</i>	44		<i>Referring to Fields</i>	61
	<i>Creating a Title Slide</i>	44	5.6	Finding and Filtering Records	62
	<i>Copying from Excel to PowerPoint</i>	45		<i>The Find Method</i>	62
	<i>Moving and Resizing PowerPoint Objects</i>	45		<i>Repeated Finds</i>	62
	<i>Saving the Presentation and Closing PowerPoint</i>	46		<i>Applying a Filter</i>	63
	<i>The Complete Example</i>	46		<i>Removing a Filter</i>	63
4.4	An Example for Outlook	47		<i>Adding Criteria to a SQL Select Statement</i>	64
	<i>Setting a Reference to the Outlook Object Library</i>	47		<i>Creating Dynamic SQL Statements</i>	64
	<i>The Complete Example</i>	47	5.7	Modifying Data	65
4.5	Controlling Applications without References	48		<i>Adding New Records</i>	65
	<i>The CreateObject Function</i>	48		<i>Editing Existing Records</i>	65
	<i>Using Object Variables</i>	48		<i>Deleting Records</i>	65
	<i>Converting Constants to Numbers</i>	49	5.8	ADO Commands	66
	<i>Getting a Reference to a Running Application</i>	50		<i>Creating a New Command Object</i>	66
	<i>Testing the Version of an Application</i>	51		<i>Setting the Command Text</i>	66
4.6	Referencing Other VBA Projects	52		<i>Executing the Command</i>	66
	<i>Setting a Reference to a VBA Project</i>	52	5.9	Using DAO	67
	<i>Creating Excel Add-Ins</i>	53		<i>Referencing the Correct Object Library</i>	67
	<i>Loading Excel Add-Ins</i>	53		<i>Opening a Database</i>	67
				<i>Creating a Recordset</i>	67

## TABLE OF CONTENTS (3 of 6)

6	FILES AND FOLDERS	Page
6.1	Working with Files and Folders	68
	<i>The Scripting Runtime Library</i>	68
	<i>Creating a FileSystemObject</i>	68
6.2	Basic File and Folder Techniques	69
	<i>Testing if a File or Folder Exists</i>	69
	<i>Creating a Folder</i>	69
	<i>Copying and Moving Files and Folders</i>	69
	<i>Deleting Files and Folders</i>	70
	<i>Renaming Files and Folders</i>	70
	<i>Getting a Reference to a File or Folder</i>	70
6.3	Looping Over Files and Folders	71
	<i>Looping Over Files</i>	71
	<i>Looping Over Folders</i>	71
	<i>Recursively Looping Over Subfolders</i>	72
6.4	Working with Text Files	73
	<i>Creating and Writing to a Text File</i>	73
	<i>Opening a Text File</i>	73
	<i>Reading from a Text File</i>	74
6.5	Using VBA's FileSystem Methods	75
	<i>Creating Folders</i>	75
	<i>Deleting Files and Folders</i>	75
	<i>Copying Files</i>	75
	<i>Renaming Files</i>	75

7	FILE DIALOG BOXES	Page
7.1	Working with File Dialogs	76
	<i>Types of File Dialog Box</i>	76
	<i>Displaying a File Dialog Box</i>	77
	<i>Performing the Default Action</i>	77
7.2	Customising File Dialogs	78
	<i>Changing the Title and Button Name</i>	78
	<i>Setting the Initial Location</i>	78
	<i>Allowing Multiple Selections</i>	79
	<i>Creating File Filters</i>	79
7.3	Picking Files and Folders	80
	<i>Returning a File or Folder Path</i>	80
	<i>Testing Which Button was Clicked</i>	80
	<i>Dealing with Multiple Selections</i>	81
	<i>Using Multiple File Dialogs</i>	81

8	CLASS MODULES	Page
8.1	What are Class Modules?	82
	<i>Why Create Classes?</i>	82
	<i>Important Terminology</i>	83
	<i>Debugging in Class Modules</i>	83
8.2	Designing a Class	84
	<i>Our Example Film Class</i>	84
8.3	Creating a Class	85
	<i>Inserting a Class Module</i>	85
	<i>Renaming a Class Module</i>	85
	<i>Creating a New Instance of a Class</i>	85
8.4	Creating Basic Properties	86
	<i>Basic Properties</i>	86
	<i>Disadvantages of Basic Properties</i>	86
8.5	Creating Full Properties	87
	<i>Assigning a Value to a Property</i>	87
	<i>Reading a Value from a Property</i>	88
	<i>Assigning an Object to a Property</i>	88
	<i>Writing Additional Code in Properties</i>	89
	<i>Read-Only Properties</i>	89
8.6	Creating Methods	90
	<i>Writing Methods in a Class Module</i>	90
	<i>Using Class Methods</i>	90
8.7	Class Module Events	91
	<i>Creating Class Module Event Handlers</i>	91
	<i>Triggering Class Events</i>	91
8.8	Sharing Class Modules	92
	<i>Step 1 – Rename the VBA Project</i>	92
	<i>Step 2 – Make the Class Public</i>	92
	<i>Step 3 – Create a Function to Return an Instance of the Class</i>	92
	<i>Step 4 – Reference the Class Project</i>	93
	<i>Step 5 – Consume the Class</i>	93

## TABLE OF CONTENTS (4 of 6)

9	COLLECTIONS AND DICTIONARIES	Page
9.1	What are Collections?	94
	<i>Custom Collections and Dictionaries</i>	94
9.2	Untyped Collections	95
	<i>Creating a New Collection</i>	95
	<i>Adding Items to a Collection</i>	95
	<i>Adding Custom Classes to a Collection</i>	96
	<i>Referencing Collection Items</i>	96
	<i>Removing Items from a Collection</i>	96
	<i>Looping Over Collections</i>	97
9.3	Typed Collections	98
	<i>The Problem with Untyped Collections</i>	98
	<i>Creating a Collection Class</i>	98
	<i>Populating a Typed Collection</i>	99
	<i>Looping Over a Typed Collection</i>	99
	<i>Referencing Items in a Typed Collection</i>	99
9.4	Dictionaries	100
	<i>Referencing the Scripting Runtime Library</i>	100
	<i>Creating a New Dictionary</i>	100
	<i>Adding Items to a Dictionary</i>	101
	<i>Referring to Dictionary Items</i>	101
	<i>Automatically Creating Keys</i>	102
	<i>Checking if a Key Exists</i>	102
	<i>The Compare Mode</i>	103
	<i>Removing Items from a Dictionary</i>	103
	<i>Replacing Dictionary Values</i>	104
	<i>Replacing Dictionary Objects</i>	104
	<i>Looping Over Dictionaries</i>	105

10	ARRAYS	Page
10.1	Overview of Arrays	106
	<i>Viewing the Contents of Arrays</i>	106
10.2	Declaring Arrays	107
	<i>Setting the Dimensions of an Array</i>	107
	<i>Changing the Base of Arrays</i>	107
	<i>Declaring Multi-Dimensional Arrays</i>	107
10.3	Populating Arrays	108
	<i>Assigning Values to an Array</i>	108
	<i>Assigning Objects to Arrays</i>	108
10.4	Reading from Arrays	109
	<i>Referring to a Specific Element</i>	109
	<i>Looping Over an Array</i>	109
	<i>The Bounds of an Array</i>	110
	<i>Using For Each Loops</i>	110
10.5	Dynamic Arrays	111
	<i>Declaring an Empty Array</i>	111
	<i>Re-Dimensioning an Array</i>	111
	<i>Preserving the Contents of an Array</i>	111
10.6	Arrays in Excel	112
	<i>Assigning a Range to an Array</i>	112
	<i>Calculating in an Array</i>	112
	<i>Assigning an Array to a Range</i>	112

## TABLE OF CONTENTS (5 of 6)

11	MODULAR CODE, PARAMETERS AND FUNCTIONS	Page
11.1	Modular Code	113
	<i>Our Example</i>	113
11.2	Breaking a Procedure into Parts	114
	<i>Creating Module Level Variables</i>	114
	<i>Getting Input from the User</i>	114
	<i>Retrieving the Related Values</i>	115
	<i>Building and Showing a Message</i>	115
	<i>Putting it all Together</i>	115
11.3	Procedures and Parameters	116
	<i>Our Example</i>	116
	<i>Defining Parameters</i>	116
	<i>Calling a Procedure which has Parameters</i>	117
	<i>Optional Parameters</i>	117
	<i>Assigning Default Values to Parameters</i>	118
	<i>Testing for Missing Arguments</i>	118
	<i>ParamArrays</i>	118
11.4	Passing Arguments ByRef and ByVal	119
	<i>Passing Arguments by Reference</i>	119
	<i>Passing Arguments by Value</i>	120
	<i>Passing Arguments in Parentheses</i>	120
11.5	Functions vs. Subroutines	121
	<i>Returning a Value from a Function</i>	121
	<i>Returning a Reference from a Function</i>	121
	<i>Calling a Function</i>	122
	<i>Using Functions in a Worksheet</i>	122
	<i>Defining Function Parameters</i>	122
11.6	Debugging Modular Code	123
	<i>Viewing the Definition of a Procedure</i>	123
	<i>Stepping Over a Procedure Call</i>	123

12	CONSTANTS AND ENUMERATIONS	Page
12.1	Working with Constants	124
	<i>Declaring Constants</i>	124
	<i>Referencing Constants</i>	124
12.2	Enumerations	125
	<i>Declaring Enumerations</i>	125
	<i>Referencing Enumerations</i>	125
	<i>Using Enumerations as Data Types</i>	126
	<i>Converting an Enumeration to Text</i>	126
	<i>Enumerations for Colours</i>	127

13	RECURSIVE PROGRAMMING	Page
13.1	Recursive Procedures	128
	<i>Why Use Recursive Procedures?</i>	128
	<i>Viewing the Call Stack</i>	128
13.2	Organisational Hierarchies	129
	<i>Creating the Base Organisation Chart</i>	129
	<i>Recursively Adding Nodes</i>	130

14	DEBUGGING	Page
14.1	Debugging Code	131
	<i>Errors vs. Bugs</i>	131
	<i>The Debug Toolbar</i>	131
14.2	Running Code	132
	<i>Running a Procedure from Start to End</i>	132
	<i>Running a Procedure in Break Mode</i>	132
	<i>Stepping Through Code</i>	133
	<i>Changing the Next Instruction</i>	133
	<i>Editing Code in Break Mode</i>	133
14.3	Breakpoints	134
	<i>Setting and Removing Breakpoints</i>	134
	<i>The Stop Statement</i>	134
	<i>Breaking Conditionally</i>	134
14.4	The Immediate Window	135
	<i>Executing Instructions in the Immediate Window</i>	135
	<i>Asking Questions in the Immediate Window</i>	135
	<i>Printing to the Immediate Window</i>	135
14.5	The Locals Window	136
	<i>Observing Variables</i>	136
14.6	The Watch Window	137
	<i>Adding an Expression to Watch</i>	137
	<i>Types of Watch</i>	137
	<i>Adding a Quick Watch</i>	138
	<i>Editing and Removing Watches</i>	138
14.7	The Call Stack	139
	<i>Displaying the Call Stack</i>	139
	<i>Using the Call Stack</i>	139
14.8	Useful Keyboard Shortcuts	140

## TABLE OF CONTENTS (6 of 6)

15	HANDLING ERRORS	Page
15.1	Run-Time Errors in VBA	141
15.2	Error Handling in VBA	142
	<i>Identifying Potential Run-Time Errors</i>	142
	<i>The On Error Statement</i>	142
15.3	Using the On Error Statement	143
	<i>Ignoring Run-Time Errors</i>	143
	<i>Disabling an Error Handler</i>	143
15.4	Creating a Custom Error Handler	144
	<i>Redirecting Your Code</i>	144
	<i>Writing the Error-Handling Section</i>	144
	<i>Exiting a Procedure before the Error-Handling Code</i>	145
	<i>The Complete Example</i>	145
	<i>Creating Multiple Error Handlers</i>	146
15.5	Resuming After an Error	147
	<i>Resuming at the Original Line</i>	147
	<i>Resuming at the Next Line</i>	147
	<i>Resuming at a Specified Line</i>	148
	<i>Why use Resume and Not GoTo?</i>	148
15.6	The Err Object	149
	<i>Getting the Error Number and Description</i>	149
	<i>A Catch-All Approach to Error-Handling</i>	149
15.7	Raising Custom Errors	150
	<i>Raising a Custom Error</i>	151
15.8	Errors in Multiple Procedures	152
	<i>Creating a Top-Level Error Handler</i>	152

16	SHAPES	Page
16.1	Introduction to Shapes	153
	<i>The Shapes Collection</i>	153
16.2	Referring to Shapes	154
	<i>Names and Index Numbers</i>	154
	<i>Referring to a Range of Shapes</i>	154
	<i>Referring to Selected Shapes</i>	154
	<i>Referring to Newly Added Shapes</i>	155
	<i>Looping Over the Shapes Collection</i>	155
16.3	Shape Size and Position	156
	<i>Changing the Size and Position</i>	156
	<i>Sizing and Positioning Relative to Other Objects</i>	156
16.4	Adding Shapes	157
	<i>Adding a Basic AutoShape</i>	157
	<i>Labels and Textboxes</i>	157
	<i>WordArt</i>	158
	<i>Pictures</i>	158
	<i>Form Controls</i>	159
16.5	Formatting Shapes	160
	<i>Changing Shape Colours</i>	160
	<i>Colour Gradients</i>	161
	<i>Other Formatting Options</i>	162
	<i>Setting Default Shape Formats</i>	163
	<i>Copying Formats between Shapes</i>	163
	<i>Using Shape Styles</i>	163
16.6	Shape Adjustments	164
	<i>Referring to Adjustments</i>	164
	<i>Adjusting Adjustments</i>	164
16.7	Adding Text to AutoShapes	165
	<i>The TextFrame and TextFrame2 Objects</i>	165
	<i>Adding Text to a Shape</i>	165
16.8	Formatting Text in a Shape	166
	<i>Basic Font Formatting</i>	166
	<i>Changing the Colour of Text</i>	166
	<i>Formatting Part of the Text</i>	167
	<i>Aligning Text in a Shape</i>	167
	<i>Changing Text Orientation</i>	167
16.9	Connectors and Lines	168
	<i>Drawing Straight Lines</i>	168
	<i>Adding Multi-Point Lines and Curves</i>	168
	<i>Drawing Freeform Lines</i>	169
	<i>Creating Enclosed Shapes</i>	169
	<i>Connectors</i>	170

## 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	<pre>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</pre>
Declare a function	<pre>Function MyFunction() As DataType      'do something useful     'then return a value     MyFunction = Something  End Function</pre>

#### 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	<pre>Workbooks("Book1.xlsx").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</pre>
Go to a worksheet	<pre>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</pre>
Go to a range	<pre>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</pre>

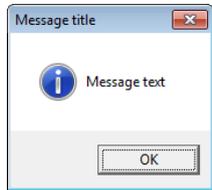
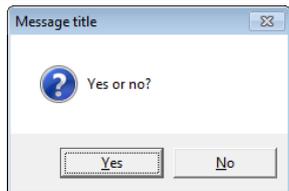
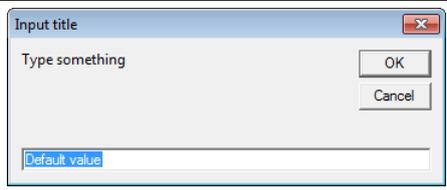
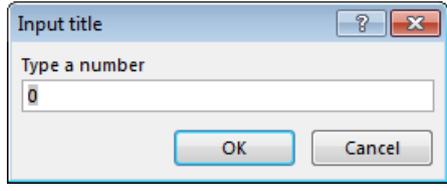
## Selecting an Excel Range Relatively

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	<pre>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</pre>
Go to the end of a list in one direction	<pre>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)</pre>
Select from one cell to the end of the list	<pre>'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</pre>

## Messages and Inputs

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

How to...	Code	Result
Show a message	<pre>MsgBox _     Prompt:="Message text", _     Buttons:=vbInformation, _     Title:="Message title"</pre>	
Ask a yes or no question	<pre>Dim Result As VbMsgBoxResult  Result = MsgBox( _     Prompt:="Yes or no?", _     Buttons:=vbQuestion + vbYesNo, _     Title:="Message title")</pre>	
Ask for a string	<pre>Dim Result As String  Result = InputBox( _     Prompt:="Type something", _     Title:="Input title", _     Default:="Default value")</pre>	
Ask for a number in Excel	<pre>Dim Result As Long  Result = Application.InputBox( _     Prompt:="Type a number", _     Title:="Input title", _     Default:=0, _     Type:=1)</pre>	

## Declaring Variables

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

How to...	Code
<i>Force explicit variable declaration</i>	<code>'add this to the top of a module Option Explicit</code>
<i>Declare data type variables</i>	<code>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</code>
<i>Assign values to data type variables</i>	<code>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"</code>
<i>Declare object variables</i>	<code>'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</code>
<i>Set a reference in an object variable</i>	<code>'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)</code>

## Conditional Statements

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

How to...	Code
<i>Write a single-line If</i>	<pre>'one logical test and one action If Range("A1").Value &lt; 0 Then Exit Sub</pre>
<i>Write a Block If</i>	<pre>'one logical test and multiple actions If Range("A1").Value &lt; 0 Then     MsgBox "No negative numbers"     Exit Sub End If</pre>
<i>Include an Else clause</i>	<pre>'one logical test with two outcomes If Range("A1").Value &lt; 0 Then     MsgBox "No negative numbers"     Exit Sub Else     MsgBox "Value is valid" End If</pre>
<i>Use ElseIf statements</i>	<pre>'multiple logical tests with multiple outcomes If Range("A1").Value &lt; 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</pre>
<i>Write a Select Case statement</i>	<pre>'conditions using SELECT CASE Select Case Range("A1").Value     Case Is &gt; 0         MsgBox "A1 is positive"     Case Is = 0         MsgBox "A1 is zero"     Case Else         MsgBox "A1 is negative" End Select</pre>

## Looping

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

How to...	Code
<i>Loop a number of times</i>	<pre>Dim Counter As Long  For Counter = 1 To 10      Cells(Counter, 1).Interior.ColorIndex = Counter  Next Counter</pre>
<i>Loop until a condition is met</i>	<pre>Range("A1").Select  Do Until ActiveCell.Value = ""      Debug.Print ActiveCell.Value     ActiveCell.Offset(1, 0).Select  Loop</pre>
<i>Loop while a condition is true</i>	<pre>Range("A1").Select  Do While ActiveCell.Value &lt;&gt; ""      Debug.Print ActiveCell.Value     ActiveCell.Offset(1, 0).Select  Loop</pre>

## 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
<i>Exit from a For Next loop</i>	<pre>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</pre>
<i>Exit from a Do Loop</i>	<pre>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</pre>

## 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
<i>Object</i>	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 <i>class</i> , 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
<i>Collection</i>	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
<i>Method</i>	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
<i>Property</i>	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
<i>Statement</i>	Code that doesn’t necessarily perform an action but can affect what your program does.	Dim; If; Select Case; Do Until; On Error
<i>Function</i>	An item which returns a value or a reference to an object when you call it.	Date; Environ; Format; Instr; MsgBox
<i>Parameter</i>	The name of a piece of information passed to another procedure.	Prompt; Buttons; Title
<i>Argument</i>	The actual value that you pass to another procedure.	This could be any value
<i>Constant</i>	A named item which holds an underlying numeric value.	vbRed; xlDown; vbNo
<i>Variable</i>	A named item which stores a value when your code runs.	Almost anything you like
<i>Operator</i>	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 basic syntax is: 'CollectionName("ObjectName")
The name of a collection is nearly always the plural of the type of object that it contains.	Workbooks("Book1.xlsm").Activate Worksheets("Sheet1").Protect
The name of an object isn't case-sensitive when you use it in this way, but it is good practice to match case.	Presentations("Pres1.pptx").Close
An Excel <b>Range</b> isn't technically a collection but you can use it in the same way to refer to a specific cell or cells.	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!

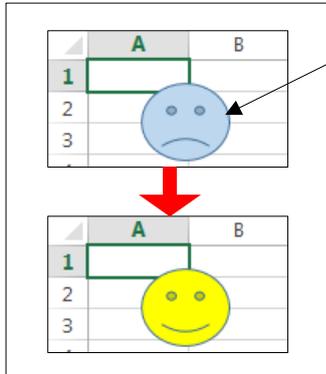
Again, the syntax for using this technique is consistent across all of the objects you may want to reference.	'The basic syntax is: 'CollectionName(n)
Each collection is indexed in a different way: documents are indexed in the order in which they were opened.	'the 1st document opened in this Word session Documents(1).Save
Worksheets, chart sheets and generic sheets are indexed by their position from left to right in the workbook.	'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



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.

## 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 <b>Sheet1</b> in <b>Book1</b>	Workbooks("Book1.xlsm").Sheets("Sheet1").Range("A1")

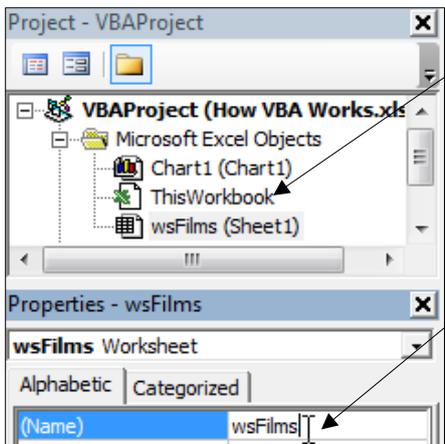
## Using Keywords to Reference Objects

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.
    Activate
    Application
    
```

## 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

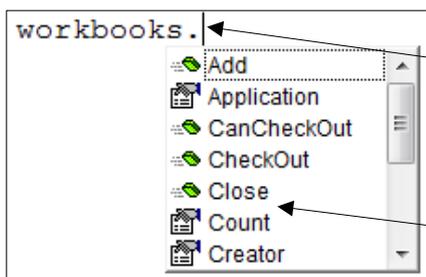
Referring to a collection object is simply a case of stating the collection's name. The table below shows examples of some of the common collections in VBA.

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.xlsx").Worksheets	All of the worksheets in <b>Book1</b> .
Worksheets ("Sheet1").ChartObjects	All of the embedded charts in <b>Sheet1</b> .
Worksheets ("Sheet1").Shapes	All of the drawn objects on <b>Sheet1</b> .
Charts ("Chart1").SeriesCollection	All of the series in <b>Chart1</b> .
Sheet1.PivotTables ("PivotTable1").PivotFields	All of the fields in a pivot table on <b>Sheet1</b> .
Documents ("Document1.docx").Paragraphs	All of the paragraphs in <b>Document1</b> .
Presentations ("Pres1.pptx").Slides(1).Shapes	All of the shapes on a slide in <b>Pres1</b> .

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



The screenshot shows the VBA Object Browser with the 'workbooks.' collection selected. A list of methods is displayed: Add, Application, CanCheckOut, CheckOut, Close, Count, and Creator. Two callout boxes provide instructions: one points to the 'workbooks.' text, and another points to the list of methods.

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 5<sup>th</sup>.

## 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").
  ▶  AutoComplete
   AutoFill
```

### 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
<code>Range("A1").Select</code>	Selects the specified range object.
<code>Worksheets("Sheet1").Delete</code>	Deletes the specified worksheet object.
<code>Workbooks.Add</code>	Creates a new blank workbook.
<code>Columns("C").AutoFit</code>	Changes the width of the specified column to fit its widest entry.
<code>ActiveDocument.PrintOut</code>	Prints the active document.
<code>Presentations(1).Save</code>	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
  SetSourceData(Source As Range, [PlotBy])
```

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.xltn"
```

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.xltn")
```

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.xltn").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").|
  Columns
  ColumnWidth
```

### 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:

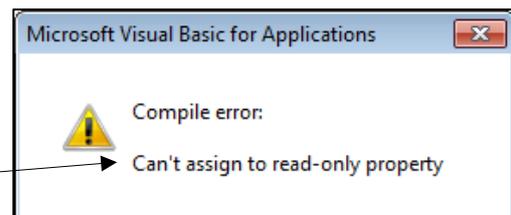
Property	What it does
<code>Range("A1").Value = 123</code>	Changes the information stored in the cell.
<code>ActiveSheet.Name = "Backup"</code>	Changes the name of the current sheet.
<code>Rows(3).Hidden = True</code>	Hides row 3 of the currently active worksheet.
<code>Columns("C").ColumnWidth = 15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

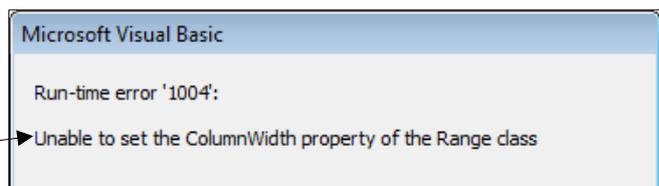


### Property Data Types

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.



## 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.

Workbooks.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 docs.

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

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**.

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 VBA... x

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.

Excel 2013 developer docs

Hide Locate Previous Next Back Forward

Contents | Index | Search | Favorites

Excel 2013

- What's new for Excel 2013 developers
- Excel 2013 developer reference
  - Concepts
  - How do I...
  - Object model

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:

Shortcut key:

Store macro in:

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.

**VBAProject (How VBA Wo**

- Microsoft Excel Objects
  - Chart1 (Chart1)
  - Sheet1 (Sheet1)
  - Sheet11 (Sheet8)
  - ThisWorkbook
- Modules
  - Module1
  - Module2

```

Sub Macro1 ()
'
' Macro1 Macro
'
'
    Sheets.Add After:=ActiveSheet
    ActiveCell.FormulaR1C1 = "The
    Range("B1").Select
        
```

# What we do!

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



