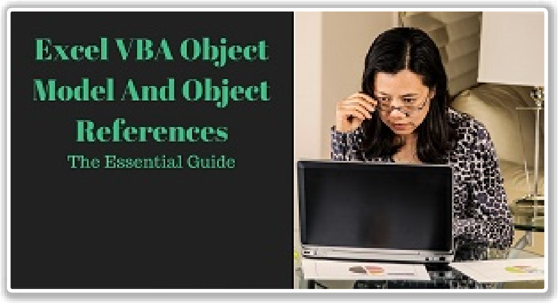
Power Spreadsheets
Become an Excel Power User

Excel VBA Object Model And Object References: The Essential Guide

By Jorge A. Gomez



If you've read any of the other macro or VBA tutorials in Power Spreadsheets, you've probably noticed that some terms keep popping up over and over.

One of the concepts that keep coming up, and will continue to come up in future tutorials, is that of objects. The main reason for this is simple:

VBA is "loosely based around the concept of Object Oriented Programming". As explained by John Walkenbach in the *Excel 2013 Bible*, this "means that it manipulates objects".

As a consequence of the above, **if you want to really master Excel macros and Visual Basic for Applications, you must have a good understanding of the following 3 topics**:

- Objects.
- How to manipulate VBA objects.
- Excel's VBA object model.

My 2 main purposes when writing this VBA tutorial are to:

- Explain the main characteristics of Excel's VBA object model.

More precisely, **in this macro tutorial I explain the following topics**:

Table of Contents [hide]

I'll say from the start that the topics of Excel's VBA object model and building VBA object references are not particularly simple. However…

Your knowledge and understanding of Excel's VBA object model and object references will improve as you continue to study, and work with, Visual Basic for Applications. Therefore, don't worry if, after reading this VBA tutorial things are not absolutely clear. **This guide should provide you with a solid base** and, with some work **I'm sure you'll master this topic and know all you need about Excel VBA objects**.

Let's begin by answering the first question that you probably have regarding the introduction I've made above by understanding…

Why Excel's VBA Object Model Is Important

Visual Basic for Applications is included in most products that are part of Microsoft Office. In addition to Excel, the list of applications that have VBA includes PowerPoint, Word and Access.

This underscores one of the great advantaged of learning VBA:

Once you know Visual Basic for Applications, you can immediately start writing macros for the other products that use VBA. In fact, you'll be able to create macros that work across all of those different applications.

> The secret to using VBA with other applications lies in understanding the *object model* for each application. VBA, after all, simply manipulates objects, and each product (Excel, Word, Access, PowerPoint, and so on) has its own unique object model.

OK. So Excel's VBA object model is clearly important. The next question you may have is…

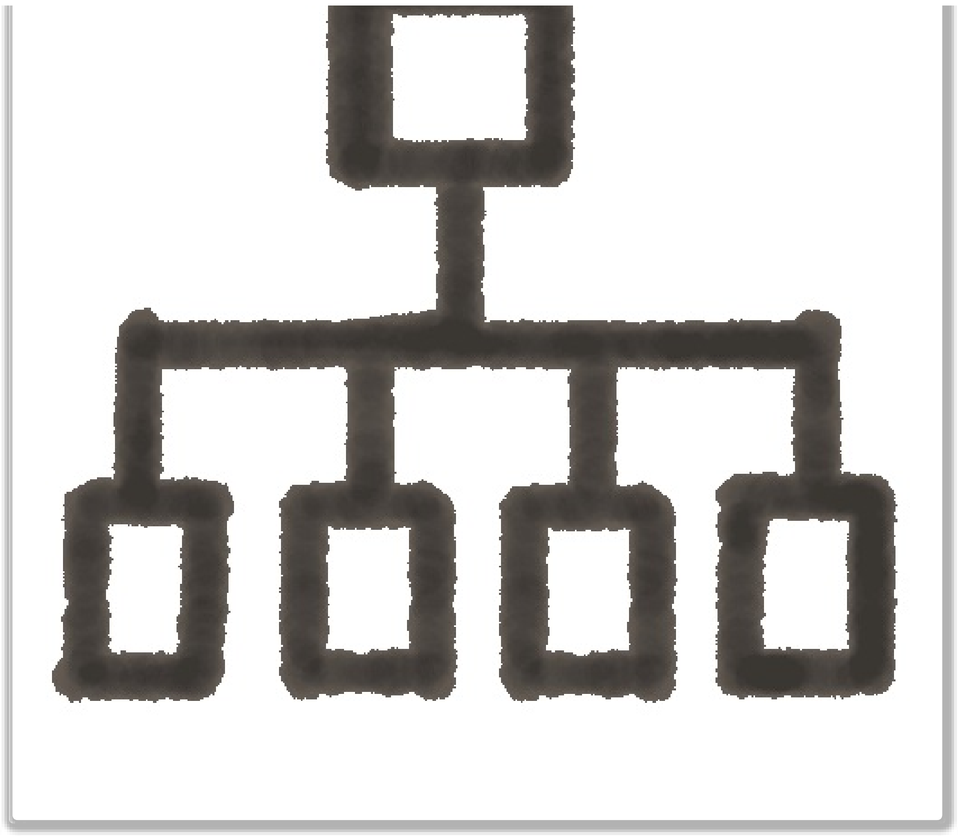
What Is Excel's VBA Object Model

I'll make this short. As explained by Excel authority Dick Kusleika:

> The object model is a big hierarchy of all the objects you can use in VBA.

Among other advantages, this hierarchy makes VBA objects easier to reference. Therefore, let's take a closer look at…

Excel's VBA Object Hierarchy



An object hierarchy looks as follows:

- **Level #1:** At the very top, you have one single object.
- **Level #2:** The object at the top of the hierarchy contains some objects.
- **Level #3:** In turn, the object in the second level of the hierarchy, may contain other objects.
- **Level #4:** The objects in level 3 may contain other objects.
- …
- You probably get the idea…

  Objects may contain other objects. The process repeats itself until you reach objects that don't contain any other objects.

When you're working with a particular software application, the first object to consider is the application itself (the Application object). Generally, the application is at the top of the hierarchy.

As noted at excel-spreadsheet.com, since Visual Basic for Applications can communicate with other applications and programs beyond Excel, this isn't strictly speaking the top level of the hierarchy. However, you'll usually see most people referring to the Application object itself as being the top of Excel's VBA object hierarchy. That's the convention I use in this macro tutorial.

The Application object contains other VBA objects. Some of the VBA objects contained by the Excel Application object are the following:

- Add-Ins, which contains all Add-In objects.
- Windows, which (at this level) contains all Window objects in the application.
- Workbooks, which contains all Workbook objects.

Each of these VBA objects, in turn, is capable of containing other objects. For example, as explained by John Walkenbach in *Excel VBA Programming for Dummies*, some of the VBA objects that can be contained within a Workbook object are the following:

- Charts, which contains Chart objects.
- Names, which contains Name objects.
- VBProjects, which represents open projects.
- Windows, which (at this level) contains Window objects in the specified Excel workbook.
- Worksheets, which contains Worksheet objects.

Again, these VBA objects can contain other objects. Continuing with Walkenbach's example, a Worksheet object can contain the following VBA objects:

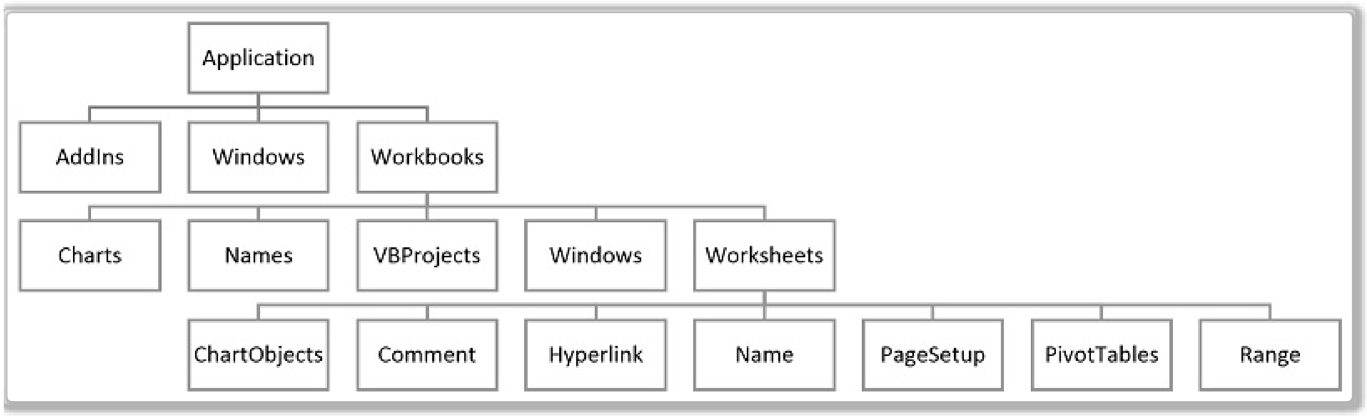
- ChartObjects, which contains ChartObject objects.
- Comment, which represents a cell comment.
- Hyperlink, which represents a hyperlink.
- Name, which represents a defined name for a particular cell range.

- PivotTables, which contains PivotTable objects.

- Range, which represents cells, rows, columns, selections of cells with contiguous blocks of cells, or 3-D ranges.

  As I explain here, the Range object is one of the most important (and most frequently used) objects.

Graphically, the portion of Excel's VBA object hierarchy described above looks roughly as follows:



The image above illustrates only **a very small portion of Excel's VBA object hierarchy**. If you want to see the huge amount of VBA objects Excel has, take a look at Excel's 2007 Object Model Map or Excel's Object Model Reference.

If this looks overwhelming, don't worry. You´re definitely not alone. Even John Walkenbach himself has written:

> Yes folks, Excel has more objects than you can shake a stick at, even old timers like me can get overwhelmed.

What can you do about this?

You may think that if an Excel guru, whose blog about Excel spreadsheets has been online since 1996 and who was recognized as a Microsoft MVP for several years in a row also gets overwhelmed by the massive amount of Excel VBA objects, the rest of us don't stand any reasonable chance of mastering Visual Basic for Applications.

Fortunately, that's not correct. You can definitely master Visual Basic for Applications despite the huge amount of Excel VBA objects. There are several reasons for this, including the following:

- If you're stuck when working on a particular problem, **you can use certain strategies for purposes of finding out which Excel VBA objects to use**. For example, in *Excel VBA Programming for Dummies*, Walkenbach suggests using the macro recorder to discover VBA objects.

Additionally, as you continue working with Visual Basic for Applications, you'll start noticing the logic behind the structure of the Excel VBA object hierarchy.

Object Collections

Collections are defined by 2 main characteristics:

- They are themselves objects.
- Their main purpose is to group and manage VBA objects of the same class.

In other words, **collections are VBA objects that are used to group and manage other objects** (which are related).

The fact you can group and manage several VBA objects by using collections is extremely useful in some situations. Imagine, for example, that you want to do something with or to a particular group of objects. If all of these objects are part of the same collection, you can structure your VBA code to go through each of the members of the collection and carry out the desired actions. As you can imagine, this structure is simpler than, for example, having to list each of the collection members individually.

In other words, **collections allow you to work with a complete group of VBA objects at the same time**, instead of working with each single object.

In *Excel VBA Programming for Dummies*, John Walkenbach lists the following examples of common collections:

- Workbooks, which is a collection of all the Excel workbooks that are currently open.
- Worksheets, the collection of all the Excel worksheets within a particular Workbook.
- Charts, which groups all chart sheets that are inside a particular Workbook.
- Sheets, which is a collection of all the sheets within a particular Workbook. In this case, it doesn't matter the type of sheet. Therefore, this collection includes both worksheets and charts sheets.

collection.

By now you probably have a firm grasp of what an object and a collection are. So let's get into the actual practice. Let's look at how you can start referencing VBA objects with Visual Basic for Applications:

Introduction To VBA Object References

Knowing how to refer to objects when writing VBA code is essential. The reason for this is that, obviously, when you want to start working with a particular VBA object, you must identify it.

The question is, *how do you do it? How do you refer to an object in Visual Basic for Applications?*

Let's take a look at some of the most common and basic situations. The purpose of this section is to serve as an introduction to VBA object references. There are many other more advanced cases. For example, I explain several ways to refer to VBA's Range object in *Excel VBA Object Model And Object References: The Essential Guide* which you can find in the Archives.

Object References: Fully Qualified References And Connecting VBA Objects

Let's start by taking a look at how to refer to an object by going through the whole hierarchy of Excel VBA objects. This is known as a fully qualified reference because you tell Excel exactly what VBA object you want to work with by referencing all of its parents.

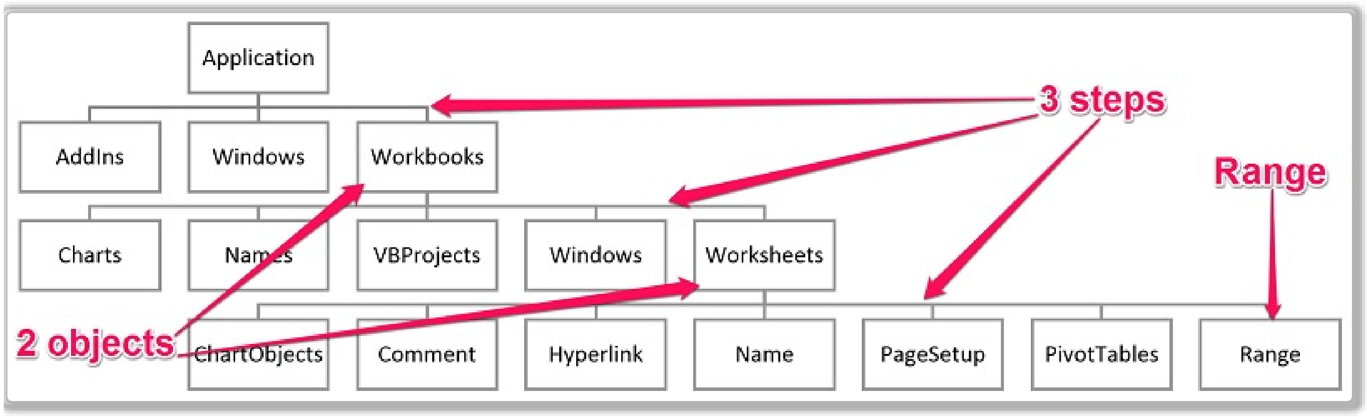
As I explain in the following sections, **you can usually simplify fully qualified references**. However, **you must learn how fully qualified references work**. They are the basis of VBA object references and, in practice, you'll use them most of the time. Additionally, they're quite useful for purposes of understanding better the VBA code behind your macros.

You already know that the object at the top of the Excel VBA object hierarchy is Application. Referring to this object is very easy. In the Visual Basic Editor, you refer to Application by typing:

> “ Application

From there on, you need to start moving along the hierarchy by using the dot (.) operator. In other words, **you connect each VBA object to the previous one (the object parent) by using a dot (.)**. As explained by Microsoft MVP Jon Acampora, those dots (.) are used to connect and reference members of Excel's VBA object model “from the top down”.

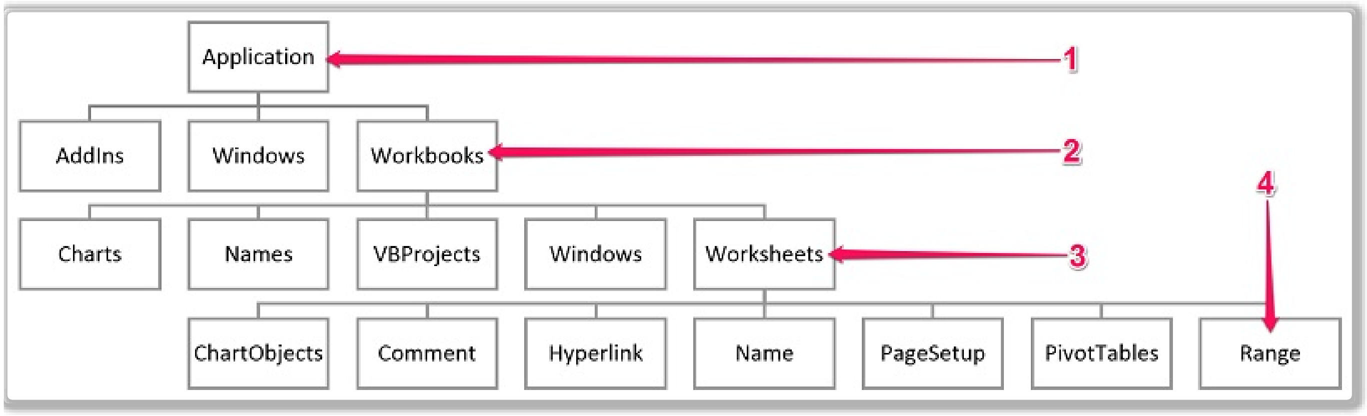
the bottom of the pyramid used in the example. There are 2 VBA objects and 3 steps between the Application and the Range object, as shown by the image below:



You already know that you simply need to connect the different objects with a dot (.) while you're going down the Excel VBA object hierarchy. In other words, you know that, in very general terms, you can refer to a Range object using the following basic structure:

> Application.Workbooks.Worksheets.Range

Graphically:



Easy, *right?*

This is, however, just a basic framework. You'll notice that this very basic structure is not actually identifying an individual VBA object. You may be wondering:

- *If there are several workbooks or worksheets how does Excel know to which one I'm referring to?*
- *How does Excel know what is the range I want to work with?*

How do you refer to a particular object within a collection?

Let's answer this question so that you can complete the fully qualified reference above.

VBA Object References: An Object From A Collection

It is likely that, **most of the time, you'll be working with a particular VBA object from a collection**. This is in contrast with the collection as a whole.

Note that **you can also work with a collection as a whole**. In fact, the ability to do this is one of the advantages of collections.

However, let's focus for now on how you can reference an object from a collection. For these purposes, you can use either of the following 2 options:

Option #1: Using The VBA Object Name.

In this case, the syntax that you must use to refer to an object is **"Collection_name("Object_name")"**. In other words:

- **#1:** The name of the relevant collection (collection_name) goes first.
- **#2:** Collection_name is followed by parentheses ().
- **#3:** Within the parentheses, you have the name of the individual VBA object (Object_name).
- **#4:** The VBA object name is within quotations ("").

  If you fail to include the quotation marks, Excel understands that the VBA object name is a variable name. Therefore, it won't be able to identify the object you want.

  In other words, **don't forget the quotations** when using this VBA object reference method.

For example, if you're working with an Excel Workbook that has 3 worksheets and you want to work with Sheet1, you can use either of the following:

> Worksheets("Sheet1")

or

Option #2: Using Index Number.

If you choose to use this option, you refer to a VBA object using **“Collection_name(Index_number)”**. This structure is **substantially the same as that above with the following 2 differences**:

- Instead of using the VBA object name, you refer to it by using its index number.
- You don’t use double quotes within the parentheses, just a number.

Going back to the example above, where you’re want to work with Sheet1, you can use either of the following 2 options:

> Worksheets(1)

or

> Sheets(1)

Now that you know how to refer to an individual VBA object within a collection, let’s go back to the fully qualified reference that I used as an example in the section above:

> Application.Workbooks.Worksheets.Range

How can you complete this, assuming that the object you want to work with is cell A1 from Worksheet Sheet1 within Workbook Book1?

If you’re using the object name to refer to each of the individual VBA objects (option #1 above), the fully qualified reference for this cell is:

> Application.Workbooks(“Book1.xlsx”).Worksheets(“Sheet1”).Range(“A1”)

As you may guess, if you had to reference each and every single object using a fully qualified reference, your VBA code would get quite long very fast. From a typing perspective, this may get a little bit annoying. Additionally, and perhaps more importantly, these very long pieces of VBA code can be difficult to read.

Simplifying Fully Qualified Object References

The ability to simplify a VBA object reference has several advantages. Mainly, this allows you to shorten your VBA code and make it easier to read.

The main reason why you can simplify fully qualified object references is because **Excel's VBA object model has some default objects**. These default objects are assumed by Excel unless you enter something different. This leads me to a very important point, which is that…

Simplifying fully qualified object references is not without dangers. In particular, the second simplification method described below relies on you correctly identifying the current active Workbook and Worksheet. If you make a mistake by for example, thinking that the current active Worksheet is Sheet1 when in reality its Sheet2, you'll face problems. The most likely issues you'll encounter in these cases are:

- Excel returns an error.
- Excel returns an erroneous result.
- Excel executes an erroneous action that you can't undo.

Another possible disadvantage of simplifying fully qualified object references is related to execution speed. This happens, for example, if you're working with a particular macro that works with several Excel worksheets. In that case, you have to go through all of them to activate them. Needless to say, this isn't very efficient.

Considering the above, ensure that you're only using these simplification methods when appropriate. Perhaps more importantly, remember that **you shouldn't blindly simplify fully qualified references all the time**.

In fact, in *Excel VBA Programming for Dummies*, John Walkenbach says that:

> It's often best to fully qualify your VBA object references.

Other Excel experts, such as Dick Kusleika, have also stated that they **generally don't rely on the default objects with a few exceptions**. One of these main exceptions, as I explain below, is relying on the Application default object. This particular VBA object is seldom included in VBA code, although there are some cases in which you must reference the Application.

- Reliability.
- Accuracy.

An alternative to the extremes of fully qualifying references or simplifying them, that both Walkenbach and Kusleika suggest, is using With…End With statements. These statements simplify macro syntax by executing several statements which refer to the same VBA object. At the same time, due to their structure, they allow you to maintain fully qualified object references.

You can see a very simple example of a With…End With statement in this macro that deletes rows based on whether a cell in a given range is blank.

Kusleika also suggests additional methods to manage your code without relying on the default objects. You can check out his suggestions here.

With the warning above in mind, let's take a look at the methods you can use to simplify fully qualified object references:

Simplification #1: The Application Object.

The **main default VBA object is the Application object**. As mentioned at dailydoseofexcel.com, **this object is always assumed** and it doesn't matter where the VBA code actually is located.

When creating macros, it is assumed that you'll be working with Excel. In other words, Excel assumes that you're working with the Application object. Therefore, as you may expect, you can generally omit this Excel VBA object from your object references.

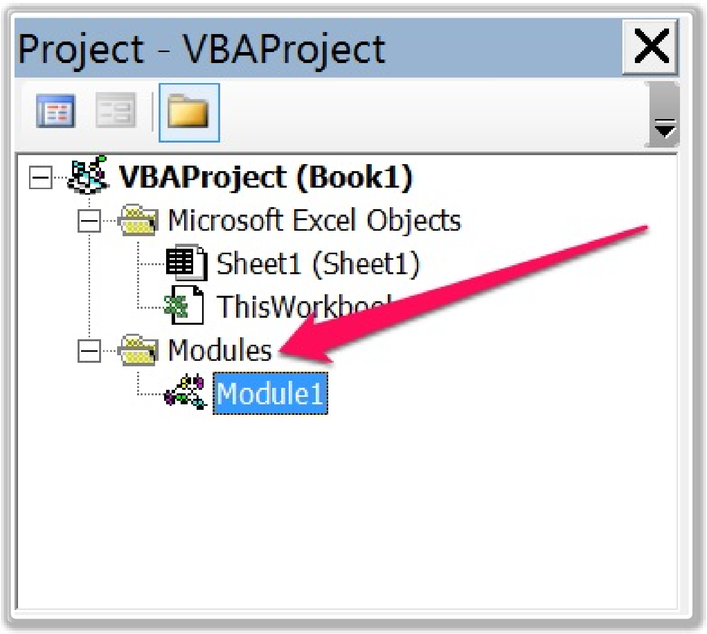
According to John Walkenbach in *Excel VBA Programming for Dummies*, **typing the Application object makes sense in only a few cases**.

Applying this shortcut to the statement referring to cell A1 in Sheet1 within Book1 that has been used as an example simplifies the reference as follows:

> Workbooks("Book1.xlsx").Worksheets("Sheet1").Range("A1")

Simplification #2: The Active Workbook and Worksheet.

modules in the Project Window under the Modules node.



In these cases, in addition to assuming that you're working with the Application object, Excel also assumes that you're working with the active Workbook.

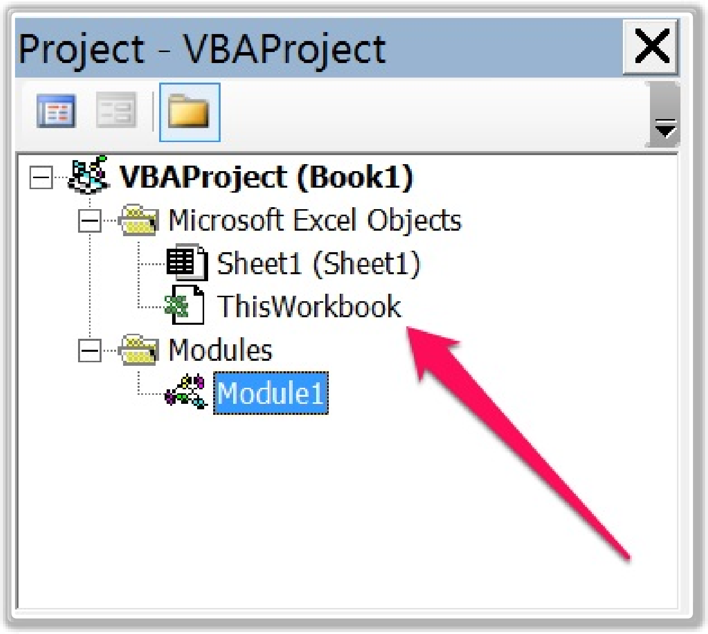
Therefore, **if you know that the current active Excel workbook is the Workbook you want to work with, you can omit that part of the VBA object reference**. Continuing with the example above, the statement can then be shortened to the following:

> Worksheets("Sheet1").Range("A1")

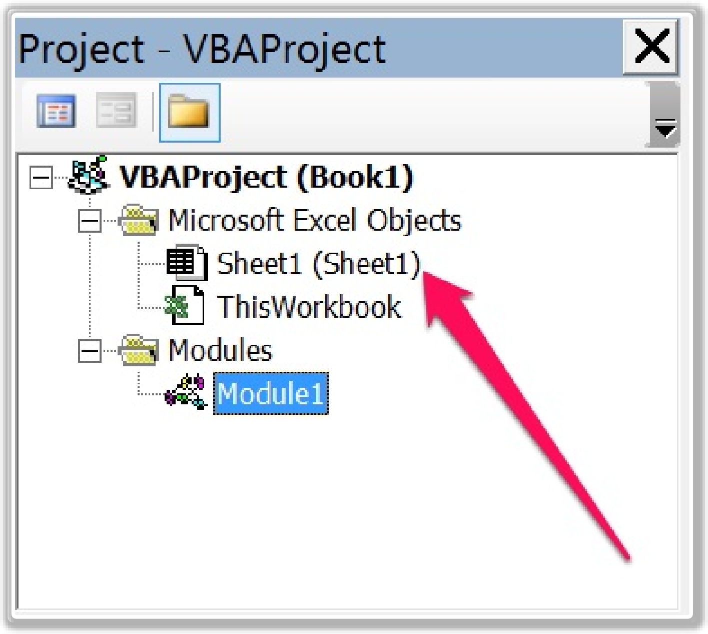
Finally, **if you're sure that the Excel worksheet you want to work with is the current active Worksheet, you can also omit that part of the VBA object reference**. The statement above can then be shortened even further:

> Range("A1")

In addition to the dangers of using this simplification that I explain at the beginning of this section, there is a further aspect you must consider. **The 2 assumptions that I've listed in Simplification #2 above only work as long as you're in a standard module**. Therefore, you must avoid relying on these assumptions when working in another type of module. For example, as explained at Daily Dose of Excel:



- If you're in a sheet module, such as the Sheet1 module, and you don't refer to the appropriate sheet, Excel's default is the sheet that contains the VBA code.



Conclusion

Excel's VBA object model is extremely important. **You can't ignore this topic if you really want to become a master in Excel macros and Visual Basic for Applications**. In the words of John Walkenbach:

Excel's VBA object model is not the simplest topic to understand but, if you practice and study, you'll eventually master the topic. Then, you'll be on your way to creating powerful macros that increase your productivity and efficiency when working with Excel.

If you've studied this particular VBA tutorial, **you not only have a good understanding of what is Excel's VBA object model, but also know how to start building object references in Visual Basic for Applications**. This ability to create appropriate VBA object references is what allows you to tell Excel which object you want to work with and manipulate. This is an essential skill that you now have in your VBA knowledge-box.

Due to the complexity and extensiveness of Excel's VBA object model, **this is a topic that we're all constantly studying and learning about**. Therefore, it would be great if you shared…

What are your favorite methods to improve your knowledge about Excel's VBA object model?

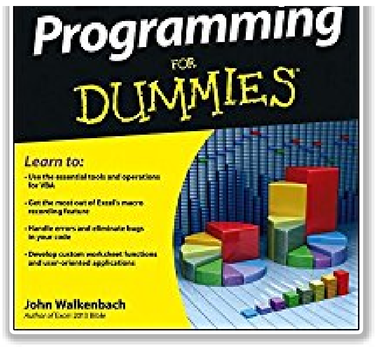
Please let us know in the comments below what are the methods that you think are most effective for purposes of improving our knowledge about Excel's VBA model. For example:

- *Reading books and blogs about Visual Basic for Applications?*
- *Using the macro recorder?*
- *Studying VBA code?*
- *Constantly exploring the Object Browser?*

I look forward to reading your ideas and suggestions!

Books Referenced In This Excel Tutorial

Click on any of the images below to purchase the book at Amazon now.



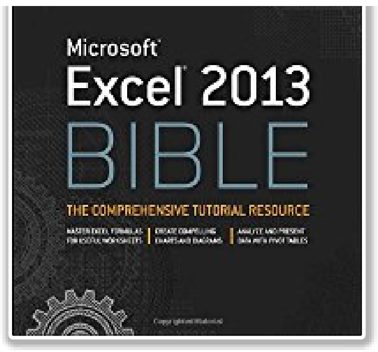


Photo: "System Maps" by Peter Morville, licensed under CC BY 2.0.

Comments for this thread are now closed. ✖

5 Comments **Power Spreadsheets** 1 **Login**

♡ **Recommend** **Share** **Sort by Newest**

Phillip Lampé • 4 months ago

I may have actually answered my question after having found the following:

"If you don't specify an object qualifier, this property returns the active sheet in the active workbook."

In the example I provided in my previous comment there is no object qualifier (Workbook), thus the ActiveSheet property returns the active sheet in the active workbook.

• Share ›

Jorge A. Gomez Mod → Phillip Lampé • 4 months ago

Hi Phillip,

Many thanks for your question (in other message) and your further comment.

As you rightly point out, the parts of the object model that deal with charts can quickly get confusing. In addition to the useful comment you made in your last post, I'll add a few further comments regarding this topic in case they're useful for future readers. I hope to eventually do more detailed work on this topic (charts).

- I agree with you regarding the ActiveSheet property: when there's no object qualifier, this property returns the active sheet in the active workbook. References in MSDN's documentation: https://msdn.microsoft.com/... and https://msdn.microsoft.com/....

- The code example you mention seems to work with the ChartObjects method (for ex., https://msdn.microsoft.com/... or https://msdn.microsoft.com/... which (in this example) returns the collection of embedded charts on the active sheet. The Add method (https://msdn.microsoft.com/... creates a new embedded chart.

- The ChartObjects collection (https://msdn.microsoft.com/... represents embedded charts. As appropriate sheet. Those are just 2 examples, as you already discussed, ActiveSheet is another option.

Thanks again for your thoughtful question/comment, Phillip! If you have further thoughts/comments/questions regarding this topic, please let me know.

^ | ∨ • Share ›

Phillip Lampé ➦ Jorge A. Gomez • 4 months ago

Thank you very much for the detailed response, Jorge. I'm new to VBA and I'm finding your posts to be extremely helpful.

Keep up the great work!

^ | ∨ • Share ›

Jorge A. Gomez Mod ➦ Phillip Lampé • 4 months ago

Many thanks for your nice comments Phillip! I appreciate them and appreciate your visits to the site.

I'm very happy to read you're finding the posts helpful. If I can be of help at any time in the future or you have further comments/feedback about the site, please let me know

^ | ∨ • Share ›

Phillip Lampé • 4 months ago

I have a question regarding the application of an object's property with an object 'below' it i regards to the Object Model's hierarchy concept.

Example:
Dim myChart As CharObject
Set myChart = ActiveSheet.ChartObjects.Add(100,50, 200, 200)

ActiveSheet is a property of Workbook. How is it that it can be used here in conjunction with an object that is below Workbook (ChartObjects)? How is it that the ActiveSheet property can be placed before ChartObjects?

^ | ∨ • Share ›

Excel Power Users

Receive FREE updates about new Tutorials and FREE resources that will help you become an Excel Power User.

Enter your email here...

BECOME AN EXCEL POWER USER

POWER SPREADSHEETS IN SOCIAL MEDIA

Excel Resources | Excel Shortcuts

SECURITY MONITORED BY Gravityscan™ 2017-09-30

Contact