



# Power Spreadsheets

Become an Excel Power User

## Define Variables In VBA: How To Declare Variables And Assign Them Expressions

By Jorge A. Gomez

When thinking about the main purpose of Visual Basic for Applications, I usually go to the following quote by one of my favorite Excel authors, John Walkenbach. In *Excel VBA Programming for Dummies*, Walkenbach explains that:

“ VBA's main purpose is to manipulate data.

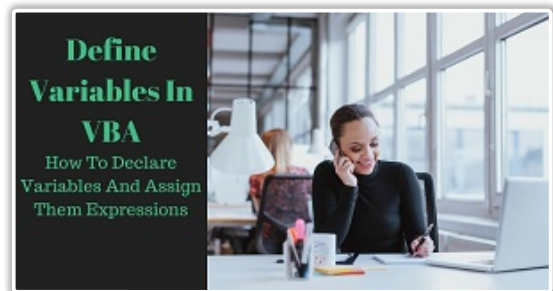
This data is generally stored in the computer's memory. For purposes of this Excel guide, we'll distinguish between the following 2 places to store data:

- Objects.
- Variables.

The former topic (objects) is covered in other Excel tutorials within Power Spreadsheets, including [here](#) and [here](#). This blog post focuses on the latter topic (VBA variables).

You'll notice that variables are often used when working with Visual Basic for Applications. In fact, as explained in *Mastering VBA for Microsoft Office 2013*:

“ Variables are used in nearly all computer programs, even short programs like macros.



When it comes to VBA variables, poor coding practices can lead to potential problems down the road. These problems can lead to big headaches.

My purpose with this Excel tutorial is to help you avoid those headaches. Therefore, in this blog post, I **explain the most important aspects surrounding VBA variable declaration**. More specifically, this guide is divided in the following sections:

### Table of Contents [hide]

What Is A VBA Variable

Why Should You Declare Variables Explicitly In VBA

    The Disadvantage Of Declaring VBA Variables Explicitly

How To Remember To Declare Variables

How To Remember To Declare Variables Always

How To Declare A Variable In VBA

    How To Declare A VBA Variable Using The Dim Statement

    How To Declare A Variable Using The Static Statement

    How To Declare A VBA Variable Using The Private And Public Statements

How To Determine The Scope Of A VBA Variable

    Variable Scope #1: Procedure-Level (Or Local) VBA Variables

    Variable Scope #2: Module-Level (Also Known As Private) VBA Variables

    Variable Scope #3: Public VBA Variables

What Is The Life Of A VBA Variable And How Is It Determined

How To Name VBA Variables

How To Assign A Value Or Expression To A VBA Variable

How To Declare VBA Variables Implicitly (a.k.a. How To Create Variables Without Declaring Them)

Conclusion

What are your best practices when naming VBA variables?

Books Referenced In This Excel Tutorial

Let's start by taking a detailed look at...

## What Is A VBA Variable

For purposes of this blog post, is enough to remember that **a variable is, broadly, a storage location paired with a name and used to represent a particular value**.

As a consequence of this, variables are a great way of storing and manipulating data.

variable.

1. Is a storage location within the computer's memory.
2. Is used by a program.
3. Has a name.

In turn, these 3 main characteristics of variables provide a good idea of what you need to understand in order to be able to declare variables appropriately in VBA:

1. How do you determine the way in which the data is stored, and how do you actually store the data.

These items relate to the topics of data types (which I cover in [a separate post](#)) and VBA variable declaration (which I explain below).

2. How do you determine which program, or part of a program, can use a variable.

This refers to the topics of variable scope and life.

3. How do you name VBA variables.

I explain each of these 3 aspects below.

**This Excel tutorial doesn't cover the topic of object variables.** This is [a different type of VBA variable](#), which serves as a substitute for an object. I may, however, cover that particular topic in a future blog post.

Since this how-to guide focuses on declaring variables in VBA, and we've already seen what a variable is, let's take a look at...

## Why Should You Declare Variables Explicitly In VBA

Before I explain some of the most common reasons to support the idea that you should declare your VBA variables explicitly when working with Visual Basic for Applications, let's start by understanding **what we're actually doing when declaring a variable explicitly**. As explained at Excel VBA website [GlobaliConnect.com](#):

“ Declaring a variable is telling the computer to reserve space in memory for later use.

There are a few reasons for this, but the **strongest has to do with Excel VBA data types**. The following are the main points that explain why, from these perspective, you should get used to declaring variables when working in Visual Basic for Applications:

- VBA data types determine the way in which data is stored in the computer's memory.
- You can determine the data type of a particular VBA variable when declaring it. However, you can also get away with not declaring it and allowing Visual Basic for Applications to handle the details.
- If you don't declare the data type for a VBA variable, Visual Basic for Applications uses Variant (the default data type).
- Despite being **more flexible than other data types**, and quite powerful/useful in certain circumstances, relying always on Variant has some downsides. Some of these potential problems include inefficient memory use and slower execution of VBA applications.

**When you declare variables in VBA, you tell Visual Basic for Applications what is the data type of the variable.** You're no longer relying on Variant all the time and, as explained in *Excel VBA Programming for Dummies*:

“ Declaring your variables makes your macro run faster and use memory more efficiently.

Even though the above is probably the main reason why you should always declare variables in Visual Basic for Applications, it isn't the only one. Let's take a look at **other reasons why you should declare variables** when working with Visual Basic for Applications:

## Additional Reason #1: Declaring Variables Allows You To Use The AutoComplete Feature

Let's take a look at the following piece of VBA code. This is the first section of a VBA Sub procedure (named Variable\_Test) suggested as example by **Microsoft**. For illustration purposes, I've changed the name of the variables originally used.

---

```
Dim Variable_Two As Integer

Variable_One = 10
Variable_Two = 100
```


You can see the whole VBA code of the Variable\_Test macro further below. Additionally, **you can get immediate access to the Excel workbooks that accompany this Excel tutorial and contains that macro, for free, by [clicking here](#).**

The next statements in the macro (after those that appear above) are:

```
" MsgBox "the value of Variable One is " & Variable_One & _  
  
Chr(13) & "the value of Variable Two is " & Variable_Two
```

Let's imagine that you're typing these statements and are currently at the following point:

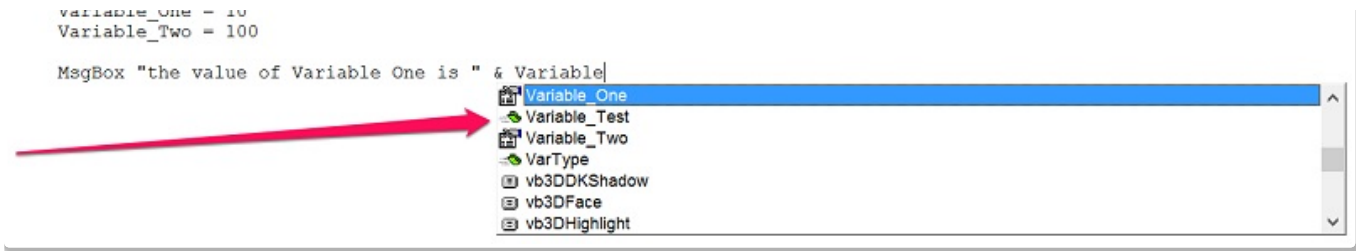
```
Sub Variable_Test()  
  
    Dim Variable_One As Integer  
    Dim Variable_Two As Integer  
  
    Variable_One = 10  
    Variable_Two = 100  
  
    MsgBox "the value of Variable One is " & Variable
```



Theoretically, you must type the whole name of the variable: "Variable\_One". However, declaring the VBA variable previously allows you to **take advantage of a keyboard shortcut**: "Ctrl + Space". Once you've typed the first few (can be as little as 2 or 3) letters of a declared variable name, you can press "Ctrl + Space" and the Visual Basic Editor does one of the following:

- As a general rule, the VBE completes the entry.
- If there is more than 1 option available, the Visual Basic Editor displays a list of possible matches to complete the entry.

In the case above, using the "Ctrl + Space" shortcut results in the Visual Basic Editor displaying a list of possible matches to complete the entry. This is shown in the following image.



Notice how the list above, in addition to Variable\_One and Variable\_Two, also includes other reserved words and functions.

As explained by John Walkenbach in *Excel VBA Programming for Dummies*, this is because the same keyboard shortcut ("Ctrl + Space") also works for those 2 other elements. However, those are topics for future blog posts.

Now, let's assume that you pressed the "Ctrl + Space" keyboard shortcut at the following point:

```
Sub Variable_Test()  
  
    Dim Variable_One As Integer  
    Dim Variable_Two As Integer  
  
    Variable_One = 10  
    Variable_Two = 100  
  
    MsgBox "the value of Variable One is " & Variable_O
```

A red arrow points from the right side of the code block towards the end of the line "MsgBox 'the value of Variable One is ' & Variable\_O", indicating the position where the keyboard shortcut was used.

In this case, the Visual Basic Editor completes the entry.

```
Sub Variable_Test()  
  
    Dim Variable_One As Integer  
    Dim Variable_Two As Integer  
  
    Variable_One = 10  
    Variable_Two = 100  
  
    MsgBox "the value of Variable One is " & Variable_One
```

A red arrow points from the right side of the code block towards the end of the line "MsgBox 'the value of Variable One is ' & Variable\_One", indicating the position where the keyboard shortcut was used.

## Additional Reason #2: Declaring Variables Makes VBA Carry Out Additional Error Checking

By declaring your VBA variables and defining their data type, you get some additional help from Visual Basic for Applications. The reason for this, as explained by John Walkenbach in *Excel 2013 Power Programming with VBA*, is that when you declare the data type of your VBA variables explicitly, “VBA can perform some additional error checking at the compile stage”.

In some cases, as explained by Richard Mansfield in *Mastering VBA for Microsoft Office 2013*, VBA can actually catch some errors at design time.

Mansfield also explains that the most common type of error that VBA can catch this way is data-typing errors. These errors are caused by incorrectly assigning certain information to a particular variable. More precisely, this usually happens when the assignment results in a mismatch between the following:

- The data type of the variable; and
- The type of the data assigned to the variable.

The following are some examples of how this can occur:

- You declare an Integer or Byte variable, but try to assign a string to it.

The Integer and Byte data types are designed to store certain integers, not strings.

- You declare a Boolean variable, but try to assign a large integer to it.

Variables of the Boolean data type can only be set to the 2 Boolean values (TRUE and FALSE).

As mentioned at [ExcelFunctions.net](http://ExcelFunctions.net), declaring your VBA variables allows you to notice these kind of mistakes as soon as possible after the variable has received an unexpected data type.

## Additional Reason #3: Declaring Variables Improves Code Readability, Makes Debugging Easier And Reduces The Risk Of Certain Mistakes

As I explain below, whenever you declare VBA variables explicitly, you generally place all the variable declarations at the beginning of a module or procedure. This improves the readability of your VBA code.

Additionally, as explained in *Mastering VBA for Microsoft Office 2013*, declaring variables explicitly makes your VBA code easier to debug and edit.

In fact, **Microsoft** states that:

“ Explicitly declaring all variables reduces the incidence of naming-conflict errors and spelling mistakes.

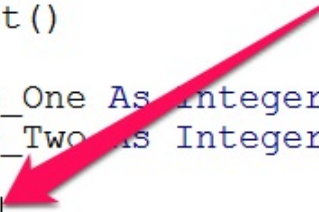
Naming-conflict errors are quite dangerous. If you don't declare VBA variables, there is a higher risk of accidentally over-writing or wiping out a pre-existing variable when trying to create a new one (as explained in *Mastering VBA for Microsoft Office 2013*).

Spelling mistakes can also create big problems. I explain one of the main ways in which taking measures to declare variables always helps you reduce spelling mistakes below.

However, in addition to that particular way, variable declaration also helps reduce spelling mistakes by capitalizing all the lower-case letters (in a variable name) that were capitalized at the point of the declaration statement.

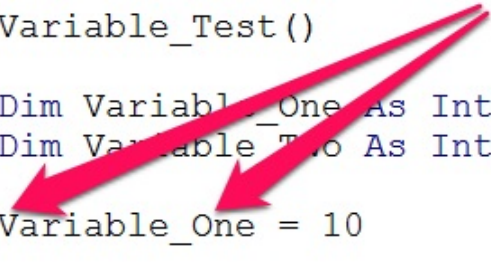
To see how this works, let's go back to the VBA code for the Variable\_Test macro. Let's assume that you type the name of Variable\_One without capitalizing (as shown below):

```
Sub Variable_Test()  
  
    Dim Variable_One As Integer  
    Dim Variable_Two As Integer  
  
    variable_one
```



Once you complete the statement, the Visual Basic Editor automatically capitalizes the appropriate letters. This looks as follows:

```
Sub Variable_Test()  
  
    Dim Variable_One As Integer  
    Dim Variable_Two As Integer  
  
    Variable_One = 10  
    |
```



You can, for example, do as **Chris Macro** and make it a habit to **(i)** use some capital letters in your VBA variable names but **(ii)** type the body of your VBA code in only lowercase letters. If the Visual Basic Editor



By the way, as explained at [ozgrid.com](http://ozgrid.com), Excel does this automatic capitalization for all keywords (not only VBA variables).

All of the advantages that I list under Additional Reason #3 are, in the end, related to the fact that variable declaration helps you (and VBA itself) have a better idea of what variables exist at any particular time. As explained by Excel blogger and Microsoft MVP [Chandoo](#):

“ (...) if you are not declaring variables, then you don't know what is available for you to use.

So I suggest you save yourself some potential problems by getting used to declaring your VBA variables explicitly.

## The Disadvantage Of Declaring VBA Variables Explicitly

Despite the advantages described above, **the idea that you should always declare VBA variables explicitly is not absolutely accepted without reservations.**

For example, in *Mastering VBA for Microsoft Office 2013*, Richard Mansfield explains how declaring variables explicitly is almost always a good idea. However, Mansfield suggests that both approaches have their pros and cons. Mansfield goes on to explain that the disadvantage of declaring variables explicitly is that this takes “a little more time, effort and thought”.

However, Mansfield concludes that, **in most cases, the disadvantage described above is far outweighed by all the other advantages that declaring variables have.**

In practice, however, it seems that **for several VBA programmers the fact that declaring variables require slightly more time and effort isn't outweighed by the advantages** described above. In fact, as explained by Bill Jelen & Tracy Syrstad in *Excel 2013 VBA and Macros*:

“ Generally, developers do not take the time to declare variables.

I provide a general explanation of how to create a VBA variable without declaring it (known as declaring a variable implicitly) below.

However, let's assume for the moment that you're convinced about the benefits of declaring VBA variables. You want to do it always: no more undeclared variables.

In order to ensure that you follow through, you may want to know...

Visual Basic for Applications has a **particular statement that forces you to declare any variables you use**. It's called the **Option Explicit statement**.

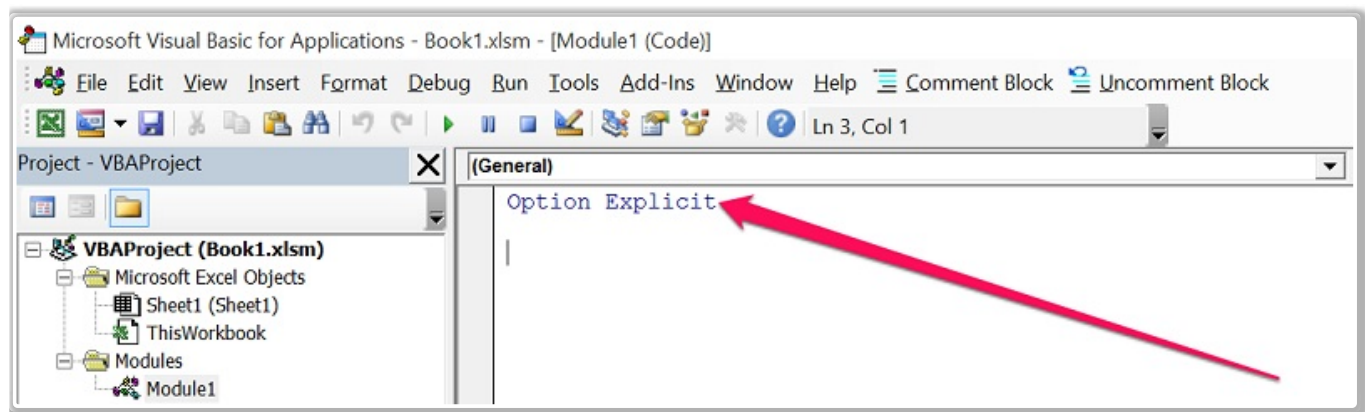
Whenever the Option Explicit statement is found within a module, **you're prevented from executing VBA code containing undeclared variables**.

The Option Explicit statement just needs to be used once per module. In other words:

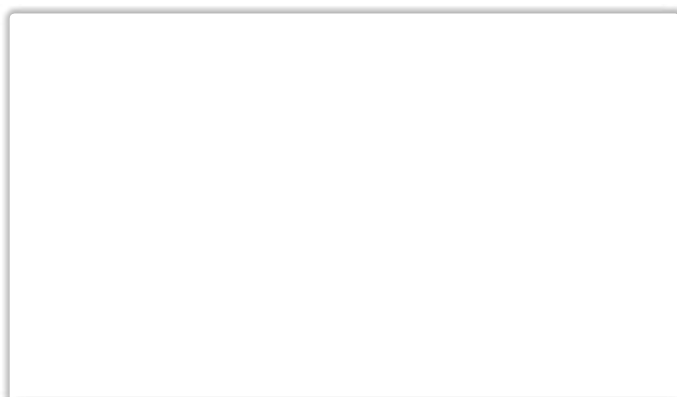
- You only include 1 Option Explicit statement per module.
- If you're working on a particular VBA project that contains more than 1 module, you must have 1 Option Explicit statement in each module.

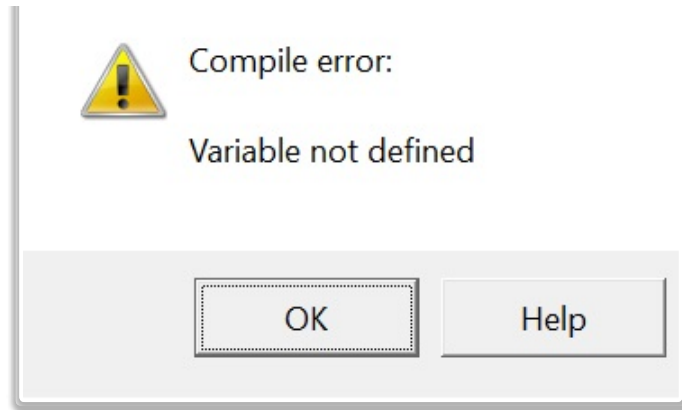
Including the Option Explicit statement in a module is quite simple:

Just **type "Option Explicit" at the beginning of the relevant module, and before declaring any procedures**. The following image shows how this looks like:



Once you've included the Option Explicit statement in a module, Visual Basic for Applications doesn't execute a procedure inside that module unless absolutely all the variables within that procedure are declared. If you try to call such procedure, Excel displays the following compile-time error message:





In addition to the reasons I provide above to support the idea that you should declare your VBA variables, there is **an additional advantage when using the Option Explicit statement and having to declare absolutely all of your variables:**

**The Visual Basic Editor automatically proofreads the spelling of your VBA variable names.**

Let's see exactly what I mean by taking a look at a practical example:

Let's go back to the Variable\_Test macro that I introduced above. The following screenshot shows the full Sub procedure:

```
Sub Variable_Test()  
  
    Dim Variable_One As Integer  
    Dim Variable_Two As Integer  
  
    Variable_One = 10  
    Variable_Two = 100  
  
    MsgBox "the value of Variable One is " & Variable_One & _  
        Chr(13) & "the value of Variable Two is " & Variable_Two  
  
End Sub
```

Notice how, at the beginning of the Sub procedure, the variables Variable\_One and Variable\_Two are declared. I explain how to declare VBA variables below.

```
Dim Variable_Two As Integer
```

```
Variable_One = 10  
Variable_Two = 100
```

```
MsgBox "the value of Variable One is " & Variable_One &  
Chr(13) & "the value of Variable Two is " & Variable_Two
```

```
End Sub
```

Notice, furthermore, how each of these variables appears 2 times after being declared:

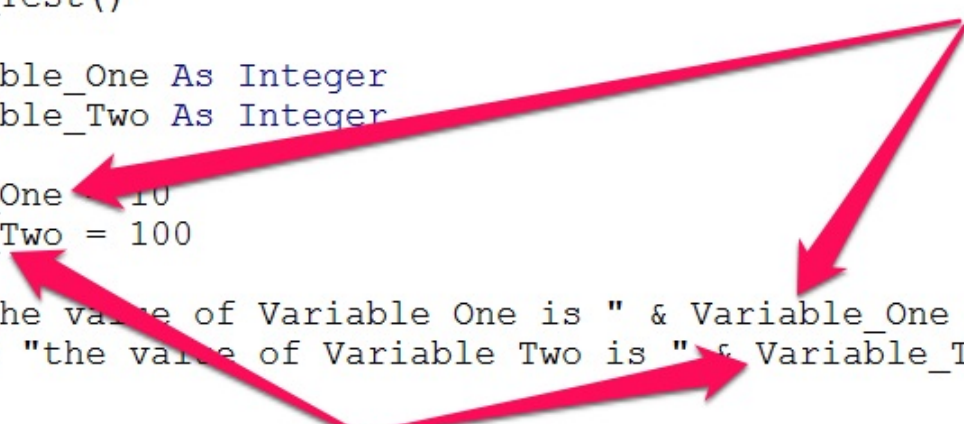
```
Sub Variable_Test()
```

```
Dim Variable_One As Integer  
Dim Variable_Two As Integer
```

```
Variable_One = 10  
Variable_Two = 100
```

```
MsgBox "the value of Variable One is " & Variable_One &  
Chr(13) & "the value of Variable Two is " & Variable_Two
```

```
End Sub
```




Let's assume that while writing the VBA code, you made a typo when writing the name of the VBA variable Variable\_One (you typed "Variable\_Ome"). In such a case, the code of the Variable\_Test macro looks as follows:

```
Sub Variable_Test()
```

```
Dim Variable_One As Integer  
Dim Variable_Two As Integer
```

```
Variable_One = 10  
Variable_Two = 100
```

```
MsgBox "the value of Variable One is " & Variable_Ome &  
Chr(13) & "the value of Variable Two is " & Variable_Two
```



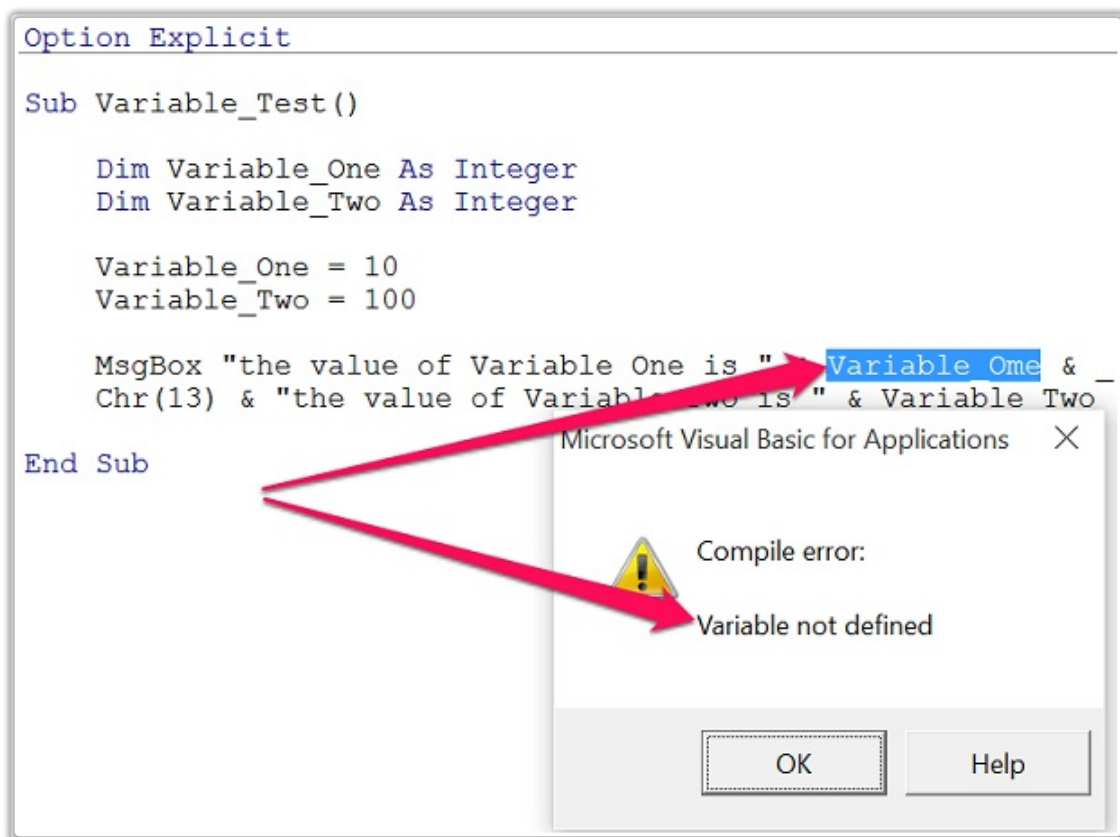
```
End Sub
```

different variables. This may be a big problem...

As explained by Richard Mansfield in *Mastering VBA for Microsoft Office 2013*, if you fail to notice these kind of typos within in the names of variables, Excel interprets the different spellings as different variables (for example, Variable\_One and Variable\_Ome). In other words, Excel creates a new variable using the misspelled name. This **leads to having several variables even though you think you only have one and, in certain cases, may end in the macro returning the wrong result.**

However, **when you misspell the name of a VBA variable while having the Option Explicit statement enabled, Excel warns you about it.** The following image shows the error message that is displayed when I try to run the Variable\_Test macro with the typo shown above. Notice how the Visual Basic Editor:

- Clearly informs you that there is an undefined variable.
- Highlights the element of the VBA code where the mistake is.



**Using the Option explicit statement is strongly suggested by most Excel authorities.** For example, [Excel authority Daniel Ferry](#) suggests that you should always start your modules with this statement.

Therefore, let's take this a step further by seeing how to ensure that the Option Explicit statement is always enabled:

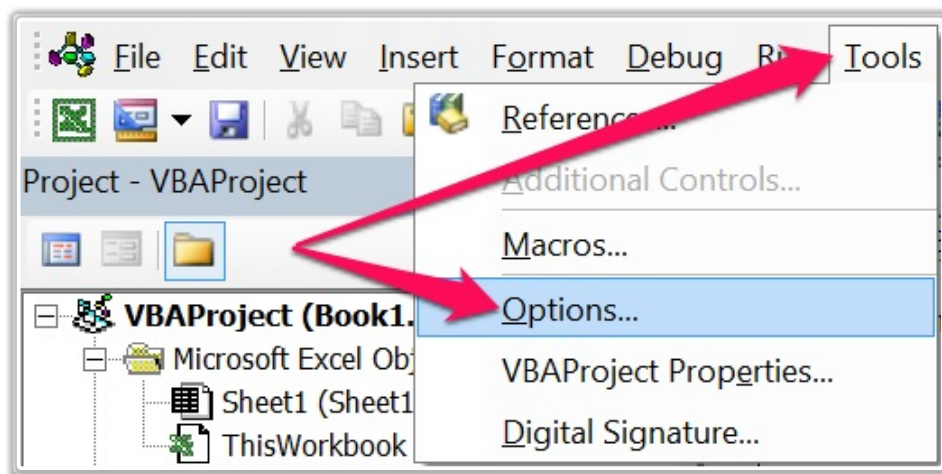
The **Option Explicit** statement is **not enabled by default**. This means that, if you want to use it always, you'd have to enable it for every single module you create.

However, the Visual Basic Editor is highly customizable. **One of the customizations you can make is have the VBE always require that you declare your variables**. The Visual Basic Editor does this by automatically inserting the Option Explicit statement at the beginning of any VBA module.

**Enabling this option is highly recommended by Excel authorities** such as John Walkenbach, Richard Mansfield, [Chris Macro](#) and [Microsoft MVP John Peltier](#). If you want to have the Visual Basic Editor insert the Option Explicit statement in all your future modules, follow these 2 easy steps:

## Step #1: Open The Options Dialog

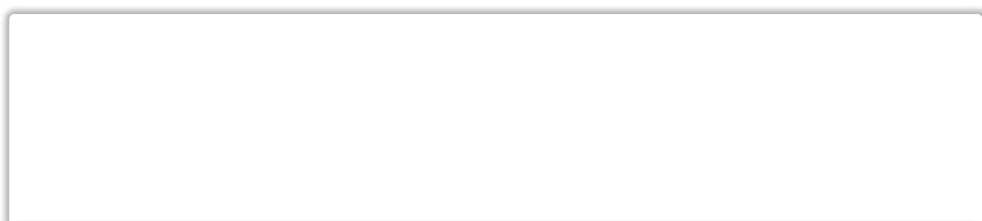
You can get to the Options dialog of the Visual Basic Editor by clicking on the Tools menu and selecting "Options...".



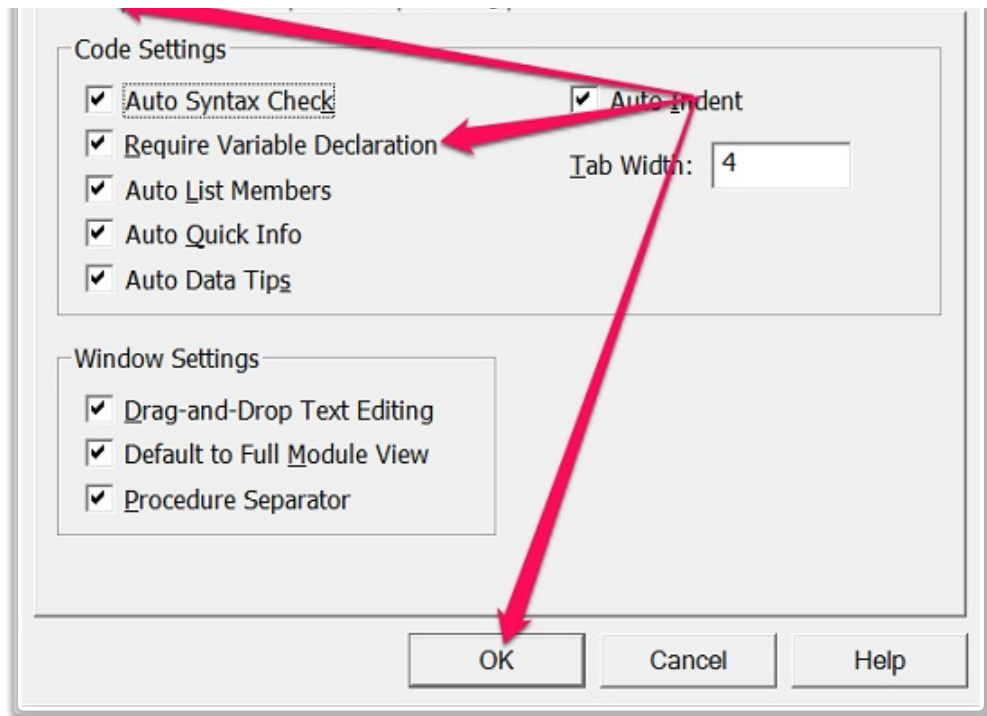
## Step #2: Enable "Require Variable Declaration" And Click On The OK Button

The Visual Basic Editor should display the Editor tab of the Options dialog by default. Otherwise, simply click on the relevant tab at the top of the dialog.

Within the Editor tab, you simply need to enable "Require Variable Declaration" by ticking the box next to it. Then, click the OK button at the lower right corner of the dialog to complete the operation.







Note that enabling Require Variable Declaration option **only applies to modules created in the future**. In other words, Require Variable Declaration doesn't insert the Option Explicit statement in previously existing modules that were created while the option wasn't enabled.

You can learn more about customizing the Visual Basic Editor through the Options dialog, including a detailed description of the Require Variable Declaration setting, by clicking [here](#).

You already know how to determine the way in which data is stored by using VBA data types. Let's take a look at how you actually store the data or, in other words...

## How To Declare A Variable In VBA

The **most common way of declaring a variable explicitly is by using the Dim statement**. As mentioned in *Mastering VBA for Microsoft Office 2013*, you'll probably want to use Dim for most of your variable declarations.

You've already seen this type of statement within the Variable\_Test macro shown above. In that particular case, the Dim statement is used to declare the Variable\_One and Variable\_Two variables as follows:

```
Dim Variable_One As Integer  
Dim Variable_Two As Integer
```

Both variables are being declared as being of the Integer data type.

1. Static.
2. Public.
3. Private.

As explained below, you generally use these 3 latter statements to declare variables with special characteristics regarding their scope or lifetime.

Regardless of the keyword that you’re using to declare a variable, the **basic structure of the declaration statement is the same**:

```
“ Keyword Variable_Name As Data_Type
```

In this structure:

- “Keyword” is any of the above keywords: Dim, Static, Public or Private.

In the following sections, I explain when is appropriate to use each of these.

- “Variable\_Name” is the name you want to assign to the variable.

I explain how to name VBA variables below.

- “Data\_Type” makes reference to the data type of the variable. This element is optional, but recommended.

The question of which is the exact keyword that you should use to declare a particular VBA variable depends, mostly, on the scope and life you want that particular variable to have.

The following table shows **the relationship between the 4 different statements you can use to declare a VBA variable and the 3 different scopes a variable can have**:

VARIABLE SCOPE	POSSIBLE KEYWORDS	LOCATION OF VARIABLE DECLARATION STATEMENT
Procedure-only	Dim or Static	Within procedure



Module-only	Dim or Private	Within module, but before any procedure
Public	Public	Within module, but before any procedure

I introduce the Dim, Static, Private and Public statements in the following sections. However, as you read their descriptions, you may want to constantly refer to the section below which covers VBA variable scope.

## How To Declare A VBA Variable Using The Dim Statement

The **Dim statement is the most common way to declare a VBA variable whose scope is procedure-level or module-level**. I cover the topic of variable scope below.

Just in case you're wondering, Dim stands for dimension. John Walkenbach explains in *Excel 2013 Power Programming with VBA* that, in the older versions of BASIC, the Dim "statement was used exclusively to **declare the dimensions for an array**".

Nowadays, in VBA, you use the Dim keyword to declare any variable, regardless of whether it is an array or not.

The image above shows the most basic way in which you can use the Dim statement to declare VBA variables. However, **you can also use a single Dim statement to declare several variables**. In such cases, you use a comma to separate the different variables.

The following screenshot shows how you can declare Variable\_One and Variable\_Two (used in the sample Variable\_Test macro) by using a single Dim statement:

```
Dim Variable_One As Integer, Variable_Two As Integer
```

As explained in *Mastering VBA for Microsoft Office 2013*, even though **this structure allows you to write fewer lines of VBA code, it makes the code less readable**. Therefore, "it's usually not a good idea".

Some advanced Excel users, such as **Excel expert Dick Kusleika**, sometimes use a single line (as shown above) to declare several related variables. He explains however, that most his variables "are on their own line". **Chip Pearson** suggests that, if you decide to use a single declaration statement for several variables, you must be particularly careful to avoid making mistakes such as the one I describe next:

Regardless of whether you declare VBA variables in separate lines or in a single line, note that **you can't declare several variables to be of a determined data type by simply separating the variables with a comma before declaring one data type**. As explained in *Excel 2013 Power Programming with VBA*:

In other words, the following statement is not the equivalent as that which appears above:

```
Dim Variable_One, Variable_Two As Integer
```

In this latter case, only Variable\_Two is being declared as an Integer.

In the case of Variable\_One, this piece of code doesn't declare the VBA data type. Therefore, Visual Basic for Applications uses the default type: Variant.

The **location of the Dim statement within a VBA module depends on the desired variable scope**. I explain the topic of variable scope, and where to place your Dim statements depending on your plans, below.

## How To Declare A Variable Using The Static Statement

You can use the **Static statement to declare procedure-level VBA variables**. It's an alternative to the Dim statement.

I explain the topic of variable scope below. However, for the moment, is enough to understand that **procedure-level VBA variables can only be used within the procedure in which they are declared**.

The **main difference between VBA variables declared using the Dim statement and variables declared using the Static statement is the moment in which the variable is reset**. Let's take a look at what this means precisely:

As a general rule (when declared with the Dim statement), all procedure-level variables are reset when the relevant procedure ends. **Static variables aren't reset; they retain the values between calls of the procedure**.

Static VBA variables are, however, reset **when you close the Excel workbook** in which they're stored. They're also reset (as I explain below) when a procedure is terminated by an End statement.

Despite the fact that Static variables retain their values after a procedure ends, this doesn't mean that their scope changes. **Static variables continue to be procedure-level variables** and, as such, they're only available within the procedure in which they're declared.

These particular characteristics of Static variables make them particularly useful for purposes of storing data regarding a process that needs to be carried out more than 1 time. There are some special scenarios where this comes in very handy. Richard Mansfield (in *Mastering VBA for Microsoft Office 2013*) and John

- Storing data for a procedure that is executed again later and uses that information.
- Keeping track (counting) of a running total.

Let's take a quick look at these 2 examples:

### Example #1: Toggling

Mansfield explains that you can use a Static variable within a procedure that "*toggles* something between 2 states".

The example provided by Mansfield is a procedure that turns *italics* on and off. Therefore, the first time the procedure is called, *italics are turned on*. In the second execution, italics are turned off. For the third call, they're *turned back on*. And so on...

### Example #2: Counters

Walkenbach explains how a Static variable can be useful to keep track of the number of times a particular procedure is executed. You can set up such a counter by, for example:

- Declaring a counter variable as a Static VBA variable.
- Increasing the value assigned to the Static variable by 1 every time the procedure is called.

## How To Declare A VBA Variable Using The Private And Public Statements

You can **use the Private statement to declare module-level variables**. In these cases, **it's an alternative to the Dim statement**. I explain the topic of module-level variables in more detail below.

The fourth and final statement you can use to declare a VBA variable is Public. The **Public statement is used to declare public variables**. I explain public variables below.

At the beginning of this Excel tutorial, we saw that one of the defining characteristics of a VBA variable is that it is used by a program. Therefore, let's take a look at how do you determine which program, or part of a program, can use a determined variable. We start doing this by taking a look at...

## How To Determine The Scope Of A VBA Variable

Understanding how to determine the scope of a variable is very important because:

- A single module can have several procedures.
- A single Excel workbook can have several modules.

If you're working with relatively simple VBA applications, you may only have one module with a few procedures. However, as your work with VBA starts to become more complex and sophisticated, this may change.

You may not want all of VBA variables to be accessible to every single module and procedure. At the same time, you want them to be available for the procedures that actually need them. Appropriately determining the scope of VBA variables allows you to do this.

Additionally, the scope of a VBA variable has important implications in connection with the other characteristics of the variable. An example of this is the life of a variable, a topic I explain below.

Therefore, let's take a look at the **3 main different scopes that you can apply**:

## Variable Scope #1: Procedure-Level (Or Local) VBA Variables

This is the most restricted variable scope.

As implied by its name, **procedure-level variables (also known as local variables) can only be used within the procedure in which they are declared**. This applies to both Sub procedures and Function procedures.

In other words, when the execution of a particular procedure ends, any procedure-level variable ceases to exist. As explained in *Excel 2013 Power Programming with VBA*, Excel "frees up the memory that the variable used".

The fact that Excel frees up the memory used by procedure-level variables when the procedure ends makes this type of VBA variables particularly efficient. Additionally, as explained in *Mastering VBA for Microsoft Office 2013*, in typical macros there's generally no reason to justify a variable having a broader scope than the procedure.

This is an important point: **as a general rule, you should make your VBA variables as local as possible**. Or, in the words of [Daniel Ferry](#):

Excel doesn't store the values assigned to procedure-level VBA variables for use or access outside the relevant procedure. Therefore, if you call the same procedure again, any previous value of the procedure-level variables is lost.

**Static variables (which I explain above) are, however, an exception to this rule.** They generally retain their values after the procedure has ended.

Let's take a look at how you declare a procedure-level variable:

As I explain above, the usual way to declare a VBA variable is by using the Dim statement. This general rule applies to procedure-level variables.

You can also use the Static statement for declaring a procedure-level variable. In such a case, you simply use the Static keyword instead of the Dim keyword when declaring the VBA variable. **You'd generally use the Static statement instead of Dim if you need the variable to retain its value once the procedure has ended.**

The key to making a variable procedure-level is in the location of the particular declaration statement. This **statement must be within a particular procedure.**

As explained in *Mastering VBA for Microsoft Office 2013*, the **only strict requirement regarding the location of the variable declaration statement within a procedure is that it is before the point you use that particular VBA variable.** Nonetheless:

“ Custom and good sense recommend declaring all your variables at the beginning of the procedure that uses them.

More precisely, the most common location of a variable declaration statement within a procedure is:

- Immediately after the declaration statement of the relevant procedure.
- Before the main body of statements of the relevant procedure.

The VBA code of the Variable\_Test macro is a good example of how you usually declare a VBA variable using the Dim statement. Notice how the Dim statement used to declare both variables (Variable\_One and Variable\_Two) is between the declaration statement of the Sub procedure and the main body of statements:



```
Dim Variable_Two As Integer
```

## 2. Dim statements

```
Variable_One = 10  
Variable_Two = 100
```

## 3. Sub procedure's code

```
MsgBox "the value of Variable One is " & Variable_One &  
Chr(13) & "the value of Variable Two is " & Variable_Two
```

```
End Sub
```

As you can imagine, placing all your VBA variable declaration statements at the beginning of a procedure makes the code more readable.

Since the scope of procedure-level VBA variables is limited to a particular procedure, you can use the same variable names in other procedures within the same Excel workbook or module. Bear in mind that, even though the name is the same, each variable is unique and its scope is limited to the relevant procedure.

I'm only explaining this for illustrative purposes, not to recommend that you repeat VBA variable names throughout your procedures. In fact, as mentioned by Richard Mansfield in *Mastering VBA for Microsoft Office 2013*, "using the same variable names in different procedures rapidly becomes confusing when debugging". As you'll see below, **a common recommendation regarding variable name choice is to use unique variable names.**

Let's see how this works in practice by going back to the module containing the Variable\_Test Sub procedure and adding a second (almost identical procedure): Variable\_Test\_Copy.

```

Dim variable_two As Integer

Variable_One = 10
Variable_Two = 100

MsgBox "the value of Variable One is " & Variable_One & _
Chr(13) & "the value of Variable Two is " & Variable_Two

End Sub

```

---

```

Sub Variable_Test_Copy()

Dim Variable_One As Integer
Dim Variable_Two As Integer

Variable_One = 20
Variable_Two = 200

MsgBox "the value of Variable One in the Copy is " & Variable_One & _
Chr(13) & "the value of Variable Two in the Copy is " & Variable_Two

End Sub

```

Notice how, the only difference between the 2 Sub procedures is in the values assigned to the variables and the text that appears in the message box:

```

Sub Variable_Test()

Dim Variable_One As Integer
Dim Variable_Two As Integer

Variable_One = 10
Variable_Two = 100

MsgBox "the value of Variable One is " & Variable_One & _
Chr(13) & "the value of Variable Two is " & Variable_Two

End Sub

```

---

```

Sub Variable_Test_Copy()

Dim Variable_One As Integer
Dim Variable_Two As Integer

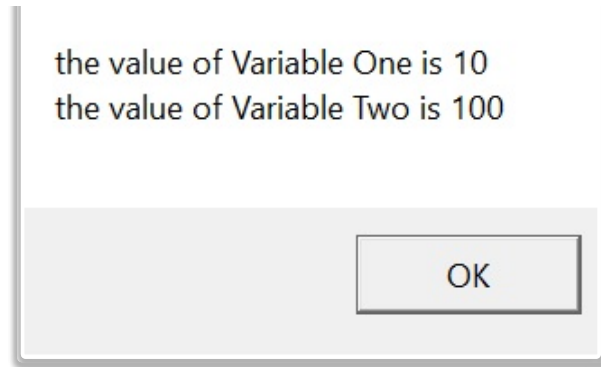
Variable_One = 20
Variable_Two = 200

MsgBox "the value of Variable One in the Copy is " & Variable_One & _
Chr(13) & "the value of Variable Two in the Copy is " & Variable_Two

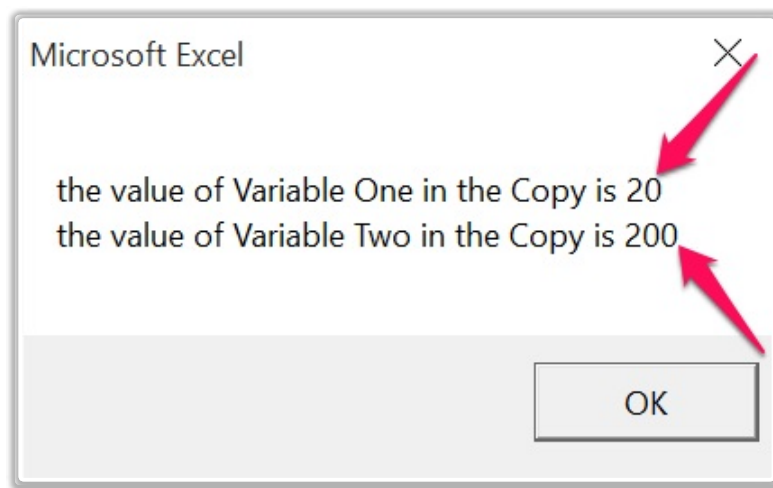
End Sub

```

If I run the Variable\_Test macro, Excel displays the following message box:



If I run the Variable\_Test\_Copy macro, Excel displays a very similar message box. Notice in particular how the values of Variable One and Variable Two change:



Let's move on to a wider variable scope:

## Variable Scope #2: Module-Level (Also Known As Private) VBA Variables

The ideas behind the concept of module-level (also known as private) VBA variables are, to a certain extent, similar to what you've already seen regarding procedure-level variables (with a few variations).

As implied by their name, **module-level variables are available for use in any procedure within the module in which the VBA variable is declared**. Therefore, module-level variables survive the termination of a particular procedure. However, as explained by [Microsoft](#), they're not available to procedures in other modules within the same VBA project.

As a consequence of the above, Excel stores the values assigned to module-level variables for use or access within the relevant module. Therefore, **module-level variables retain their values between procedures, as long as those procedures are within the same module**. As explained by Richard Mansfield



Despite the above, module-level variables don't retain their values between procedures (even if they're within the same module) if there is an End statement. As I explain below, VBA variables are purged and lose their values when VBA encounters such a statement.

The **most common way to declare module-level VBA variables is by using the Dim statement**.

Alternatively, you can use the Private statement.

As explained at [excel-easy.com](https://www.excel-easy.com), module-level variables declared using the Dim statement are by default private. In other words, at a module-level, Dim and Private are equivalent. Therefore, you can use either.

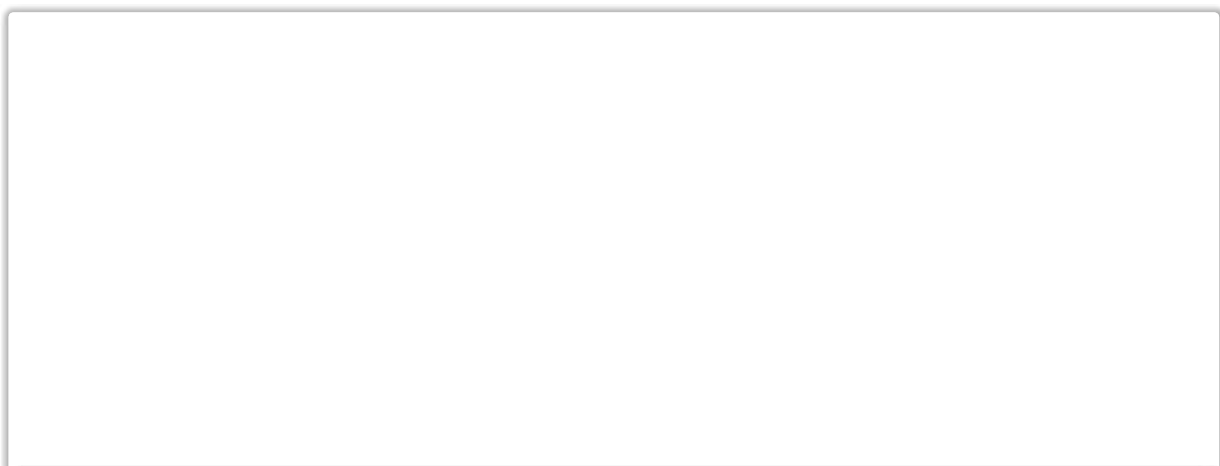
However, Richard Mansfield and [Microsoft](#) suggest that **using the Private statement can be a better alternative for declaring module-level variables**. The reason for this is that, since you'll generally use the Dim statement to declare procedure-level variables, using a different keyword (Private) to declare module-level variables makes the code more readable by allowing you to distinguish between procedure-level and module-level variables easily.

Just as is the case for procedure-level variables, the **key to determining the scope of a module-level variable is the location of the particular declaration statement**. You declare a module-level VBA variable:

- Within the module in which you want to use the variable.
- Outside of any of the procedures stored within the module.

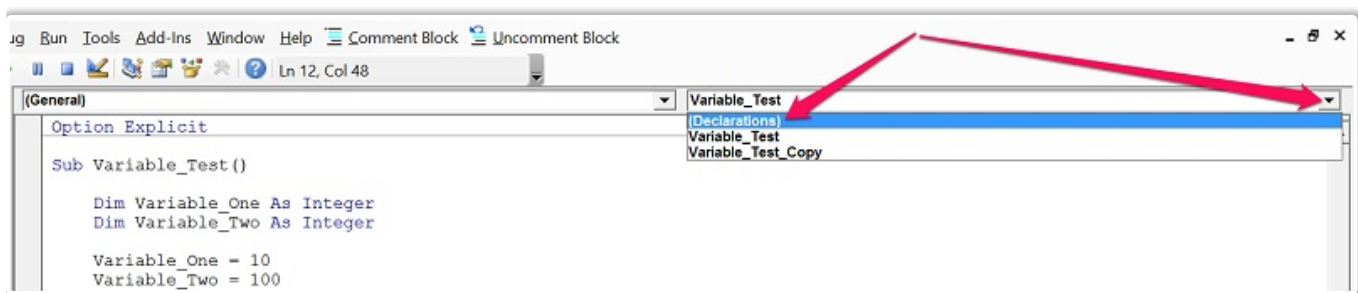
The way to achieve this is simply by **including the relevant variable declaration statements within the Declarations section** that appears at the beginning of a module.

In the case of the sample Excel workbooks that accompany this tutorial, the Declarations section is where the Option Explicit statement appears.

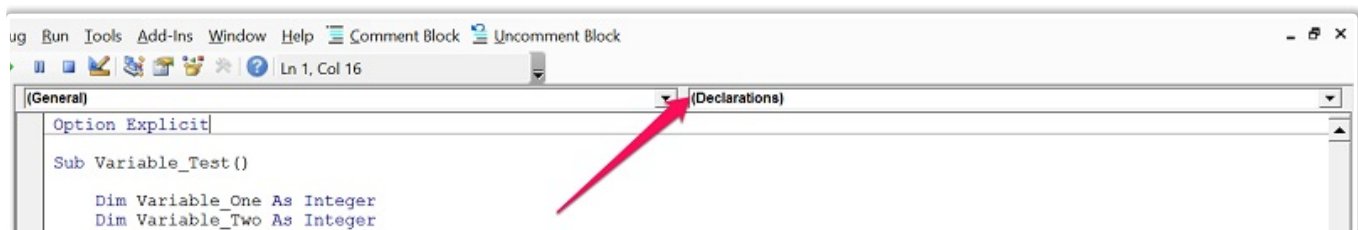


```
Sub variable_test()  
  
    Dim Variable_One As Integer  
    Dim Variable_Two As Integer  
  
    Variable_One = 10  
    Variable_Two = 100  
  
    MsgBox "the value of Variable One is " & Variable_One & _  
        Chr(13) & "the value of Variable Two is " & Variable_Two  
  
End Sub  
  
Sub Variable_Test_Copy()  
  
    Dim Variable_One As Integer  
    Dim Variable_Two As Integer  
  
    Variable_One = 20  
    Variable_Two = 200  
  
    MsgBox "the value of Variable One in the Copy is " & Variable_One & _  
        Chr(13) & "the value of Variable Two in the Copy is " & Variable_Two  
  
End Sub
```

You can, when working within any VBA module, easily get to the Declarations section within the Visual Basic Environment by clicking on the drop-down list on the top right corner of the Code Window (known as the Procedure Box) and selecting "(Declarations)".



This same drop-down list continues to display "(Declarations)" as long as you're working within the Declarations section of the relevant module.



Let's see the practical consequences of working with module-level variables by modifying the VBA code above as follows:

- Deleting the statements that assigned values to Variable\_One and Variable\_Two in the Variable\_Test\_Copy macro. The assignment made within the Variable\_Test Sub procedure is maintained.

The resulting VBA code is as follows:

```
Option Explicit

Dim Variable_One As Integer
Dim Variable_Two As Integer

Sub Variable_Test()

    Variable_One = 10
    Variable_Two = 100

    MsgBox "the value of Variable One is " & Variable_One & _
        Chr(13) & "the value of Variable Two is " & Variable_Two

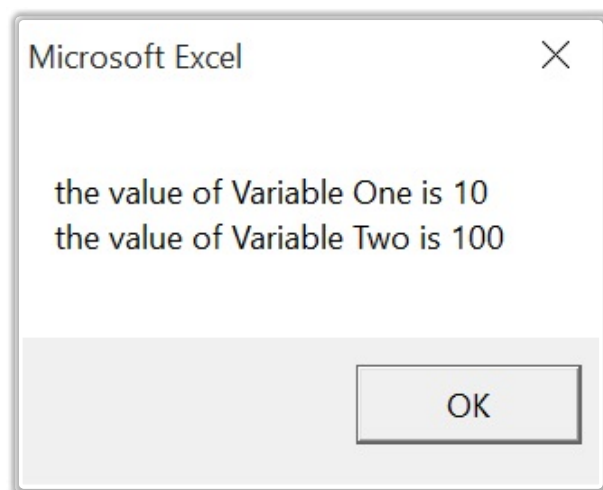
End Sub

Sub Variable_Test_Copy()

    MsgBox "the value of Variable One in the Copy is " & Variable_One & _
        Chr(13) & "the value of Variable Two in the Copy is " & Variable_Two

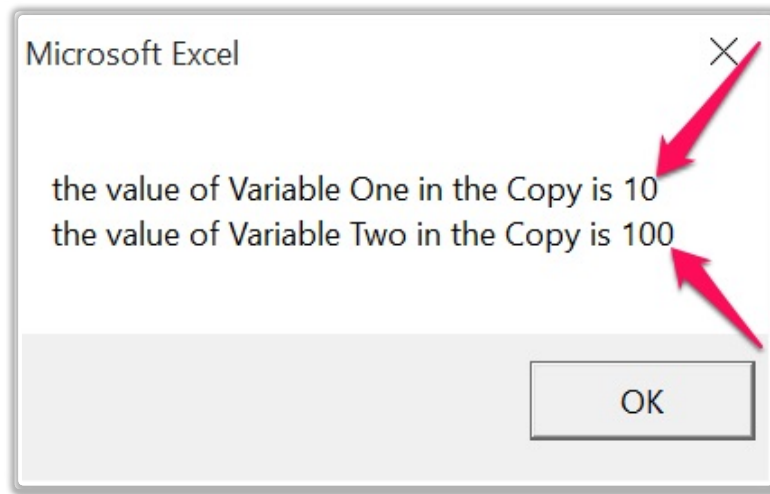
End Sub
```

The following message box is displayed by Excel when the Variable\_Test macro is executed:



And the following is the message box displayed when the Variable\_Test\_Copy macro is executed. Notice how both variables (Variable\_One and Variable\_Two):

- have retained the values that were assigned by the variable\_test macro for use in the Variable\_Test\_Copy macro.



Now, let's take a look at the third type of variable scope:

## Variable Scope #3: Public VBA Variables

The availability of public VBA variables is even broader than that of module-level variables:

- At the basic level, **public variables they are available for any procedure within any VBA module stored in the relevant Excel workbook.**
- At a more complex level, **you can make public variables available to procedures and modules stored in different workbooks.**

As explained by [Microsoft](#), variables that are declared with the Public statement are visible to all procedures in all modules "unless the Option Private Module is in effect". If the Option Private Module is enabled, variables are only available within the project in which they are declared.

When deciding to work with public VBA variables, remember the general rule I describe above: **make your VBA variables as local as possible.** [This website](#) (suggested in *Mastering VBA for Microsoft Office 2013*) has some interesting points regarding the potential problems of using unnecessary public variables, and why their convenience sometimes outweighs those potential problems.

Let's start by taking a look at the basic case:

## How To Declare Public VBA Variables: The Basics

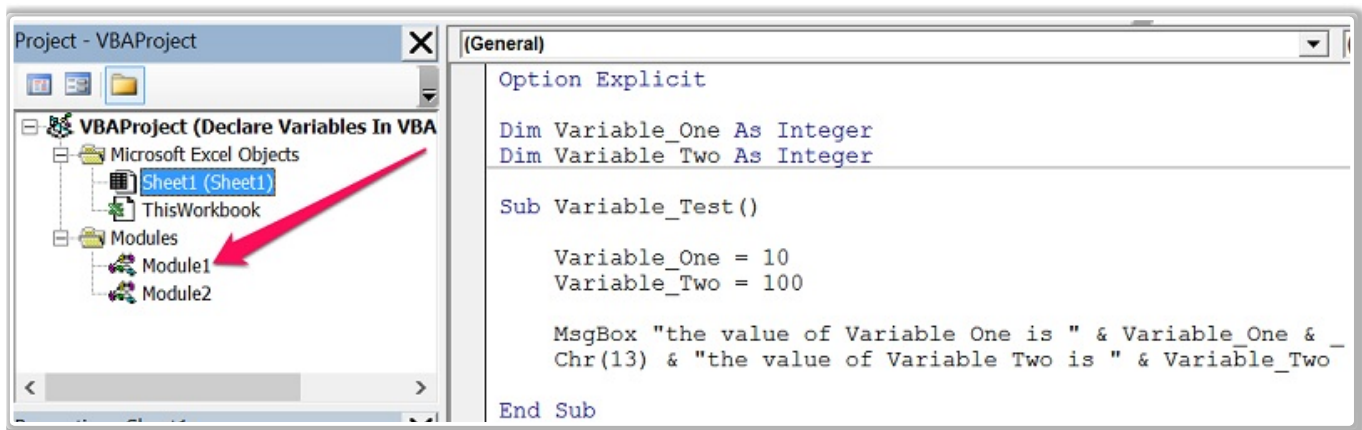
In order to **declare a public VBA variable**, the declaration needs to meet the following conditions:

- Use the Public statement in place of the Dim statement.
- The declaration must be made in a standard VBA module.

VBA variables whose declarations meet these conditions **are available to any procedure (even those in different modules) within the same Excel workbook.**

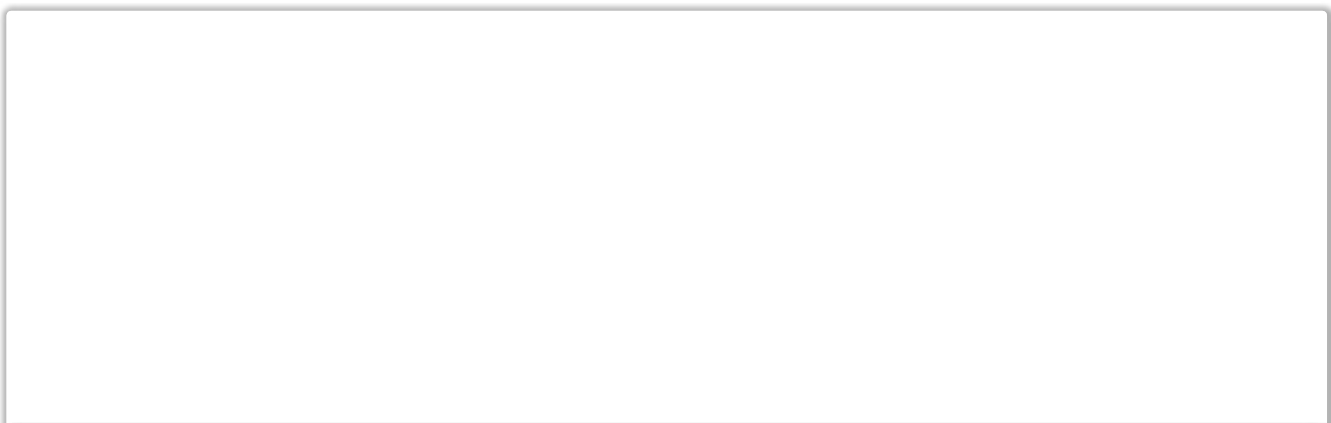
To see how this works, let's go back to the macros above: the Variable\_Test and Variable\_Test\_Copy macros and see how the variables (Variable\_One and Variable\_Two) must be declared in order to be public.

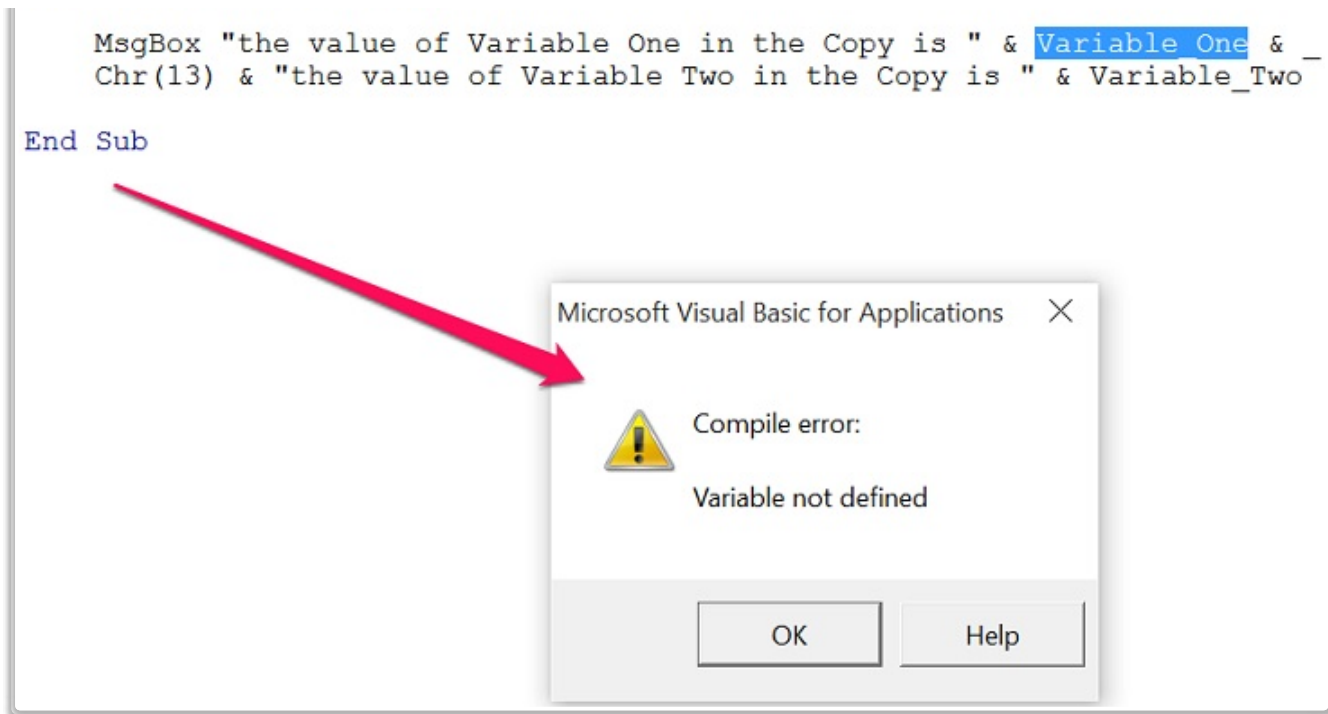
For this example, I first store both macros in different modules. Notice how, in the screenshot below, there are 2 modules shown in the Project Explorer.



Module1 stores the Variable\_Test macro while Module2 stores the Variable\_Test\_Copy macro. The variables are declared in Module1. You can **get immediate access (for free) to the Excel workbook that contains these modules by [clicking here](#).**

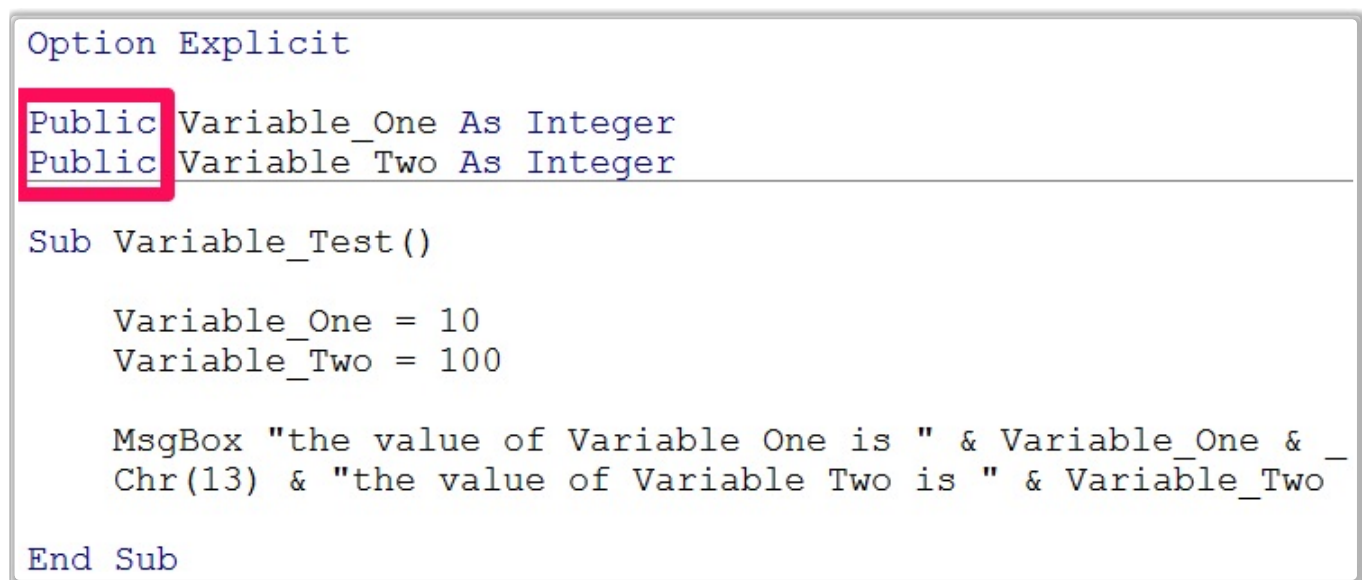
Notice how, in the screenshot above, both variables are declared using the Dim statement. Therefore, as it stands, they're module-level variables. If I try to execute the Variable\_Test\_Copy macro (which now has no variable declaration nor assignment) which is stored in Module2, Excel returns the following error message:



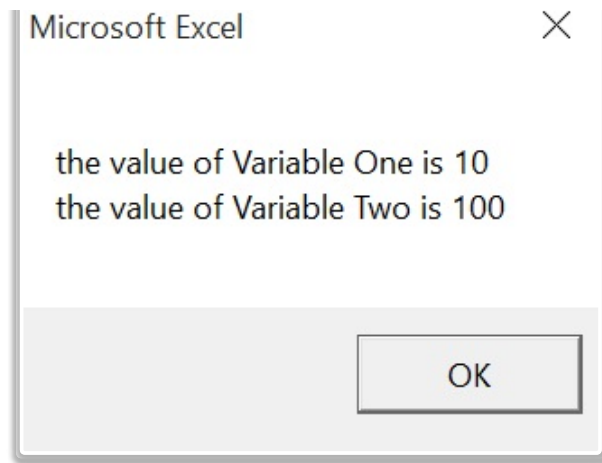


This is exactly the same error message displayed in the example above, when examining the Option Explicit statement and the reasons why you should always declare variables.

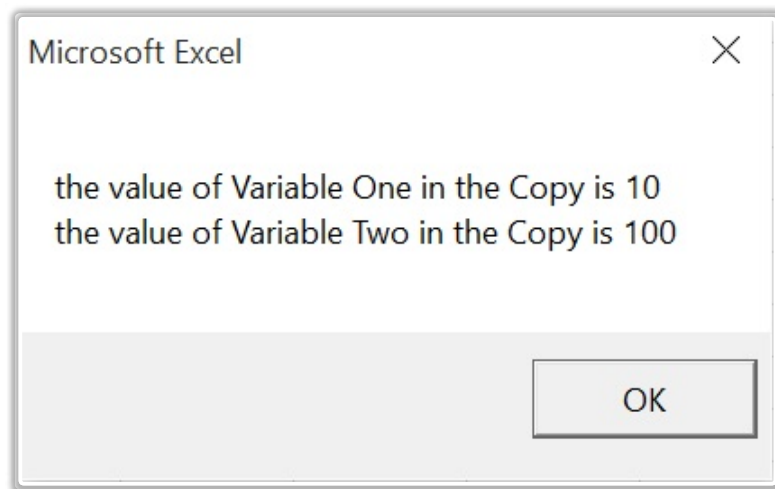
Let's go back to Module1, where both Variable\_One and Variable\_Two are declared, and change the keyword used to declare them. Instead of using Dim, we use Public. The rest of the macro remains unchanged.



If I run again both macros (Variable\_Test and Variable\_Test\_Copy) in the same order I did above, the message boxes displayed by Excel now show the same values. More precisely:



- And, since the variables are now declared using the Public statement, Excel displays the following upon the Variable\_Test\_Copy macro (which doesn't declare the variables nor assigns values to them) being called.



As shown by the screenshots above:

- Both variables are now available to all the procedures within the relevant workbook, even though they are stored in separate modules.
- The variables retain the values assigned by the first macro (Variable\_Test) for use by the second macro (Variable\_Test\_Copy).

This basic method of declaring public VBA variables is, in practice, the only one that you really need to know. However, in case you're curious, let's take a look at...

## How To Declare Public VBA Variables Available Across Different Workbooks



## SECTION II.

However, it is also possible to make public VBA variables available to procedures within modules stored in different Excel workbooks. This is rare, very rare...

In fact, John Walkenbach (one of the most experienced Excel authorities out there) states the following in *Excel VBA Programming for Dummies*:

“ In practice, sharing a variable across workbooks is hardly every done. In fact, I’ve never done it once in my entire VBA programming career. But I guess it’s nice to know that it can be done, in case it ever comes up as a *Jeopardy!* question.

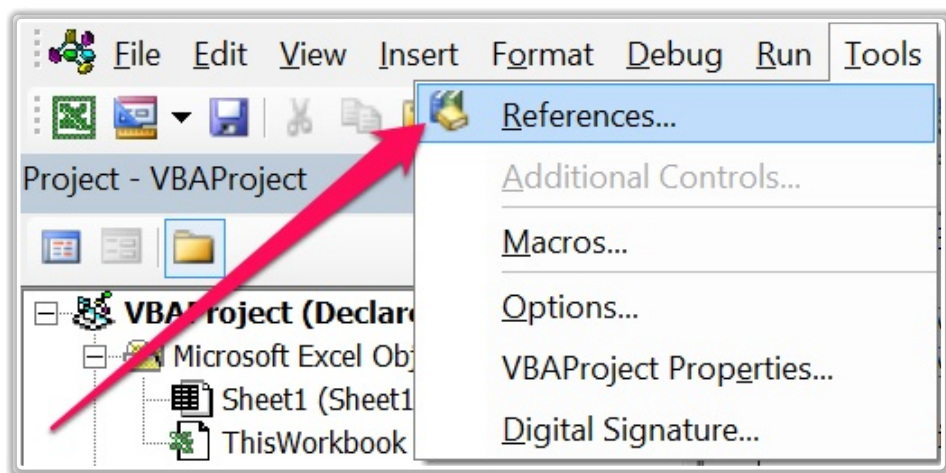
*Still interested in knowing how to declare a public VBA variable that is available across different Excel workbooks?*

OK, let’s practice for Jeopardy! then:

You **make a variable that is available to modules stored in Excel workbooks different from that in which the variable is actually declared** by following these 2 simple steps:

- **Step #1:** Declare the variable as Public, following the indications in the previous section.
- **Step #2:** Create a reference to the Excel workbook where the relevant variable is declared.

You can set up a reference by using the References command in the Visual Basic Editor. This command appears in the Tools menu of the VBE:



As a general rule, the scope of a variable determines its lifetime. Therefore, let’s take a look at...



The term **“life”** refers to how long a variable is stored in the computer’s memory.

The relationship between the scope and life of a VBA variable is explained in *Excel VBA Programming by Dummies* as follows:

“ The scope of a variable not only determines where that variable may be used, but also affects the circumstances under which the variable is removed from memory.

In other words, **the life of a VBA variable depends, usually, on its scope.**

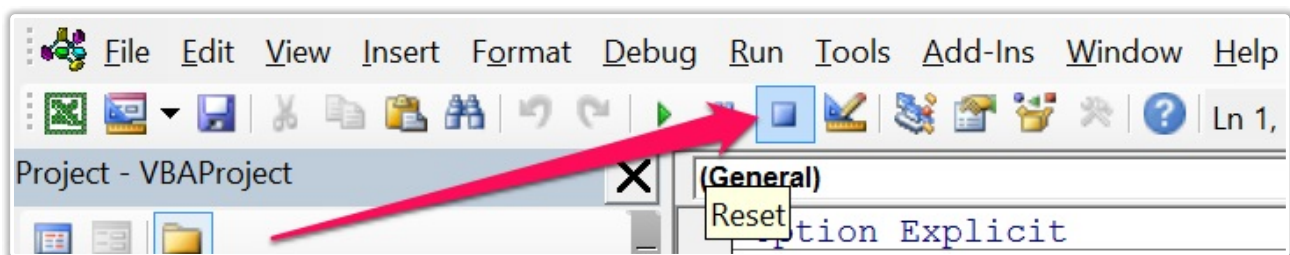
Fortunately, the topic of variable life is simpler than that of scope. More precisely, there are **2 basic rules**:

- **Rule #1:** Procedure-level VBA variables declared with the Dim statement are removed from the memory when the relevant procedure finishes.
- **Rule #2:** Procedure-level Static variables, module-level variables and public variables retain their values between procedure calls.

In addition to the above, John Walkenbach explains how **you can purge absolutely all the variables that are currently stored in the memory using any of the following 3 methods**:

- **Method #1:** Click the Reset button.

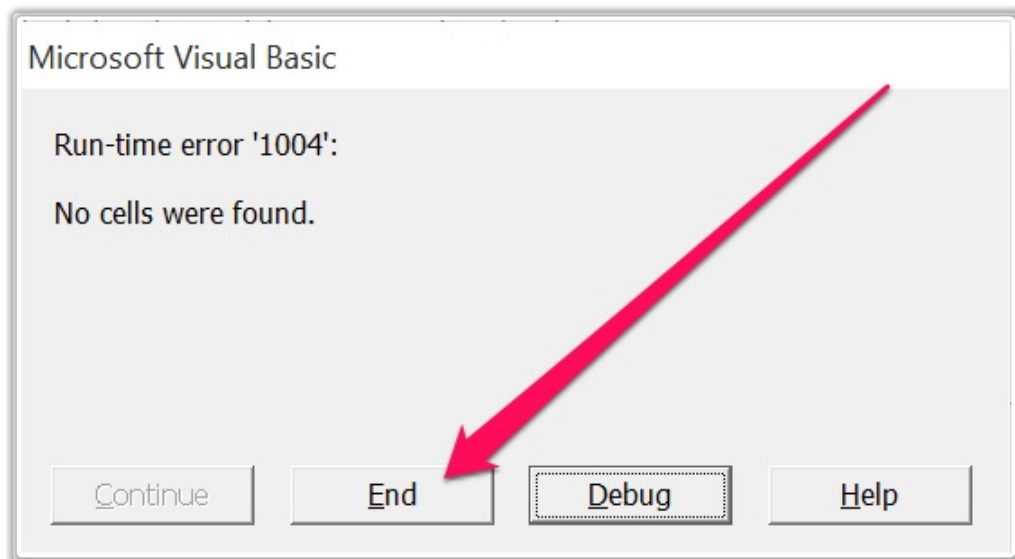
You can find the Reset button on the Standard toolbar within the Visual Basic Editor or within the Run menu, as shown by the images below.





- **Method #2:** Click “End” when a Run-time error dialog is displayed.

The following image shows a Run-time error caused by trying to execute a macro to delete rows if there are blank cells in a specified range without an On Error Resume Next error handler which is needed in that case. The End button is located at the bottom of the dialog.



- **Method #3:** Include an **End statement** within your VBA code.

From a broad point of view, the End statement has different syntactical forms. However, the one I'm referring to here is simply “End”. You can put this End statement anywhere in your code.

The End statement can have several consequences, such as terminating code execution and closing files that were opened with **the Open statement**. For purposes of this Excel tutorial, the most important consequence of using the End statement is that it clears variables from the memory.

Having a good understanding of these 3 variable-purging methods is important, even if you don't plan to use them explicitly. The reason for this is that **a variable purge may cause your module-level or public variables to lose their content (even without you noticing)**.

Now that we've fully covered the topic of how to determine which application or part of an application can use a variable (by looking at the topics of VBA variable scope and life), let's understand...

## How To Name VBA Variables

Visual Basic for Applications provides a **lot of flexibility in connection with how you can name variables**. As a general matter, the rules that apply to variable naming are substantially the same rules that apply to most other elements within Visual Basic for Applications.

The **main rules that apply to VBA variable naming** (as listed in *Excel VBA Programming for Dummies*, *Mastering VBA for Microsoft Office 2013*, [ozgrid.com](http://ozgrid.com) and [FunctionX](#)) **can be summarized as follows**. I also cover the topic of naming (in the context of naming procedures) within Visual Basic for Applications [here](#) and [here](#), among other places.

- The first character of the variable name must be a letter.
- Characters (other than the first) can be letters, numbers, and some punctuation characters.

One of the most commonly used punctuation characters is the underscore (\_). Some VBA programmers, use it to improve the readability of VBA variable names. For illustration purposes, take a look at the macro names I use in [this blog post](#). According to Richard Mansfield, author of *Mastering VBA for Microsoft Office 2013*, is more common to omit the underscore and use mixed case variable names (as explained below).

- VBA variable names can't include spaces ( ), periods (.), mathematical operators (for example +, -, /, ^ or \*), comparison operators (such as =, < or >) or certain punctuation characters (such as the type-declaration characters @, #, \$, %, & and ! that I explain below).
- The names of VBA variables can't be the same as those of any of Excel's reserved keywords. In *Excel VBA Programming for Dummies*, John Walkenbach lists several examples of words that fall within these restrictions such as Sub, Dim, With, End, Next and For.

If you try to use any of this words, Excel returns a syntax error message. However, as mentioned in *Excel 2013 Power Programming with VBA*, these messages aren't very descriptive. Therefore, **if you get a strange compile error message, it may be a good idea to confirm that you're not using a reserved keyword to name any of your VBA variables**.

Walkenbach also explains that, from a theoretical standpoint, you could use names that match names

In a similar fashion, Richard Mansfield (in *Mastering VBA for Microsoft Office 2013*) suggests **avoiding variable names that match any names used by Visual Basic for Applications, or that match the name of built-in functions, statements or object members**. He explains how, even if it doesn't cause direct problems, it may prevent you from using the relevant function, statement or object member "without specifically identifying it to VBA by prefacing its name with VBA".

In summary: it's best to **create your own variable names, and make them unique and unambiguous**.

- Visual Basic for Applications doesn't distinguish between upper and lowercase letters. For example, "A" is the same as "a".

Regardless of this rule, some VBA programmers use mixed case variable names for readability purposes. For an example, take a look at the variable names used in [macro example #5](#), such as aRow and BlankRows.

Additionally, at [GlobaliConnect](#), Excel expert Amit Tandon suggests that it may be a good idea to start variable names with lowercase letters. This reduces the risk of confusing those names with built-in VBA keywords.

- The maximum number of characters a VBA variable name can have is 255. However, as mentioned in *Excel 2013 Power Programming with VBA*, very long variable names aren't recommended.
- The name of each VBA variable must be unique within its relevant scope. This means that the name of a procedure-level variable must be unique within the relevant procedure, the name of a module-level variable must be unique within its module, and so on.

This requirement makes sense if you consider how important it is to ensure that Visual Basic for Applications uses the correct variable (instead of confusing it with another one).

At a more general level, it makes sense to **have (almost) always unique names**. As I explain above, re-using variable names can be confusing when debugging. Some Excel authorities, such as [Dick Kusleika](#), consider that there are some reasonable exceptions to this rule, such as certain control variables.

In addition to these rules, and suggestions related to those rules, you can find a few additional practices or suggestions made by Excel experts below.

“ You make your life easier if you get into the habit of making your variable names as descriptive as possible.

In some case, you can use comments to describe variables that don't have descriptive names. I suggest, however, that you try to use descriptive variable names. They are, in my opinion, more helpful.

Finally, some Excel authorities (including Richard Mansfield) **suggest that you add a prefix to the names of VBA variables for purposes of identifying their data type**. Mansfield argues that adding these prefixes makes the code more readable and easier to modify since “each variable, everywhere in the code, is identified as a particular type”.

The following table shows **the main tags that apply to VBA variables**:

DATA TYPE	PREFIX
Boolean	bln
Currency	cur
Double	dbl
Integer	int
Long	lng
Single	sng
String	str
Type (User-Defined)	typ
Variant	var

In *Excel 2013 Power Programming with VBA*, John Walkenbach introduces a similar naming convention. In that case, the convention calls for using a lowercase prefix indicating the data type. The following table shows these prefixes (some are the same as those in the table above):

Boolean	b
Integer	i
Long	l
Single	s
Double	d
Currency	c
Date/Time	dt
String	str
Object	obj
Variant	v
User-Defined	u

Walkenbach doesn't use this convention because he thinks it makes the code less readable. However, other Excel authorities such as [Daniel Ferry](#), do include a couple of letters at the beginning of the variable name to represent the data type.

In the end, you'll develop your own variable-naming method that works for you. **What's more important than following the ideas of other Excel users, as explained at [ozgrid.com](#), is to "formalize some sort of naming convention" and use it consistently.** Please feel free to use any of the ideas and suggestions that appear (or are linked to) above when creating your variable-naming system...

And make sure to let me know your experience and ideas by **leaving a comment below**.

The sections above cover all the main items within the declaration statement of a VBA variable. Let's take a quick look at what happens after you've declared the variable by understanding...

## How To Assign A Value Or Expression To A VBA Variable

- The default value for numeric data types (such as Byte, Integer, Long, Currency, Single and Double) is 0.
- In the case of string type variables (such as String and Date), the default value is an empty string (""), or the **ASCII code** 0 (or Chr(0)). This depends on whether the length of the string is variable or fixed.
- For a Variant variable, the default value is Empty. How the Empty value is represented depends on the context:

In a numeric context, the Empty value is represented by 0.

In a string context, it's represented by the zero-length string ("").

As a general rule, however, **you should explicitly assign a value or expression to your VBA variables** instead of relying in the default values.

The statement that you use to assign a value to a VBA variable is called, fittingly, **an Assignment statement**. Assigning an initial value or expression to a VBA variable is also known as initializing the variable.

**Assignment statements take a value or expression (yielding a string, number or object) and assign it to a VBA variable or a constant.**

You've probably noticed that, as explained at **Home & Learn**, you can't declare a VBA variable and assign a value or expression to it in the same line. Therefore, in Visual Basic for Applications, assignment statements are separate from variable declaration statements.

This Excel tutorial refers only to variables (not constants) and expressions are only covered in an introductory manner. Expressions are more useful and advanced than values. Among other strengths, expressions can be as complex and sophisticated as your purposes may require.

Additionally, if you have a good understanding of expressions, you shouldn't have any problem with values. Therefore, let's take a closer look at what an expression is.

Within Visual Basic for Applications, **expressions can be seen as having 3 main characteristics:**

- They are composed of a combination of one or more of the following 4 elements:

**Element #2:** Operators.

**Element #3:** Variables.

**Element #4:** Constants.

- Their result is one of the following 3 items:

**Item #1:** A string.

**Item #2:** A number.

**Item #3:** An object.

- They can be used for any of the following 3 purposes:

**Purpose #1:** Perform calculations.

**Purpose #2:** Manipulate characters.

**Purpose #3:** Test data.

Expressions aren't particularly complicated. As explained by John Walkenbach in *Excel 2013 Power Programming with VBA*:

“ If you know how to create formulas in Excel, you'll have no trouble creating expressions in VBA.

The **main difference between formulas and expressions** is what you do with them:

- When using **worksheet formulas**, Excel displays the result they return in a cell.
- When using expressions, you can use Visual Basic for Applications to assign the result to a VBA variable.

**As a general rule, you use the equal sign (=) to assign a value or expression to a VBA variable.** In other words, the equal sign (=) is the assignment operator.



The **basic syntax of an assignment statement using the equal sign** is as follows:

- On the right hand side of the equal sign: The value or expression you're storing.
- On the left hand side of the equal sign: The VBA variable where you're storing the data.

As mentioned by Richard Mansfield in *Mastering VBA for Microsoft Office 2013*, [Microsoft](#) and [ExcelFunctions.net](#), **you can change the value or expression assigned to a variable during the execution of an application.**

To finish this section about Assignment statements, let's take a quick look at the assignments made in the sample macros Variable\_Test and Variable\_Test\_Copy. The following image shows how this (quite basic and simple) assignments were made.

```
Sub Variable_Test()  
  
    Dim Variable_One As Integer  
    Dim Variable_Two As Integer  
  
    Variable_One = 10  
    Variable_Two = 100  
  
    MsgBox "the value of Variable One is " & Variable_One & _  
        Chr(13) & "the value of Variable Two is " & Variable_Two  
  
End Sub  
  
Sub Variable_Test_Copy()  
  
    Dim Variable_One As Integer  
    Dim Variable_Two As Integer  
  
    Variable_One = 20  
    Variable_Two = 200  
  
    MsgBox "the value of Variable One in the Copy is " & Variable_One & _  
        Chr(13) & "the value of Variable Two in the Copy is " & Variable_Two  
  
End Sub
```

Notice the use of the equal sign (=) in both cases. In both the Variable\_Test and the Variable\_Test\_Copy macros, the value that appears on the right side of the equal sign is assigned to the variable that appears on the left side of the sign.

By now we've covered pretty much all the basics in connection with declaring variables in Visual Basic for Applications. You may, however, be wondering...

The following section explains how you can do this...

## How To Declare VBA Variables Implicitly (a.k.a. How To Create Variables Without Declaring Them)

The first few sections of this Excel tutorial explain why declaring variables explicitly is a good idea.

However, realistically speaking, this may not be always possible or convenient. In *Excel 2013 VBA and Macros*, Bill Jelen and Tracy Syrstad explain that usually developers don't really take the time to declare variables. In fact, they explain that:

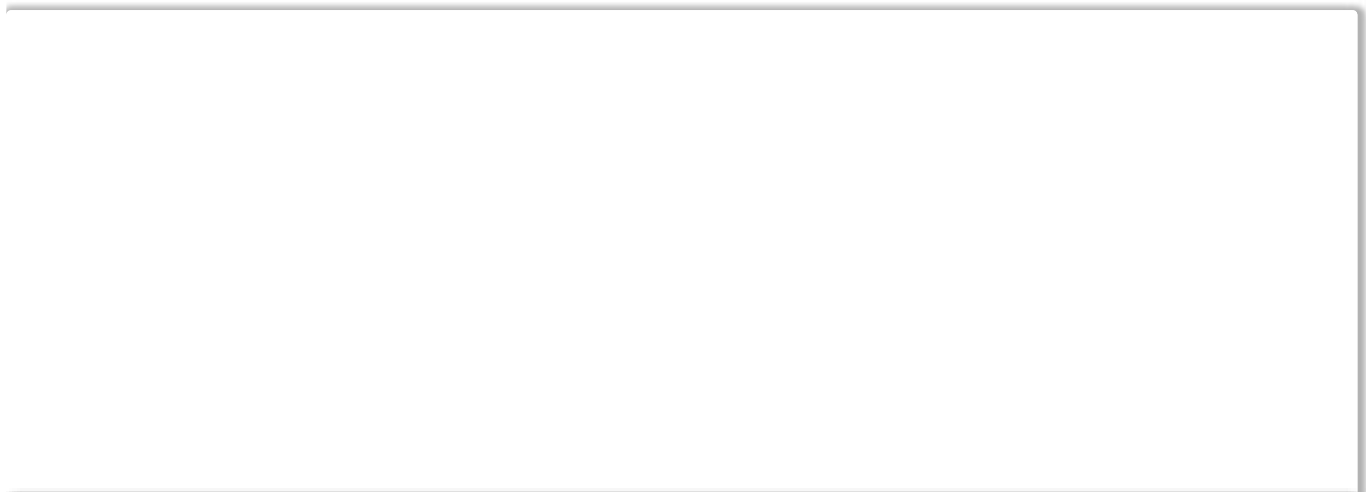
“ (...) developers then to whip up a new variable on the fly as the need arises.

Doing this is relatively simple and is known as declaring a variable implicitly. In practical terms, **this means using a VBA variable in an assignment statement without declaring explicitly** (as explained throughout most of this blog post) first.

The reason why this works, as explained by [Chip Pearson](#), is the way Visual Basic for Applications proceeds when it encounters a name that it doesn't recognize (for example as a variable, reserved word, **property** or **method**): it creates a new VBA variable using that name.

Let's take a look at a practical example by going back to the Variable\_Test and Variable\_Test\_Copy macros...

The image below shows the VBA code of both the Variable\_Test and Variable\_Test\_Copy macros without Variable\_One and Variable\_Two being ever declared explicitly. Notice how there is no Dim statement, as it is the case in the image above. Both VBA Sub procedures begin by simply assigning a value to the relevant variables.



```

    Variable_Two = 100

    MsgBox "the value of Variable One is " & Variable_One & _
    Chr(13) & "the value of Variable Two is " & Variable_Two

End Sub

Sub Variable_Test_Copy()

    Variable_One = 20
    Variable_Two = 200

    MsgBox "the value of Variable One in the Copy is " & Variable_One & _
    Chr(13) & "the value of Variable Two in the Copy is " & Variable_Two

End Sub

```

When this happens, **Visual Basic for Applications proceeds as follows:**

- **Step #1:** It carries out a check to confirm that there isn't an existing variable with the same name.
- **Step #2:** If there are no other variables with the same name, it creates the variable.

Since the data type of the variable isn't declared, VBA uses the default Variant type and assigns one of its sub-types.

**You can, actually, set the data type of an implicitly declared variable by using type-declaration (also known as type-definition) characters.** These are characters that you can add at the end of the variable name to assign the data type you want.

As explained by John Walkenbach in *Excel 2013 Power Programming with VBA*, type-declaration characters are a "holdover" from BASIC. He agrees that **it's generally better to declare variables using the methods I describe above.**

The following table displays the main type-declaration characters you can use:

CHARACTER	DATA TYPE
%	Integer
&	Long
@	Currency

!	Single
#	Double
\$	String (variable length)

For example, in the case of the Variable\_Test and Variable\_Test\_Copy macros that appear above, you can assign the Integer data type to the variables Variable\_One and Variable\_Two by adding the % type-declaration character at the end of their respective names. The VBA code looks, then, as follows:

```
Sub Variable_Test()  
    Variable_One% = 10  
    Variable_Two% = 200  
    MsgBox "the value of Variable One is " & Variable_One & _  
        Chr(13) & "the value of Variable Two is " & Variable_Two  
End Sub  
  
Sub Variable_Test_Copy()  
    Variable_One% = 20  
    Variable_Two% = 200  
    MsgBox "the value of Variable One in the Copy is " & Variable_One & _  
        Chr(13) & "the value of Variable Two in the Copy is " & Variable_Two  
End Sub
```

As shown above, you **only need to include the type-declaration character once**. Afterwards, you can use the variable name as usual (without type-declaration character). The following image shows how this is the case in the Variable\_Test and Variable\_Test\_Copy macros:

```
variable_two% = 100

MsgBox "the value of Variable One is " & Variable_One & _
Chr(13) & "the value of Variable Two is " & Variable_Two

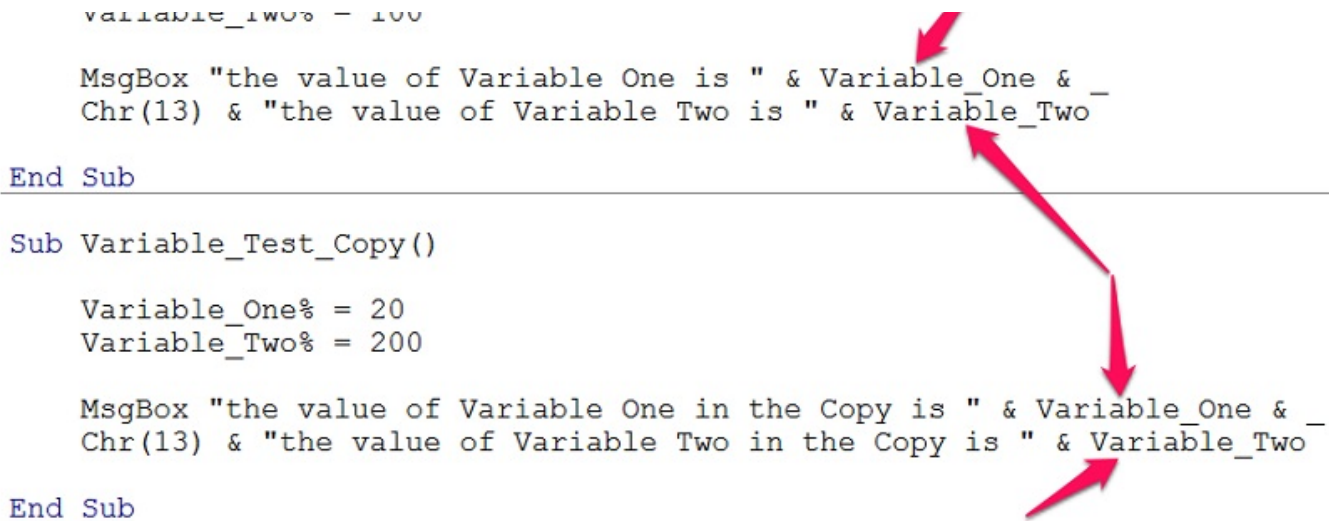
End Sub

Sub Variable_Test_Copy()

Variable_One% = 20
Variable_Two% = 200

MsgBox "the value of Variable One in the Copy is " & Variable_One & _
Chr(13) & "the value of Variable Two in the Copy is " & Variable_Two

End Sub
```

A diagram with three red arrows illustrating variable scope resolution. One arrow points from the 'Variable\_One' in the first MsgBox statement to the 'Variable\_One' in the 'Variable\_Test\_Copy' sub procedure. A second arrow points from the 'Variable\_Two' in the first MsgBox statement to the 'Variable\_Two' in the 'Variable\_Test\_Copy' sub procedure. A third arrow points from the 'Variable\_One' in the 'Variable\_Test\_Copy' sub procedure to the 'Variable\_One' in the second MsgBox statement.

According to *Mastering VBA for Microsoft Office 2013*, there are **2 main advantages to declaring variables implicitly as described above**:

- **Advantage #1:** More flexibility:

By not declaring variables you are more flexible since, whenever you need a variable, you simply use it in a statement to declare it implicitly.

- **Advantage #2:** You write less VBA code.

However, before you decide to start declaring your variables implicitly as explained in this section, remember the advantages of always declaring your variables and using the Option Explicit statement, and the potential problems whose origin is the lack of variable declaration.

## Conclusion

Variables are a very flexible element that you're likely to encounter constantly while working with Visual Basic for Applications. As a consequence of that flexibility and ubiquity, **is important to have a good understanding of how to work with them**.

Otherwise, you may eventually run into a lot of problems while debugging your VBA code. Or even worse: your macros may simply not work as intended (without you noticing) and return wrong results.

**This Excel tutorial has covered the most important topics related to VBA variable declaration.**

Additionally, we've seen several different suggestions and best practices regarding work with VBA variables.

**Some of the most important topics covered by this VBA tutorial are** the following:

- Why is it convenient to declare VBA variables explicitly.
- How can you remember to declare variables explicitly.
- How to declare VBA variables taking into consideration, among others, the desired scope and life of that variable.
- How to name variables.
- How to assign values or expressions to VBA variables.

Finally, in case I didn't convince you to declare your VBA variables explicitly, the last section of this Excel tutorial covered the topic of implicit variable declaration. This section provides some guidance of how to work with variables without declaring them first.

As you may have noticed, there are a lot of different opinions on the topic of best practices regarding VBA variables. It's not surprise that Excel MVP Daniel Ferry stated that:

“ Everyone and their brother seems to have a different method for naming variables.

As explained above, **I encourage you to develop your own method for naming variables**. See what works for you, formalize it and use it consistently.

And in any case, **please make sure to share with the rest of us...**

## What are your best practices when naming VBA variables?

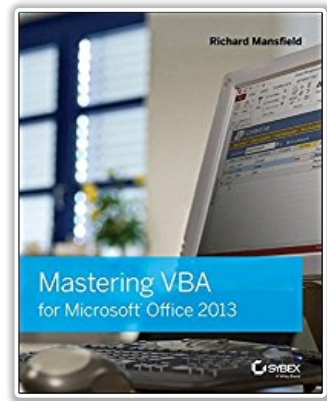
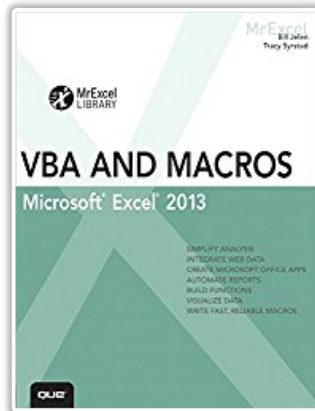
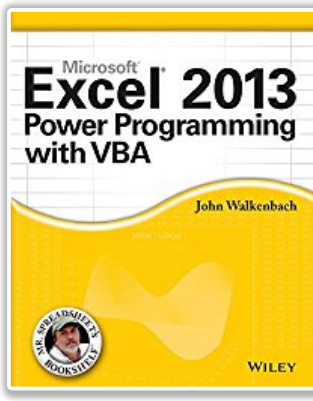
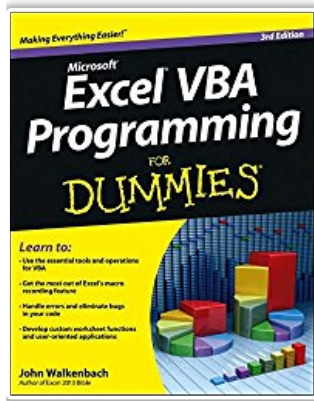
I'm quite interested in reading your ideas about the best practices regarding variable naming. Please make sure to share them by leaving a comment below.

Your comments can include answers to the following questions:

- *Do you agree with or use the ideas and suggestions described above in connection with variable naming?*
- *Do you name your variables using other strategies that I didn't mention above?*

# Books Referenced In This Excel Tutorial

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



**Clay Tillman** • 2 months ago

This is a great explanation, with so much useful information. However, why do you not have a category that talks about what each type of variable is? EX: A Boolean is a True False, etc. Is this discussed elsewhere?

^ | ▾ • Share ›

**Jorge A. Gomez** Mod → **Clay Tillman** • 2 months ago

Many thanks for your comments, Clay!

I cover the topic of data types in a separate post: [https://powerspreadsheets.c....](https://powerspreadsheets.c...) I think that post may discuss what you're looking for.

Please let me know if you have additional questions/comments.

^ | ▾ • Share ›

**Clay Tillman** → **Jorge A. Gomez** • 2 months ago

Thank you!

^ | ▾ • Share ›



## Join thousands of 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...



## POWER SPREADSHEETS IN SOCIAL MEDIA



[Excel Resources](#) | [Excel Shortcuts](#)



[Contact](#)

---

Copyright © 2015–2017 Jorge Alberto Gomez Soto. All rights reserved

[Privacy Policy](#) | [Terms and Conditions](#) | [Limit of Liability and Disclaimer of Warranty](#)

Excel ® is a registered trademark of the Microsoft Corporation. Power Spreadsheets is not affiliated with the Microsoft Corporation.

Some of the links in Power Spreadsheets are affiliate links. If you make a purchase through one of them, I may receive a commission. This commission comes at no additional cost to you,