### COURSE OUTLINES

**Module 0 –** [**Getting Started**](#Module00)

* [Introduction](#Introduction)
* [Getting to the Visual Basic Editor](#GettingToTheVisualBasicEditor)
* [Working with Projects](#WorkingWithProjects)
* [Working with Modules](#WorkingWithModules)
* [VBA Environment](#VBAEnvironment)
* [Writing VBA Code](#WritingVBACode)
* [Changing Things in VBA by Using Properties](#ChangingThingsInVBAbyUsingProperties)
* [Using the Immediate Window](#UsingTheImmediateWindow)

**Module 1 –** [**Building Blocks in VBA**](#Module01)

* [Modular and Event Driven Concepts](#ModularAndEventDrivenConcepts)
* [Modules](#Modules)
* [Class Model](#ClassModel)
* [User Form](#UserForm)
* [Object Model](#ObjectModel)

**Module 2 –** [**Parameter Passing Mechanisms**](#Module02)

* [Parameter Passing Mechanisms](#ParameterPassingMechanisms)
* [Optional Parameter](#OptionalParameter)
* [Named Parameter](#NamedParameter)
* [Variable Parameter with ParamArray](#VariableParameterWithParamArray)

**Module 3 –** [**Subroutines, Functions, Event Handlers and Macro**](#Module03)

* [When to use which](#WhentoUseWhich)?
* [Return value from Function](#ReturnValueFromFunction)
* [Termination](#Termination)
* [Macro: A special form of Sub](#Macro)
* [Event Handler: A special form of Sub](#EventHandler)

**Module 4 –** [**Advanced Array Handling**](#Module04)

* [LBound and UBound Functions](#LBoundAndUBoundFunctions)
* [Multi-Dimensional Array](#MultiDimensionalArray)
* [Redim](#Redim)
* [Redim Preserve](#RedimPreserve)

**Module 5 –** [**Advanced Types**](#Module05)

* [Variant](#Variant)
* [Using Type](#UsingType)
* [Object](#Object)
* [Predefined Constants](#PredefinedConstants)

**Module 6 –** [**Options**](#Module06)

* [Option Base](#OptionBase)
* [Option Explicit](#OptionExplicit)
* [Option Compare](#OptionCompare)
* [Option Private](#OptionPrivate)

**Module 7 –** [**Error Handling**](#Module07)

* [On Error](#OnError)
* [On Error Resume Next](#OnErrorResumeNext)
* [On Error Goto](#OnErrorGoto)

**Module 8 –** [**Special Constructs**](#Module08)

* [For Each](#ForEach)
* [Abnormal termination with Exit keyword](#AbnormalTerminationWithExitKeyword)

**Module 9 –** [**User Form**](#Module09)

* [The purpose](#ThePurpose)
* [How to Declare?](#HowToDeclare)
* [How to call it?](#HowToCallIt)

**Module 10 –** [**Class Module**](#Module10)

* [The Purpose](#ThePurpose2)
* [Properties Declaration](#PropertiesDeclaration)
* [How to use it?](#HowToUseIt)

**Module 11 –** [**Document Object Model**](#Module11)

* [Dealing with events](#DealingWithEvents)
* [Event Parameters](#EventParameters)

**Module 12 –** [**Advanced Topics**](#Module12)

* [Using Declare keyword to extend VBA ability using external DLL](#UsingDeclareKeywordToExtendVBAAbility)
* [Advanced Techniques](#AdvancedTechniques)
* [Security Concerns](#SecurityConcerns)
* [Common Pitfalls](#CommonPitfalls)
* [Good Practices](#GoodPractices)

Module 0 – Getting Started

Introduction

Microsoft® Visual Basic® for Applications, or **VBA** for short, is a powerful programming language that is embedded in every Microsoft Office product.  Whether you are using Word, Excel or PowerPoint, you are only ever a few mouse clicks away from starting to write your own programs.

Getting to the Visual Basic Editor

Whichever Microsoft Office application you happen to be using, you can be sure that embedded within it is another application called the **V**isual **B**asic **E**ditor, or **VBE**.  This is the application that you use to write your VBA code.  There are various menu or Ribbon options that will take you to the VB Editor, depending on which application and which version of Office you are using, but you can always get to the VB Editor with a keyboard shortcut.  To do this, hold down the ALT key on the keyboard, and then press F11.

|  |  |
| --- | --- |
| The Visual Basic Editor | This is how the VB Editor should look the very first time you open it. The two windows that we've highlighted are:   1. The **Project Explorer** 2. The **Properties Window** |

The VB Editor (VBE) application is the same regardless of which Office application you are in when you open it.  You will see slightly different things in the **Project Explorer** and **Properties** window depending on which application you are using - the diagram above is using Excel 2007.

**Note:** You can use the shortcut ALT + F11 at any time to toggle between the VB Editor and the Office application you are working in.  Much quicker than clicking with the mouse!

If you can't see the two windows that we've highlighted in the image above, you can go to the **View** menu at the top of the VB Editor to display them.

|  |  |
| --- | --- |
| Displaying the Project Explorer and Properties window | You can click the options shown here to show the two relevant windows, or use the keyboard shortcuts that are listed next to the options in the menu. |

Working with Projects

A **Project** is the name for the collection of VBA objects that are part of the file you are working on.  It is created automatically when new Office file (such as an Excel workbook, Word document, or PowerPoint presentation) is opened, and only one project associated with one file.

The only useful thing you can really do with a project at this point is to rename it.  To do this:

|  |  |
| --- | --- |
| Renaming a project | 1. *Click on the project in the* ***Project Explorer****.* 2. *Type in a new name for the project in the* ***Properties*** *window.*   A project name can't contain spaces and various other punctuation characters - it's best to stick to text and numbers. |

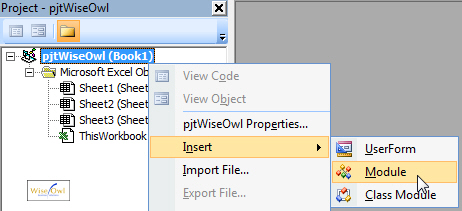
Although you can only have one VBA project for each Office file, you might also see other projects listed when you go into the VB Editor.  These extra projects include things like the Excel **Personal Macro Workbook**, the Word **Normal template**, Excel **Add-Ins**, and the Project **Global template**.

|  |  |
| --- | --- |
| *The Project Explorer in Microsoft Projectg* | *The Project Explorer in Microsoft Word* |
| The VBE in Microsoft Project always contains the **ProjectGlobal** VBA project.  This project is attached to the **Global.MPT** template file. | In Microsoft Word you will always see a VBA project associated with the template your document is based on - usually this will be the **Normal** template. |

Working with Modules

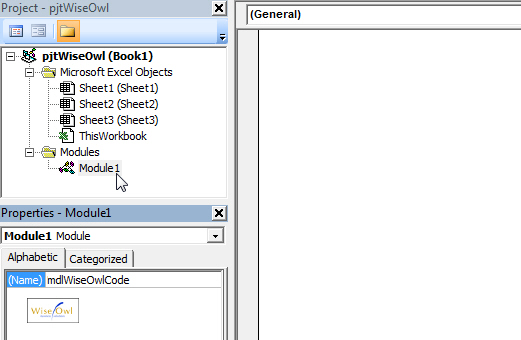
Before you can start writing any code, you need to insert a **Module**.  A module is the electronic equivalent of a blank piece of paper.  Each project can contain multiple modules - the exact number you have will depend on the size and complexity of the system you are developing, as well as your personal preference on how to organise your code.

To insert a new module, right-click on the VBA project you want to insert the module into, and from the menu choose: **Insert -> Module**



*You can actually right-click anywhere in the project you are working on to do this, but it's easier to spot the bold text of the project name when you have several files open.*

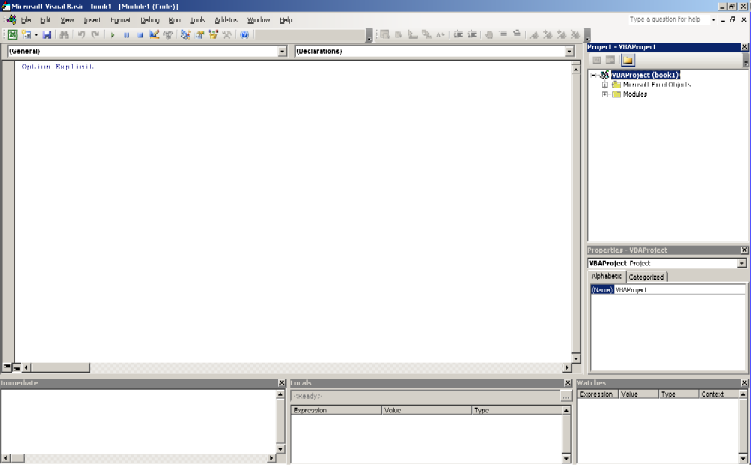
When you've done this, you'll see a new item in the **Project Explorer** window, and the great white expanse of the module you've just created taking up most of the right hand side of your screen.



**Note:** *You can rename a module in the same way you renamed a project earlier: select the module in the* ***Project Explorer****, and type in a new name in the* ***Properties*** *window.  Again, you can't use spaces or most of the punctuation characters - stick to text and numbers to be safe!*

Now that you've inserted and renamed a module, you can click into it with the mouse and start typing your VBA code.  Read the next part of this series to find out exactly what you need to type!

### VBA Integrated Development Environment

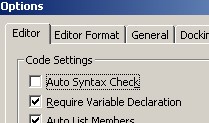


You can set up the **I**ntegrated **D**evelopment **E**nvironment (**IDE**) to support the way you work. Some people like to have all the main tool windows open all the time. Others like to just have one or two open. Note though once you have them docked like above you quickly learn not to change it as it is so hard to re-dock them the same way.

VBA Environment

Unfortunately the default environment settings are not geared towards high quality development, they are set to make it easy for beginners to get productive quickly.

These are our preferred settings for professional quality development. Under the VBE, select **Tools**>>**Options**



The default is to have those top 2 reversed.

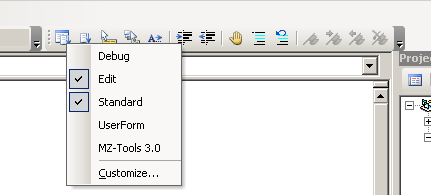
You always get Auto Syntax Check (line goes red), removing the tick stops the disruptive modal error message boxes from constantly popping up every time you move off a line to copy something:



If the advice were a bit more helpful it might be worthwhile, but even beginners would struggle to get any value out of the above example. And it isn't going to get better as VBA is the end of lined. Advice: Turn off Auto Syntax Check.

Require Variable Declaration puts an 'Option Explicit' at the top of each new code resource you open (module, class, form etc) (note it is not retrospective, hence the need to set it ASAP). Not using option explicit is just sloppy and is sure to lead to hard to spot errors in any significant coding. If you want to use a variable called x (you may be able to think of a more meaningful name) Option Explicit forces you to 'Dim' it first, and optionally to decide what data type it should be. If Option Explicit is missing VBA will implicitly Dim the variable as a variant the first time you use it, if you later mistype the variable name VB will create another new variable, rather than warn you 'Variable not defined' which would be more useful.

Another bizarre default setting is the way the IDE does not show one of the most important toolbars as standard. Luckily you can right click in the menu area and show the Edit toolbar.



The 2 very useful commands here are the ones with blue lines; the first one comments out a line, the one with the blue arrow un-comments it.

Also of note is the option to show the MZ-Tools toolbar. This is an excellent free add-in that is vital to serious Excel VBA development.

**Note**: These settings will be reflected in other Office applications.

Writing VBA Code

You are now ready to write some real code.  This session will explain some of the basic rules of writing VBA and show you a couple of tricks to get you creating code as quickly as possible.

The Basic Grammar of VBA

VBA is a language, and like any language it has grammatical rules that you need to follow if you want to make sense when you're "speaking" it.  Generally when you're writing VBA programs, most of the times you'll be attempting to perform some kind of action on some sort of object.  The structure of a line of code that performs an action on an object is very consistent in VBA - you always start the sentence by referring to the thing you want to do something to, followed by the action you want to perform, and you use a full stop to separate the two.  So, very generally speaking, a basic sentence in VBA would look like this:

****Thing.Action****

In VBA terms, the **Thing** part of the above sentence would technically be called either a **collection** or an **object**.  The **Action** part of the sentence would be referred to as a method.  So, the technical way of representing the above sentence would look like this:

Object.Method  
 or  
Collection.Method

Bearing this in mind, we're going to write a line of code that will apply the **Add** method to the **Workbooks** collection.

Writing Your First Line of Code

In our first line of code, the collection part of the sentence is the word **Workbooks**.  “**Workbooks”** is the name for the collection of all of the currently open Excel files. Type it into your code and follow it immediately with a full stop.

|  |  |
| --- | --- |
| The Intellisense list | Typing in a full stop after a word that VBA recognises presents you with a list of other words you can use to finish the sentence.  You can identify the methods in the list by their "green flying brick" symbol. |

After typing in the full stop you should see a list of keywords appear automatically.  This feature is referred to as **Intellisense** - horrible name, useful feature!  The next section describes several ways to use **Intellisense** to save you as much typing as possible.

Using Intellisense to Complete a Sentence

After the **Intellisense** list appears we can complete our sentence in a number of ways.  The method part of our line of code is the word **Add** - to get this word into your code you could do any of the following:

|  |  |
| --- | --- |
| **Option** | **Effect** |
| Type the word manually. | The word is typed in and the cursor stays at the end of the line. |
| Use the mouse to scroll to the word you want and double-click on it. | The word is inserted automatically and the cursor appears immediately after the word. |
| Use the arrow keys or start typing the first letters of the word to highlight it in the list, then press Tab on the keyboard. | The word is inserted automatically and the cursor appears immediately after the word. |
| Use the arrow keys or start typing the first letters of the word to highlight it in the list, then press Enter on the keyboard. | The word is inserted automatically and the cursor appears on a new line below the previous one. |

Probably the quickest technique to use in this example is to type in the letter A which will automatically select the word **Add** in the list, and then press Enter.

|  |  |
| --- | --- |
|  | Congratulations, you've finally written your first line of code!  When we get around to running our subroutine, this line will create a new workbook |

Changing Things in VBA by Using Properties

So far we've seen how to create a new workbook in Excel VBA by applying the **Add** method to the **Workbooks** collection.  Now that we've done this we need to add some text to some of the cells in the file that we've just created.  We're going to do this by modifying a property of an object.  Properties are like methods in that they always appear after the object or collection in a VBA sentence.  Unlike methods, you can assign a value to a property to change it in some way.  The basic grammar of a line of code to do this would look like this:

Object.Property = SomeValueor  
Collection.Property = SomeValue

The object we are interested in is a cell, which is referred to as a **Range** in Excel VBA, and the property we want to change is called **Value**.  Type the following into your code:

|  |  |
| --- | --- |
|  | You can identify the properties in the list by their "finger pointing at a piece of paper" symbol. |

Referring to a **Range** object is slightly more complicated than referring to the **Workbooks** collection because we have to say which cell we are interested in.  We do this by typing a cell reference (here it is **A1**) inside a set of round brackets and double-quotes.  Next, we can type in a full stop to display the list of properties and methods.

The quickest way to select the **Value** property from the list is to do the following:

1. Type in the letter V to jump to the word **Validation**.
2. Press the down arrow key on the keyboard to select **Value**.
3. Press Tab.

This should leave you with a subroutine looking like this:

|  |  |
| --- | --- |
|  | The only remaining thing is to say what we want to change the value of the cell to. |

We can now say what text we want to appear in the cell.  To do this we need to type in an equals sign, = followed by the text.  All literal text in VBA must be enclosed in a set of double-quotes.  Type in the following and press Enter at the end of the line.

|  |  |
| --- | --- |
|  | When you press Enter at the end of the line you should see a space appear on either side of the equals sign. |

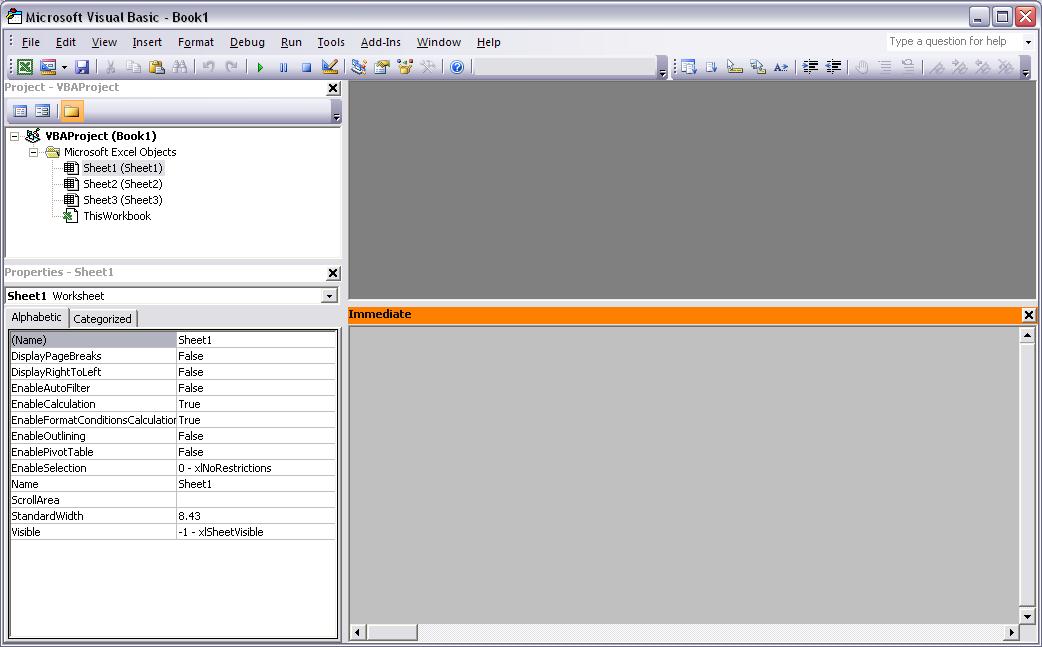
As a final flourish in our very basic program, we're going to write a line that will put today's date into another cell on the spreadsheet.  This line will look very similar to the one we've just created, so type in the following code.  Try to remember the quickest way to use the **Intellisense** list - hint: it doesn't involve the mouse!

|  |  |
| --- | --- |
|  | Press Enter at the end of the line and you should see the word **date** becomes capitalised. |

Rather than putting in the date as a string of literal text, we've used a built-in VBA function called **Date**.  This function calculates what today's date is each time the code is run (based on your computer's clock) and puts the result of the calculation into the cell.

Using the Immediate Window

Figure below shows the VBE environment. If the **Immediate Window** isn’t visible in your VBE, you can display it by pressing Ctrl + G or selecting **View**/**Immediate Window** from the main menu.



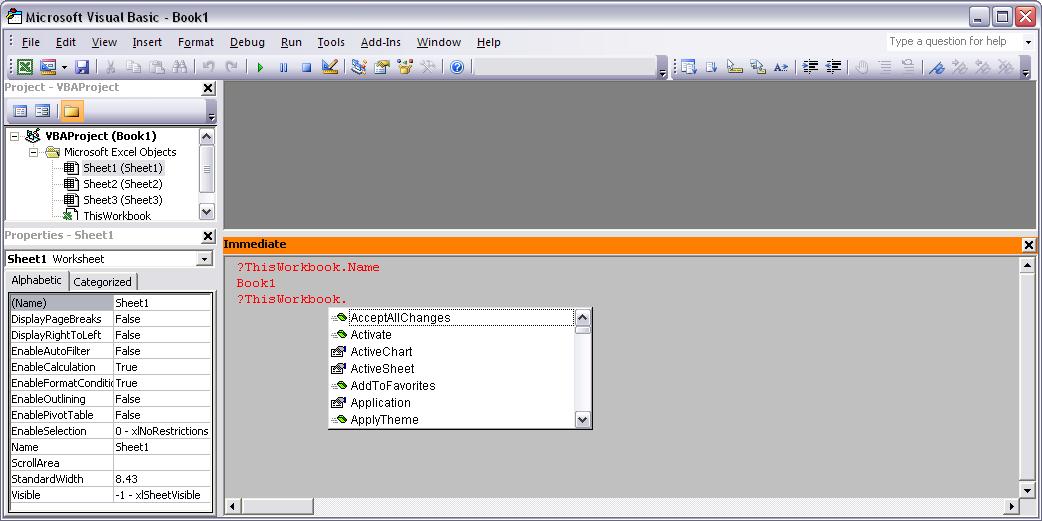
One use of the Immediate window is to evaluate individual VBA statements. In this scenario, you precede the VBA statement you want evaluated with a question mark. You can evaluate the values of object properties, functions, and mathematical statements, among other things. For example, Enter

?ThisWorkbook.Name

In the Immediate window and press Enter. Notice that after you pressed Enter, the name of the current workbook was displayed on the next line. You probably noticed Intellisense appears after the period (.) is entered between ThisWorkbook and Name.

Intellisense displays a drop-down list of applicable properties and methods given a VBA object. For example, ThisWorkbook is a Workbook object. The items listed in the drop-down box are the properties and methods (collectively referred to as members) of a Workbook object. Properties are represented by the hand and index card icon, while methods have a “flying square” icon next to them.

For now, it’s not important to know the difference between properties and methods, but it would be good to start mentally paying attention to the fact there is a difference. Figure below shows an example of Intellisense in the Immediate window.



This is important because it is a very good way to learn about and experiment with VBA statements and the various object models.You can also use the Immediate window to execute VBA statements — rather than displaying the value of a given property, you can change the property’s value. For example, changing the Visible property of a Worksheet object using the Properties window in conjunction with the Project window. You could also perform this feat using the Immediate window. Try this:

1. In the Immediate window enter:

?ThisWorkbook.Worksheets(2).Visible

1. Press Enter. Notice that the value of the Visible property is displayed.
2. In the Immediate window enter (without a question mark):

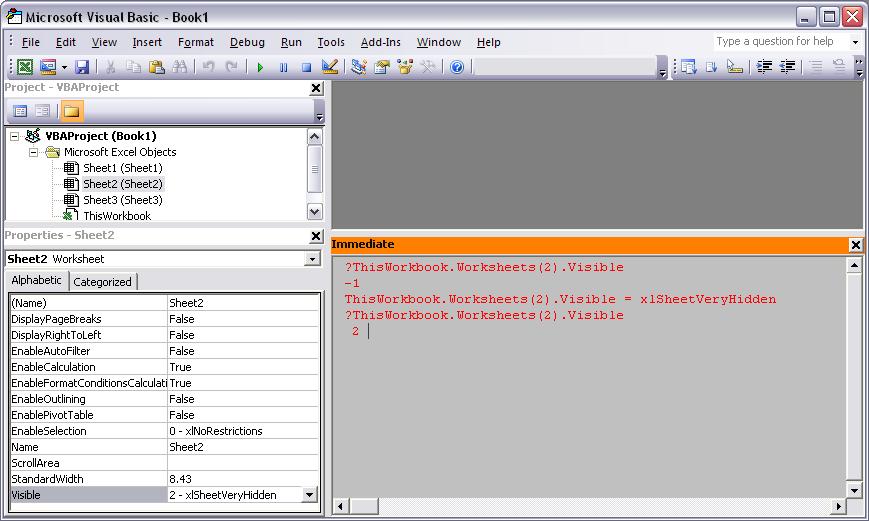
ThisWorkbook.Worksheets(2).Visible = xlSheetVeryHidden

1. Switch to Excel and notice that the worksheet is very hidden.
2. Switch back to the VBE and enter:

?ThisWorkbook.Worksheets(2).Visible

1. Notice that the Visible property has been changed.

An example of this exercise is shown in Figure below:



The final use of the Immediate window is as an output window using Debug.Print when executing code. Typically you would display diagnostic information in the Immediate window. This can be preferable to using the MsgBox function because if there are several things you want to display, the MsgBox method can grow tiresome rather quickly. As an example, try the following exercise.

1. If there is text in the Immediate window, you can clear it all by putting the cursor in the Immediate window, pressing Ctrl + A, and then pressing Delete.
2. Insert a module by selecting **Insert**/**Module**.
3. Enter the following procedure in the module:

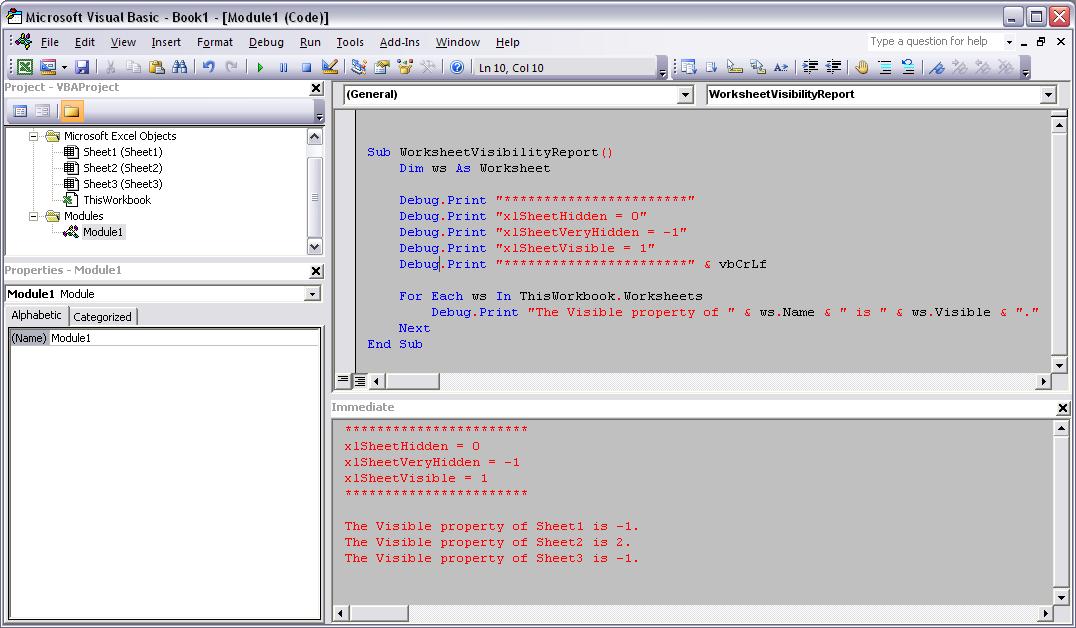
Sub WorksheetVisibilityReport()  
 Dim ws As Worksheet

Debug.Print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"  
 Debug.Print "xlSheetHidden = 0"  
 Debug.Print "xlSheetVeryHidden = -1"  
 Debug.Print "xlSheetVisible = 1"  
 Debug.Print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" & vbCrLf

For Each ws In ThisWorkbook.Worksheets  
 Debug.Print "The Visible property of " & ws.Name & " is " & ws.Visible & "."  
 Next  
End Sub

1. To execute the procedure, place the caret anywhere inside the procedure and press F5.

A completed example of this exercise is shown in Figure below:



Module 1 – Building Blocks in VBA

Modular and Event Driven Concepts

VBA is a structured programming language, where individual statements are defined using the various building blocks of VBA such as objects, methods, and properties. These VBA statements are grouped in larger blocks called **procedures**. A procedure is a named group of statements that run as a unit to perform a specific task or calculate a specific result.

The first step to learning how to create procedures is to learn about the building blocks.

Objects  
VBA is an object-oriented programming language, which means the statements created in VBA act on specific objects rather than being general commands. The office application and individual office document is made of objects that you can manipulate through the VBA statements. In fact a document is an object itself, as are the elements/components that form it. There are many more types of objects defined in the object model.  In many cases an object (e.g., Document) is a container for the collection of other objects (e.g., worksheet, cells, etc.).

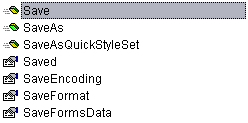
Collections  
Collection is the group of all like objects contained in a parent object.  For example, a document object functions as the container object for the collection of many other objects such as worksheet objects, or cells objects.  As worksheet object contains collections of cells.  A VBA statement that makes reference to an object in a collection uses the objects name or index.  For example you might reference a named worksheet object or an indexed worksheet object:

            'Return the name an open workbook in the worksheets collection named   
            MsgBox Worksheets("Sheets").Name  
            'Return the name of a active worksheet in the current workbook  
            MsgBox ActiveWorksheet.Name

MethodsMethod is an action that can be performed on an object. VBA objects are separated from their methods by periods. For example, if you wanted to save a particular file as part of a VBA program you could include the following statement in the code:

             Workbooks(“Exercise.xlsm”).Save

Remember, a method is an “action” that you perform on an object. The following screen shot shows some of methods and properties (discussed later) that are associated with the ActiveDocument in MS-Word object (using the Auto List Members feature). Notice the symbol for a method looks a bit like a hurled green brick.



Properties  
Properties are used to describe an object. Some properties are read-only, while others are read/write.  For example, the document VBA Basics.docm is saved to a particular path on my computer. That path is a property of the document object. The .Path property is read-only as it cannot be changed, without saving the file to a different location. Properties are separated from objects by periods just as methods are. The following sentence will display the current path of “VBA Basics.docm”, as it is defined on my computer, in an onscreen message box:

  MsgBox Documents(“VBA Basics.docm”).Path

If the property is read/write and you set the property equal to something, it changes the current value of that particular property and therefore changes the description of the object. Otherwise VBA can tell you the properties current value. For example, the following statements change the .Name property of the selected text to “Times New Roman” then reports the .Name property applied:

            Selection.Font.Name = "Times New Roman"  
            MsgBox Selection.Font.Name

Also, in the example above, "Selection" is a global property of the global application property of a document object. It returns a selection object that represents the "selected range" or the insertion point in a document. Global objects or properties are top level and does not need to be preceded by the parent object as the following code statements should illustrate:

            MsgBox Selection.Font.Name  
            MsgBox Application.Selection.Font.Name  
            MsgBox ActiveDocument.Application.Selection.Font.Name

Each of the statements returns the same value.  As the “Selection” property is Global it does not require a reference to the "Application" property or the “ActiveDocument” object.

Functions  
Functions provide information or perform calculations that are useful in building VBA procedures. In the previous examples, the MsgBox function was used to display information on the screen. Other examples of functions include returning the current date or time, or converting data types:

            MsgBox Date 'Date function returns current system date  
            MsgBox Time 'Time function returns current system time  
            MsgBox Val("1") + Val("1") 'Val function returns numbers in a string as a ' numeric value of appropriate type

Events  
"Event" is an action initiated either by user action or by other VBA code. An "Event Procedure" is a Sub procedure that you design according to the specification of the event. The procedure is called automatically when the event occurs. For example, a Document object has events named Open. If you have properly programmed the event procedure for the Open event, Word will automatically call that procedure, always named Document\_Open and always located in the "ThisDocument" module of the project, whenever the document is opened.  In the following example the Document\_Open event procedures stores a value of the current time.  The Document\_Close event procedure compares the stored value with the current time and reports the time the document was opened:

Dim tStart As Date 'Module level declaration

Private Sub Document\_Open()  
            tStart = Now  
End Sub

Private Sub Document\_Close()  
            Dim tStop As Long  
            tStop = DateDiff("s", tStart, N/ow)  
            MsgBox "This document has been open for: " & \_

ConvertTimeString(tStop, True)  & "."  
End Sub

Storage and Organization

A "Project" (document or template) is the top level VBA container for storage and organizing your VBA solutions.  A project consists of one or more modules.

Modules

A module can be one of four types:

1. Code module (or Standard module)
2. Document module (i.e., ThisDocument)
3. MSForm module (or UserForm module), or
4. Class module.

Code within a module consists of individual lines of code (or statements). There are three types of statements

1. Declaration
2. Assignment
3. Executable

Module level statements appear at the top of the module before any procedures (discussed later).

Dim oRng as Word.Range 'Declaration statement  
 Set oRng = ActiveDocument.Range 'Assignment statement  
 oRng.Delete 'Executable statement

Variables/Constants

Variables/Constants are used to store information temporarily. As a procedure is executed, it holds values temporarily in memory. Variables/constants define the name and data type that the procedure associates with specific locations in memory. Sometimes this information will change during the execution of the code (variable) and sometimes it will be static (constant).

Each variable/constant has a specific type that indicates how much memory the data requires and the operations that can be performed on that kind of data.

Dim (stands for dimension) and Const (stands for Constant) statements are used to declare variables and constants and allocate storage space. They can appear in a General Declarations section at the top of a code module -or- immediately following a procedure declaration. For example:

Sub ConstantVariableDemo()  
            Const pStr As String = "ABCD" 'Constant value that doesn't change during ' execution  
            Dim i As Long 'Variable value that changes during execution (from 1 through 4)  
            For i = 1 To Len(pStr)  
                MsgBox Mid(pStr, i, 1)  
            Next i  
            For i = 1 To Len(pStr)  
                MsgBox Left(pStr, i)  
            Next i  
End Sub

Declaring Variables/Constants

You should make a point to include an explicit "type" clause (i.e., *... as Integer* -or- *... as String*) when declaring variables and constants. Valid data types are Byte, Boolean, Integer, Long, Currency, Single, Double, Date, String(*variable-length text*), String (*fixed-length text*), Object, Variant, a user-defined type, or a specific object type. A table summarizing the available data types is include at the end of this tips page.

When you declare variables, you should choose meaningful variable names that describe the variable's purpose. Variable names must meet the same criteria as procedure names. There are 3 levels at which we can declare or dimension (Dim) variables/constants. These are:

1. Procedure-Level
2. Module-Level (Private)
3. Project-Level or Public Module-Level

In each of these levels the variable/constant differs in scope and lifetime. This is discussed below:

***Procedure-Level -*** These are probably the most widely used. They are declared inside the Procedure itself using the Dim or Const statements. See example below:

  Sub ProcLevelDeclaration()  
            Dim i As Long  
            Const pText As String = " seconds and counting"  
            For i = 10 To 1 Step -1  
                MsgBox i & pText  
            Next i  
            MsgBox "Blast off"  
  End Sub

Variables/constants declared at the procedure level are not available to other procedures and they only retain their values for the life of that procedure. As soon as the procedure finishes, the variable/constant and its value are destroyed. This refers to a variable's/constant's scope.

***Module-Level (Private) -*** These are variables/constants that are declared outside the individual procedures at the top of the module. See example below:

Option Explicit

Private Cnt As Long  
Private Const pStr As String = "Testing "  
Sub Procedure1()  
            For Cnt = 1 To 4  
                MsgBox pStr & Cnt  
                If Cnt = 4 Then  
                    Cnt = 3  
                    Exit For  
              End If  
            Next Cnt  
            Procedure2  
End Sub

Sub Procedure2()  
            For Cnt = Cnt To 1 Step -1  
                MsgBox pStr & Cnt  
            Next Cnt  
            MsgBox "Test Complete"  
End Sub

Variables/Constants declared using the Private or Private Const statements at the module-level (or within a form's General Declarations section) are available to all procedures within that module or form and they retain their assigned values, unless the Document closes or the End statement is used. However, these variables are not available to procedures outside the module in which they are declared.

***Project-Level, Document Level, or Public Module-Level*** *-* These are variables/constants that are declared "Public" at the module-level (or within a form's General Declarations section). See example below:

Option Explicit  
Public Counter As Long  
Public Const pString As String = "Testing "  
  
Sub ThisModProcedure1()  
            For Counter = 1 To 4  
                MsgBox pString & Counter  
                If Counter = 4 Then  
                    Counter = 3  
                    Exit For  
                End If  
            Next Counter  
            'Call procedure in another module  
            OtherModule.Procedure1  
End Sub

Variables/Constants declared as Public at the module level are available to all procedures, in all modules within the same project the variables are declared in. Their values are retained unless the Document closes or the End Statement is used.

***Static Declaration -*** Procedures and variables/constants may also be declared using the "Static" statement.

When the Static statement is used to declare a procedure, the procedure's variable remains in scope and retain their values until the document closes or the End statement is used:

Sub RunStaticProcDemo()  
            'Attempt Demo 9 times    
            Demo  
            Demo  
            Demo  
            Demo  
            Demo  
            Demo  
            Demo  
            Demo  
            Demo  
 End Sub

Static Sub Demo()  
'Each time run the value the variable i is preserved  
Dim i As Long  
            i = i + 1  
            MsgBox i  
            'When i grows to a value > 6 stop all code execution  
            If i > 6 Then End  
End Sub

When the Static statement is used to declare a variable, the variable remains in scope and retains its values until the document closes or the End statement is used. Run the following procedure several times as an example:

Sub CostOfPurchase()  
            'Static variables remain in scope & retain their values after procedure is run.  
            'Declare variables  
            Static sngTotal As Single  
            Dim sngCostThisItem As Single  
            sngCostThisItem = CSng(InputBox("Enter the cost of a purchase:"))  
            sngTotal = sngTotal + sngCostThisItem  
            'Display results  
            MsgBox "The cost of a new purchase is: " & sngCostThisItem  
            MsgBox "The running cost is: " & sngTotal  
End Sub

Class Model

A **Class** is the formal definition of an **Object**. The class is a template for the creation of the object during programming, and defines the properties and methods that decide how the object behaves. Class module is new in Office97. With class modules it's possible to create your own objects. These objects can have properties and methods like the built-in objects, and you can create several copies of an object if you need to.

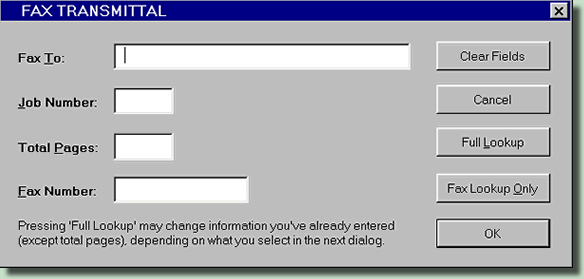
In VBA, a class is defined in class module and serves as a template for an object. The term object is deliberately vague. An object can be defined to represent whatever you want. Anything that you can describe conceptually can be represented by a class. The difference between a class and an object is that a class does nothing and consumes no memory. When you have a variable of that class type and create instance of that class with the **New** keyword, a process called instantiating, it becomes an object and consumes memory and can carry out **Actions**. A class is defined by its **Properties**, which describe **Attributes** of the class, and its **Methods** (sub and function procedures), which carry out actions in the object. If a class is analogous to a noun, a property is like an adjective -- it describes the object. A method is like a verb -- it carries out an action.

You must instantiate a class into an object in order to do anything with it. There is nothing you can do with a class module beyond creating an object from it

User Form

A userform is a ***custom dialog,*** a graphical pop-up window that collects and processes input from the user. It looks and acts just like any of the familiar ***built-in dialogs,*** such as the ones you see when you click File–›Print or Format–›Font, except that you design it yourself and tailor its elements to suit the needs of the job it performs.

Here's a screenshot of a simple userform:



You create a userform in a graphical window of the VBE***,*** where you lay out its boxes, buttons, and other "controls". You also use the Visual Basic Editor to write the code that makes the userform work.

Userforms can employ any of a huge array of graphical "controls" — command buttons, textboxes, checkboxes, listboxes, option buttons ("radio buttons"), scrollbars, and multi-page structures

Object Model

To use VBA with an application such as Access, Word, or Excel, terminology and language constructions are needed to interact with the application. This portion of VBA is called the **Object Model** for the application. A map of the object model is online for [Excel](http://msdn.microsoft.com/en-us/library/bb149081%28v=office.12%29.aspx) (http://msdn.microsoft.com/en-us/library/bb149081%28v=office.12%29.aspx) and for [Word](http://msdn.microsoft.com/en-us/library/bb244515%28v=office.12%29.aspx) (http://msdn.microsoft.com/en-us/library/bb244515%28v=office.12%29.aspx) A listing of the object model is found by opening the Macro/VBA editor in the target application and then using "View" to open the "Object Browser" (F2).

Much of the difficulty in using VBA is related to learning the object model, which uses names invented by the originators of the model that may be less than transparent to a new user. One way to learn the terms and syntax of the object model is to use the **Macro Recorder** to record the steps taken to achieve a desired result using the mouse and menus of the application. Once this is done, the VBA code constructed by the recorder can be viewed in the VBA editor, and often greatly streamlined or generalized with only a modicum of understanding of VBA itself. The macro recorder does not always record everything (particularly for graphs), and some applications employing VBA do not provide a recorder at all. Use of debugging tools to discover VBA constructs for some cases where the macro recorder does not work.

Sometimes macro recording can be even totally misleading. You can write very clean and proper code if you think more BASIC than EXCEL.

**Example:** create a table 5 rows x 2 columns containing 1..5 / A..E

Sub Sample()   
'   
' Sample Macro   
' Macro recorded 25.11.2010 by   
'   
  
    ActiveCell.FormulaR1C1 = "1"   
    ActiveCell.Offset(1, 0).Range("A1").Select   
    ActiveCell.FormulaR1C1 = "2"   
    ActiveCell.Offset(1, 0).Range("A1").Select   
    ActiveCell.FormulaR1C1 = "3"   
    ActiveCell.Offset(1, 0).Range("A1").Select   
    ActiveCell.FormulaR1C1 = "4"   
    ActiveCell.Offset(1, 0).Range("A1").Select   
    ActiveCell.FormulaR1C1 = "5"   
    ActiveCell.Offset(-4, 1).Range("A1").Select   
    ActiveCell.FormulaR1C1 = "A"   
    ActiveCell.Offset(1, 0).Range("A1").Select   
    ActiveCell.FormulaR1C1 = "B"   
    ActiveCell.Offset(1, 0).Range("A1").Select   
    ActiveCell.FormulaR1C1 = "C"   
    ActiveCell.Offset(1, 0).Range("A1").Select   
    ActiveCell.FormulaR1C1 = "D"   
    ActiveCell.Offset(1, 0).Range("A1").Select   
    ActiveCell.FormulaR1C1 = "E"   
    ActiveCell.Offset(0, 2).Range("A1").Select   
    ActiveCell.FormulaR1C1 = "2"   
End Sub

Ugly right? In fact it can be rewrite with the more powerful VBA code as listed below:

Sub CreateTable()   
Dim MyRange As Range, Idx As Long   
   
    Set MyRange = Selection   
   
    For Idx = 1 To 5   
        MyRange(Idx, 1) = Idx   
        MyRange(Idx, 2) = Chr(Asc("A") + Idx - 1)   
    Next Idx   
   
End Sub

|  |
| --- |
|  |

Module 2 – Parameter Passing Mechanisms

Parameter Passing Mechanisms

When you define a procedure, you have two choices regarding how arguments are passed to it: by reference or by value. When a variable is passed to a procedure by reference, VBA actually passes the variable's address in memory to the procedure, which can modify it directly. When execution returns to the calling procedure, the variable contains the modified value.

When an argument is passed by value, VBA passes a copy of the variable to the procedure. Then, the procedure modifies the copy, and the original value of the variable remains intact; when execution returns to the calling procedure, the variable contains the same value that it had before being passed.

By default, VBA passes arguments by reference. To pass an argument by value, precede the argument with the **ByVal** keyword in the procedure definition, as shown here:

Function SomeProc(strText As String, ByVal lngX As Long) As Boolean

If you want to denote explicitly that an argument is passed by reference, you can preface the argument with the **ByRef** keyword in the argument list.

Passing by reference can be useful as long as you understand how it works. For example, you must pass arrays by reference; you will get a syntax error if you try to pass an array by value. Because arrays are passed by reference, you can pass an array to another procedure to be modified, and then you can continue working with the modified array in the calling procedure.

Optional Parameter

The majority of procedures (**Sub** and **Function** procedures) use a fixed number of parameters and most often these parameters have explicit data types. This is all well and good, but there may be circumstances in which the number of parameters cannot be known until run-time. There are two methods that you can use to handle a variable number of parameters:

1. **Optional Variants**, defines a fixed number of parameters but makes these parameters optional, effectively putting an upper limit on the number of parameters than may be passed.
2. **ParamArray** type parameter allows for any number of parameters, including none. In either method, if your procedure requires some parameters but not others, declare all the required parameters first and then declare the optional parameters with either the **Optional** attribute or with a **ParamArray** type parameter after declaring the required parameters.

Optional Variants

You can declare one or more parameters as optional parameters of the **Variant** data type. With these optional variant parameters, you can use the **IsMissing** function to determine whether a parameter was included or omitted. There are a few rules that govern the use optional parameters:

1. The **Optional** keyword must be present to make a parameter optional.
2. The data type should be (but need not be, see below) a **Variant** data type.
3. The optional parameter(s) must be at the end of the parameter list.
4. The **IsMissing** function will work only with parameters declared as **Variant**. It will return **False** when used with any other data type.
5. User defined types (UTDs) cannot be optional parameters.

We'll examine each of the rules. First, the keyword **Optional** must be used in the parameter declaration. This keyword is what makes the parameter optional. Without it, the parameter is required. For example,

Function Test(L1 As Long, L2 As Long,

Optional P1 As Variant, Optional P2 As Variant) As String

In this example, **L1** and **L2** are required parameters and **P1** and **P2** are optional parameters. Since P1 and P2 are Variant types, we can use the IsMissing to determine whether the parameters were passed to the procedure.

The data type should be a Variant type varible. Actually, this is only partially true. If you need to test specifically whether a parameter was actually passed to the procedure, you must declare it as a Variant so you can use the IsMissing function to determine if the parameter was passed. IsMissing may be used only with Variant types. It will return False for any other data type, even it that parameter is declared as Optional and is in fact missing. An optional parameter that is not a Variant will be assigned the default value for that data type (0 for numeric data types, and empty string for **String** types, and **Nothing** for all object type variables. For example,

Function Test(L1 As Long, L2 As Long, Optional P1 As Variant, Optional P2 As Variant) \_ As String

Dim S As String

If IsMissing(P1) = True Then

S = "P1 Is Missing."

Else

S = "P1 Is Present (P1 = " & CStr(P1) & ")"

End If

If IsMissing(P2) = True Then

S = S & " " & "P2 Is Missing"

Else

S = S & " " & "P2 Is Present (P2 = " & CStr(P2) & ")"

End If

Test = S

End Function

Here, both **L1** and **L2** are required but **P1** and **P2** are optional. Since both are Variant types, we can use IsMissing to determine whether the parameter was passed in. IsMissing returns True of the Variant parameter is omitted, or False is the Variant parameter is included. If the data type of the optional parameter is any data type other than Variant, IsMissing will return False.

However, you can declare optional parameters of any data type and provide a default value to be used if that parameter is omitted. But the IsMissing function works only with Variant data types, so if you use an optional Long with a default value, there is no way to determine whether that parameter was actually passed to the procedure with a value that happens to be the same as the default value, or whether the parameter was omitted. For example,

Function Test2(Optional V As Variant, Optional L As Long = -1) As String

Here, both V and L are optional, but only V is a Variant type, so only V may be tested with the IsMissing function. If L is omitted, IsMissing will return False since L is not a Variant. If omitted, L will have a value of -1. This may well be sufficient, but there is no way to determine whether L was actually passed into the procedure. If you write code such as this, you should choose a default value that will never occur if optional value is passed in. What that default value should be depends entirely on what the procedure does and the operating environment in which it is used.

Optional parameters must be the last parameters declared for the procedure. That is, once one parameter is declared as Optional only other Optional parameters may follow it. (This is not the case for Property procedures. See the section on Property Procedures later in this article.) The first procedure declarion shown below (GoodFunction) is valid because all the optional parameters are at the end of the parameter list. The second procedure declaration (BadFunction) is invalid because a required parameter (L2) follows an optional parameter (M1). You will get a compiler error (*"Expected: Optional"*) if you attempt to use code like the BadFunction declaration.

Function GoodFunction(L1 As Long, L2 As Long, \_

Optional M1 As Variant, Optional M2 As Variant) As Variant

Function BadFunction(L1 As Long, Optional M1 As Variant, \_

L2 As Long, Optional M2 As Variant) As Variant

A User Defined Type (UDT -- a structure declared with the **Type** keyword) cannot be an optional parameter, nor can a UDT be an element in a ParamArray parameter type. Code that attempts to do this will cause the compiler to emit an error.

Optional Arguments In Property Procedures

The rules for working with Optional parameters and ParamArray variables are different when working with Proprerty procedures in class modules. Typically, a Property Get declarations take no parameters an return a single value. A typical Property Let procedure take one parameter, the value of that property. However, it is perfectly legal to use optional parameters in a procedure's declaration. For example, the following code is valid:

Public Property Get Value(Optional A As Integer, Optional B As Integer) As Long

Value = pValue + A + B

End Property

Public Property Let Value(A As Integer, B As Integer, V As Long)

pValue = A + B + V

End Property

A few things are noteworthy. The parameter declarations of the Get procedure must match the parameter declaration of Let, but the last (or only) parameter to the Let procedure must match the return data type of the **Get** procedure. You can see this in the code above in which the return type of the **Get Value** procedure is a **Long**, the same as the last parameter of the Let procedure (parameter V). You'll also notice that the parameters **A** and **B** of the Get procedures are declared Optional while they are required in the **Let** procedure. Their data types (**Integer** in the example) must match, but they need not match in optional.

Like the optional parameters shown above, a Property Get/Let pair may use a ParamArray parameter. As before, the variable types of the Get/Let pair must match. For example,

Public Property Let Value2(ParamArray Args() As Variant, V As Long)

' some code

End Property

Public Property Get Value2(ParamArray Args() As Variant) As Long

' some code

End Property

Note here than the last argument to the Let procedure (parameter V) is the same type as the return type of the Get procedure. In practical terms, you will likely use the ParamArray array in a Property Let procedure than you would in a Property Get procedure, but just be aware that the ParamArray must exist in both procedures even if you use it only in one.

This differing behavior between Property Get/Let procedures and regular Sub and Function procedures (such as having optional parameters occurring before required parameters) becomes clear when you see how those properties are used in code. Using the Value2 Get/Let properties shown above, we would access these properties with code like the following:

Dim C As Class1

Dim L As Long

Set C = New Class1

' LET Value2

C.Value2(1, 2, 3) = 5

' GET Value2

L = C.Value2(11, 22, 33)

In the Let procedure, the ParamArray applies to the values within the parentheses (1, 2, 3), and the actual property assignment variable (V in the declaration, 5 in the sample code) is the last parameter declared for the Let procedure. If you find yourself declaring multiple parameters for a Get or Let procedure, you may want to consider changing it from a property to a method of the class.

Named Parameter

When a call is made to a Sub or Function Procedure, you can supply Arguments in the exact order they appear in the Procedure's definition, or you can supply them in any position by name. To illustrate this point, I'll use a fictitious Function called fCalculateInterest() which accepts 3 Arguments (Currency, Single, and Long) and returns a value of type Currency.

Public Function fCalculateInterest(curPrincipal As Currency, sngRate As Single, \_

lngTermInMonths As Long) As Currency   
  fCalculateInterest = curPrincipal \* sngRate \* lngTermInMonths   
End Function

A simple Call to this Function would be:

Dim curInterest As Currency   
curInterest = fCalculateInterest(100000,.0725, 240)

The above Function was called by supplying its Arguments in the correct positions, each delimited by a comma. If the positions of the Arguments change, you may either get a Data Type mismatch Error, or erroneous results. By the use of Named Arguments, you can supply the Arguments by name without any regard to position. A Named Argument consists of an Argument name followed by by a Colon and an Equal Sign (:=). We will now call that very same Function but this time reversing the positions of Arguments:

Dim curInterest As Curremcy   
curInterest = fCalculateInterest(lngTermInMonths:=240, sngRate:=.0725, \_

curPrincipal:=100000)

Named Arguments are extremely useful when you are calling a Procedure that has Optional Arguments. If you use Named Arguments, you don't have to include commas to denote missing position Arguments, if you don't use them, you must. The following Sub Procedure will illustrate my point, It accepts 4 Arguments, 3 of which are Optional (String, Long, Boolean).

Private Sub TestArgs(Arg1 As Integer, Optional Arg2 As String, Optional Arg3 As Long, \_

Optional Arg4 As Boolean)   
End Sub

A typical call to this Sub Procedure would be as follows:

Call TestArgs(1234, "Argument2", 999999, False)

If you wanted to call TestArgs without passing the 2nd and 3rd Optional Arguments, and without using Named Arguments, the syntax would be:

Call TestArgs(1234,,,False)

If you wanted to call TestArgs without passing the 2nd and 3rd Optional Arguments, and use Named Arguments, without regard to the positions of the Arguments, the syntax would be:

Call TestArgs(Arg4:=False, Arg1:=1234)

**NOTE:** As a more realistic and practical example, and as suggested by Mary, I've included an example using the popular SendObject() Method. All Arguments of this method were used, except [TemplateFile], and their standard positions within this Method were modified. This was only possible by using this weeks Tip, Named Arguments:

DoCmd.SendObject OutputFormat:=acFormatXLS, ObjectName:="tblEmployeePFD", \_   
ObjectType:=acSendTable, Subject:="Named Arguments", Bcc:="Joe Dunn", \_    
To:="Tom Jones", CC:="Chris Myers", EditMessage:=False, \_ MessageText:="Demonstration of the Usage of Named Arguments"

It is very advantageous to use Named Arguments, especially when utilizing Optional Arguments.

Variable Parameter with ParamArray

By using the Optional keyword, you can declare a function that accepts both required and optional parameters, but you are still limited to a fixed number of parameters. For example, the declaration

Sub Test(L1 As Long, Optional P1 As Variant, Optional P2 As Variant)

allows P1 and P2 as optional parameters, but this function is still limited to a maximum of three parameters -- you can't pass in four or more parameters. Moreover, each optional parameter must be tested with IsMissing to determine whether it was passed. This can be onerous if you have a large number of optional parameters.

The solution to this problem is the ParamArray parameter type. A ParamArray allows any number of parameters, including none at all, to be passed into a procedure. Note that a ParamArray allows for optional parameters following any number of required parameters. Only the ParamArray elements are option. Any parameters declared before the ParamArray are required. There are a few rules for using a ParamArray:

The ParamArray variable must be an array of Variant data types.

The ParamArray variable must be the last parameter in the parameter list. (The reason for this is that were another parameter to follow the ParamArray, it would be impossible for the code to determine where the ParamArray variables end and the subsequent variables begin.)

No Optional parameters may appear in the parameter list. That is, the usages of Optional and ParamArray are mutually exclusive. You can have one or neither, but not both.

The function must declare at most one ParamArray variable. It is illegal (and nonsensical if you really think about it) to have two ParamArray parameters declared for a procedure.

Since a ParamArray parameter is an array, it is always passed by reference (ByRef) so any changes in the called procedure (which declares the ParamArray) to the elements in the ParamArray are changed in the calling procedure.

The base of the ParamArray array is 0, regardless of the setting of the module's Option Base statement. If zero parameters are passed, the LBound is 0 and the UBound is -1.

A User Defined Type (UDT) cannot be an element of a ParamArray parameter type.

The following is an example of a procedure declaration that accepts one required parameter, L1, and then accepts any number of additional parameters in the ParamArray Args() parameter.

Sub Test(L1 As Long, ParamArray Args() As Variant)

Since the ParamArray Args() parameter is an array, you can use normal array methods on the array. For example, you can loop through the elemets:

Function SumUp(ParamArray Args() As Variant) As Double

Dim Sum As Double

Dim Ndx As Long

For Ndx = LBound(Args) To UBound(Args)

Sum = Sum + Args(Ndx)

Next Ndx

SumUp = Sum

End Function

Since the data type of the ParamArray Args() As Variant declares Variant data type, the elements of Args may contain anything, including numbers, strings, objects, and arrays, in addition to being Empty or NULL. Unless you have tight control over what is being passed in the array, your code should check the type of each Args element to ensure it is valid. For example, a better version of the SumUp function above would be:

Function SumUpBetter(ParamArray Args() As Variant) As Double

Dim Sum As Double

Dim Ndx As Long

For Ndx = LBound(Args) To UBound(Args)

If IsNull(Args(Ndx)) = False Then

If IsArray(Args(Ndx)) = False Then

If IsObject(Ndx) = False Then

If IsError(Ndx) = False Then

Sum = Sum + Args(Ndx)

End If

End If

End If

End If

Next Ndx

SumUpBetter = Sum

End Function

While all this testing may seem like overkill, you must remember that with a Variant, you have no control over what that variable might contain and your code should test for possible errors.

You cannot have optional arguments within the ParamArray itself. If you omit an argument in a ParamArray an error is placed in that element of the array. For example,

Function FFF(ParamArray Args() As Variant) As Long

Dim N As Long

Dim D As Double

For N = LBound(Args) To UBound(Args)

If IsError(Args(N)) = True Then

Debug.Print "ERROR In ParamArray At Element: " & CStr(N)

Else

D = D + Args(N)

End If

Next N

FFF = D

End Function

Call this function with code like the following:

Sub AAA()

Dim D As Double

D = FFF(1, , 3, 4)

Debug.Print D

End Sub

This will illustrate that the second value passed in the ParamArray (the missing element) is treated as an error by the code in function FFF.

The array variable declared as a ParamArray has a LBound value of 0, regardless of the module's Option Base setting. If no parameters were passed via the ParamArray variable, the LBound of the ParamArray is 0 and the UBound of the ParamArray is -1. The number of elements in the ParamArray can be determine code similar to the following:

Function FFF(ParamArray Args() As Variant) As Long

Dim NumArgs As Long

NumArgs = UBound(Args) - LBound(Args) + 1

FFF = NumArgs

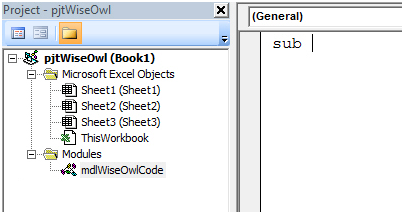
End Function

Module 3 – Subroutines, Functions, Event Handlers and Macro

This module is dealing with the building blocks that handle the logic of the VBA programs. As VBA developer, it is important to differentiate various types of these building blocks, and know when to use each of them.

Creating a Subroutine

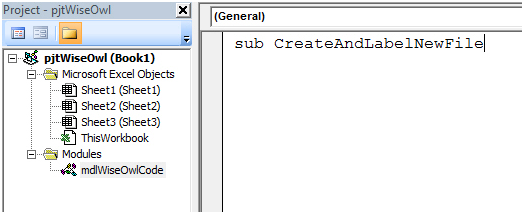
The most common type of program you're likely to write in VBA is called a **Subroutine**, or ***S*ub** for short.  You can start writing a subroutine by typing the word **Sub** at the top of the module you created earlier (we're using Excel for this demonstration).



*Type in the word* ***sub****, followed by a space to get started with your subroutine.*

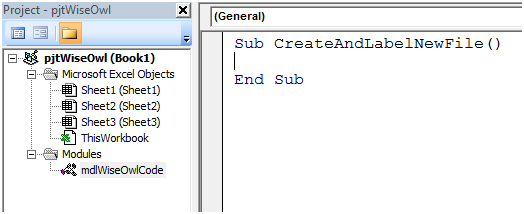
**Note:** You'll hear lots of words used to refer to a subroutine - program, procedure, macro, and sub for instance. They all mean essentially the same thing - a set of instructions that an application will follow when you run the subroutine.

The next thing you need to do is type in a name for your subroutine.  Names for subroutines follow the same rules as names for projects and modules: don't use spaces and try to avoid punctuation characters.  The program we're going to write in this first example will simply create a new Excel workbook and write a title into cell A1 on the first worksheet.  We'll call our subroutine **CreateAndLabelNewFile**.  Type this into your module after the word **sub**.



Notice we've typed in the word **sub** in lowercase letters, but used uppercase letters to start each word in the name of the subroutine. This is a common convention used in VBA programming to make it easier to read the names of things.

The only other thing you need to do to create your subroutine is to press Enter on the keyboard, so do just that!



Congratulations, you've created a subroutine!  Obviously it doesn't actually do anything yet, but we're getting to that part.

Assuming that you didn't see an error message, several things should happen automatically when you press the Enter key:

* The word **sub** is capitalised to **Sub**.
* The word **Sub** is coloured in dark blue.
* A set of parentheses is added to the end of the subroutine name.
* The words **End Sub** appear on the line below your text cursor.

All of these things are important, but the VB Editor does the work for you - it's like a more helpful and fancy version of the Notepad application.

*If something went wrong and you saw an error message when you pressed Enter, the two most likely reasons are:*

*1. You put a space or other disallowed character in the name.*

*2. You typed in a name that is reserved by the VBA language.*

|  |  |
| --- | --- |
|  | If you see this error message, just click OK and make sure that your sub's name doesn't contain any spaces or punctuation characters.  If that doesn't work, try changing the name of your sub and try again. |

So far, the code that we've written doesn't actually perform any useful actions.  That's something we'll solve in the next article in this series, but first you should learn a little about how to lay out your code neatly.

The Importance of Neat Code Layout

Once you've created your subroutine, you can start typing the VBA code on the very next line.  The problem with doing this is that when you come back to look at your code later you'll find yourself staring at a horrible looking chunk of text that's very difficult to read.

You can make your life much easier by using blank lines to separate parts of your code, and indenting text within a subroutine to make it easier to see where it starts and ends.  To do this, once you've created your subroutine by pressing Enter, press Enter once more to create a blank like, and then press Tab on the keyboard to indent your code.

|  |  |
| --- | --- |
|  |  |
| *Your text cursor should start here...* | *...and end here.* |

It may seem like a pedantic thing to do, and technically speaking it's not even necessary, but it will definitely make your life easier in the long run.  Trust us!  You can hopefully see the benefits of this approach by comparing the two screenshots below (even if you don't understand the actual code!).

|  |  |
| --- | --- |
| *Code without layout* | *Code with layout* |
| *Without a neat layout to the code it's extremely difficult to work out what's going on in this subroutine.* | *The simple addition of a few blank lines and indents makes the whole subroutine much easier to read.* |

Adding Comments to Your Code

Another thing that will help you enormously when looking at your code later is a series of comments that explain what your program does.  You can add a comment either on a separate line or at the end of a line of code by typing an apostrophe.  You can then type whatever you like and press Enter at the end of the line to create the comment.

|  |  |
| --- | --- |
| *Creating a comment* | *The finished comment* |
| *After the apostrophe, type in your comment.* | *Press Enter and the text will turn dark green to indicate that it's a comment.* |

Using Sub-Routines and Functions That Take Arguments

Considering those two helpful articles, let's look at a simple example:

Sub Main\_Procedure()  
            Dim lngArg As Long, Dim strArg As StringlngArg = 1  
            strArg = "I am what I am."  
            MsgBox "Passing: " & lngArg & " - " & strArg  
            CalledSub\_Routine1 lngArg, strArg  
            MsgBox "After Passing: " & lngArg & " - " & strArg  
            End Sub

            Sub CalledSub\_Procedure(lngPar As Long, strPar As String)  
            lngPar = lngPar + 1  
            strPar = "I ain't what I used to be."  
 End Sub

Here we have a main procedure that calls a sub-procedure.  The main procedure has two variables lngArg and strArg declared.  Both are passed as arguments to a called procedure.  The variable values are reported before and after the call to the sub-procedure.  The sub-procedure accepts the arguments as parameters and changes the parameter values.

**Note:** A calling procedure "passes" arguments while a called procedure "accepts" parameters.  I used "Arg" (for argument) in declaring variables in Main\_Routine (or calling procedure) and "Par" (for parameter) in CalledSub\_Routine (or called procedure) simply to illustrate this point.  In some VBA text you may see "argument" and "parameter" used interchangeably.

ByRef/ByVal

When you ran the example, did you notice how the variable values in the main procedure were altered by the changes made to the parameters by the called procedure?

This is the default "ByRef" behavior when passing arguments where a reference to the memory address storing the variable data is passed and changes made to the parameter result in changes to data in the memory address.  Since "ByRef" is the default behavior you don't have to explicitly use "ByRef" in your code, but it is a good practice.  The following statements are functionally the same:

            Sub CalledSub\_Procedure(lngPar As Long, strPar As String)  
            Sub CalledSub\_Procedure(ByRef lngPar As Long, ByRefstrPar As String)

When ByVal is used the actual value of the argument is passed and changes made to the parameter in the called procedure are not reflected in the variable value of the calling procedure as the following should illustrate:

Sub Main\_Procedure2()  
            Dim lngArg As Long, Dim strArg As String  
            lngArg = 1  
            strArg = "I am what I am."  
            MsgBox "Passing: " & lngArg & " - " & strArg  
            CalledSub\_Routine2 lngArg, strArg  
            MsgBox "After Passing: " & lngArg & " - " & strArg  
End Sub

Sub CalledSub\_Procedure2(ByVal lngPar As Long, ByVal strPar As String)  
            lngPar = lngPar + 1  
            strPar = "I ain't what I used to be."  
            MsgBox "lngPar = " & lngPar & " - strPar = " & strPar  
 End Sub

When to use which?

Procedures  
Procedures are organization and storage units of a VBA solution.  They are the blocks that get things done.  There are three types of VBA procedures: **Sub**, **Function**, and **Property**. Procedures are executed or run (same meaning) in order to apply their statements. When a procedure is run, its statements are processed in a top-down line by line fashion performing the defined operations.

Procedure names can contain a combination of as many as 254 letters, numbers, and the underscore character (\_). However, variable names cannot begin with a number, nor can you use reserved *keywords* that have special meaning to the VBA compiler.

Sub Procedure

A Sub procedure is a series of one or more VBA statements enclosed by the Sub and End Sub statements that performs an action or actions but doesn't return a value.

The following example illustrates a basic Sub procedure:

    Sub SubExample() 'The Sub statement (Note: All procedures are public by default)  
            With Selection.Font 'Object to act on  
                .Color = wdColorRed 'Set font color property  
                .Size = 14 'Set font size property  
            End With  
    End Sub 'The End statement

A Sub procedure can take "parameters," such as constants, variables, or expressions that are passed to it as "arguments" by another "calling" procedure. The following is an example of a Sub that takes parameters passed as arguments from a calling procedure:

Sub Main()  
            'Other code could go here  
            'Call and pass arguments to another sub

'Property values for font 'color and size are 'passed as arguments  
          FormatFontAtSelection Selection.Range, wdColorRed, 14   
            'Other code could go here  
End Sub

Sub FormatFontAtSelection(ByRef oRng As Range, oColor As Long, oSize As Long)   
            With oRng.Font  
                .Color = oColor  
                .Size = oSize  
            End With  
End Sub

Function Procedure

A function declaration has the form:

[Public or Private] Function FunctionName(Param1 As DataType1, \_

,Param2 As DataType2,...) As ReturnType

A Function procedure is a series of Visual Basic statements enclosed by the Function and End Function statements. It is similar to a Sub procedure, but a function can also return a value. A function can take arguments, such as constants, variables, or expressions that are passed to it by a calling procedure. If a Function procedure has no arguments, its statements must include an empty set of parentheses. A function returns a value by assigning a value to its name in one or more statements of the procedure.

The following is an example of a Function procedure called from a Sub procedure. In the example the Function procedure returns a value declared with data type "Double" to the calling procedure:

Sub ConvertTemp()  
            Dim dblFahTemp As Double 'Variable declaration  
            Dim dblCelTemp As Double 'Variable declaration  
            'Use the Inputbox and CDbl functions to get and convert a user input to

'a double variable data type  
            dblFahTemp = CDbl(InputBox("Enter the temperature in degrees Fahrenheit", "Temperature"))  
            'Get the converted temperature value by passing the Fahrenheit temperature 'as a variable to a Function procedure  
            dblCelTemp = Celsius(dblFahTemp)  
            'Use the MsgBox function to report the results  
            MsgBox dblFahTemp & " degrees Fahrenheit is " & dblCelTemp & \_

" degrees Celsius."  
End Sub

Function Celsius(ByRef dblFahTemp As Double) As Double  
            'Perform calculations to convert the Fahrenheit value to a Celsius value  
            Celsius = (dblFahTemp - 32) \* (5 / 9)  
            'Apply appropriate number formatting  
            If Celsius = Int(Celsius) Then  
                    Celsius = Format(Celsius, "0;-0")  
           Else  
                    Celsius = Format(Celsius, "0.00;-0.00")  
            End If  
End Function

Property Procedure

It is a procedure that creates and manipulates properties for a class module. A Property procedure begins with a Property Let, Property Get, or Property Set statement and ends with an End Property statement.

Return value from Function

Note that you must declare the data types not only of each parameter to the function, but also of the return type. Otherwise, VBA declares these items as variants.

For example, the *AddOne* function in the following example that adds 1 to the original value.

Public Function AddOne(Value As Integer) As Integer

AddOne = Value + 1

End Function

To use the return value of a function, just place the call to the function within the expression in the location where you want the value. For instance, the code:

MsgBox "Adding 1 to 5 gives: " & AddOne(5)

produces the message box below:



Note that in general, any parameters to a function must be enclosed in parentheses within the function call.

To return a value from a function, we must assign the function's name to the return value somewhere within the body of the function.in the example below shows a more complicated example of a function.

Function ReturnCount(OfWhat As String) As Long

' Return count of characters or words in the active document. Return -1 if

' OfWhat is not "Characters" or "Words"

If OfWhat = "Characters" Then

' Return count of characters

ReturnCount = ActiveDocument.Characters.Count

ElseIf OfWhat = "Words" Then

' Return count of words

ReturnCount = ActiveDocument.Words.Count

Else

' Return -1

ReturnCount = -1

End If

End Function

This function returns a count of the number of characters or the number of words in the active document, depending on the value of the parameter to the function. (I will discuss parameters in more detail later in this chapter.) Note that ReturnCount is assigned several times within the body of the function. Its value, and hence the value of the function, is set differently depending upon the value of the parameter. (Since these assignments are mutually exclusive, only one of them will occur each time the function is called.)

Ignoring Return Values

Since functions can do more than just return a value, there are situations where we want to call a function but we don't care about its return value. To illustrate, suppose that rng is a variable that represents a range of text within a Word document. Then we can adjust the end of this range using Word's MoveEnd method. This method is just a function that moves the end of the range a specified number of units and returns the actual number of units moved, which may be less than the specified number if the range "bumps up" to the beginning or end of the document before completion. For instance,

Dim dist as Long

dist = rng.MoveEnd(wdCharacter, 5)

attempts to move the end of rng five characters to the right. If it is successful, it returns the number 5. However, if there are fewer than five characters to the right of the current end position, then the range is moved as many characters as possible and this value is returned.

The return value of the MoveEnd method can often be useful, but there are times when it is not. For instance, the following code selects the first paragraph in the active document, including the ending paragraph mark:

Dim rng As Range

Set rng = ActiveDocument.Paragraphs(1).Range

rng.Select

If we do not want to include the ending paragraph, we can use the MoveEnd method to move the end of the range one character to the left before selecting it, as follows:

Dim dist as Long

Dim rng As Range

Set rng = ActiveDocument.Paragraphs(1).Range

dist = rng.MoveEnd(wdCharacter, -1)

rng.Select

Since there is at least one character to the left of the current end point (the paragraph mark is always present, so the range includes at least this character), we know that the return value dist of the MoveEnd function will be -1. (It is negative since the move is to the left.) Hence, we don't really care about dist.

In previous versions of VBA, we would have had no choice but to declare the variable dist and then just ignore its value, as in the previous code. However, the current version of VBA lets us streamline this code to:

Dim rng As Range

Set rng = ActiveDocument.Paragraphs(1).Range

rng.MoveEnd wdCharacter, -1

rng.Select

Note that the function is called without specifying a return value and without the parentheses surrounding the parameters. This tells VBA to simply discard the return value.

Termination

The Exit Statement

Use **Exit Function** or **Exit Sub** to terminate a procedure in the middle, like this:

Sub lengthy\_computation(fat, n)

If n = 0 Or n = 1 Then

fat = 1

' now terminate

Exit Sub

End If

' now compute fat ...

End Sub

Using Break Mode

At design time, you can change the design or code of an application, but you cannot see how your changes affect the way the application runs. At run time, you can watch how the application behaves, but you cannot directly change the code.

Break mode halts the operation of an application and gives you a snapshot of its condition at any moment. Variable and property settings are preserved, so you can analyze the current state of the application and enter changes that affect how the application runs. When an application is in break mode, you can:

* Modify code in the application.
* Observe the condition of the application's interface.
* Determine which active procedures have been called.
* Watch the values of variables, properties, and statements.
* Change the values of variables and properties.
* View or control which statement the application will run next.
* Run Visual Basic statements immediately.
* Manually control the operation of the application.

**Note:**   You can set breakpoints and watch expressions at design time, but other debugging tools work only in break mode.

Entering Break Mode at a Problem Statement

When debugging, you may want the application to halt at the place in the code where you think the problem might have started. This is one reason Visual Basic provides breakpoints and Stop statements. A *breakpoint* defines a statement or set of conditions at which Visual Basic automatically stops execution and puts the application in break mode without running the statement containing the breakpoint. See "Using Stop Statements" later in this chapter for a comparison of Stop statements and breakpoints.

You can enter break mode manually if you do any of the following while the application is running:

* Press Ctrl + Break
* Choose Break from the Run menu.
* Click the Break button on the toolbar.

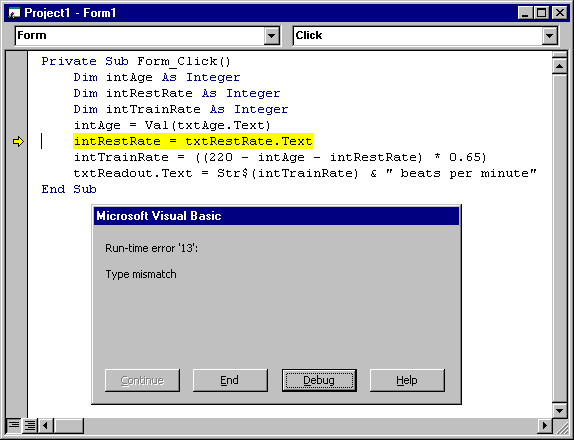
It's possible to break execution when the application is idle (when it is between processing of events). When this happens, execution does not stop at a specific line, but Visual Basic switches to break mode anyway.

You can also enter break mode automatically when any of the following occurs:

* A statement generates an untrapped run-time error.
* A statement generates a run-time error and the Break on All Errors error trapping option has been selected.
* A break expression defined in the Add Watch dialog box changes or becomes true, depending on how you defined it.
* Execution reaches a line with a breakpoint.
* Execution reaches a Stop statement.

Fixing a Run-Time Error and Continuing

Some run-time errors result from simple oversights when entering code; these errors are easily fixed. Frequent errors include misspelled names and mismatched properties or methods with objects — for example, trying to use the Clear method on a text box, or the Text property with a file list box. Below is a run-time error message.



Often you can enter a correction and continue program execution with the same line that halted the application, even though you've changed some of the code. Simply choose Continue from the Run menu or click the Continue button on the toolbar. As you continue running the application, you can verify that the problem is fixed.

If you select Break on All Errors from the Default Error Trapping State option group on the General tab on the Options dialog box (available from the Tools menu), Visual Basic disables error handlers in code, so that when a statement generates a run-time error, Visual Basic enters break mode. If Break on All Errors is not selected, and if an error handler exists, it will intercept code and take corrective action.

**Note:**   When you change the Default Error Trapping State option via the Options dialog box, this setting becomes the default for all subsequent sessions of VB. To change error handling for just the current session, select Toggle from the code window context menu to open a submenu that allows selection of the break mode.

Some changes (most commonly, changing variable declarations or adding new variables or procedures) require you to restart the application. When this happens, Visual Basic presents a message that asks if you want to restart the application.

Macro: A special form of Sub

Macro is special Subroutine without parameter and with non-private access modifier. It can be handcoded or recorded by using **Macro Recorder**.

Up to Excel 2007 you didn't need to install the Visual Basic Editor if you wanted to develop macros (VBA procedures). In Excel 2007 you must specify that VBA be installed when you install Excel from the Office CD. See how to install Visual Basic for Application for Excel 2007.

In Excel 2010 this problem has been corrected and you do not need to install anything extra to work with macros. YOu might have to make the "Developer" ribbon visible by right clicking on the "Home" item of the menu bar and then selecting "Customize the ribbon..." and making sure that the "Developer" ribbon is checked.

In Excel 2007 and 2010 you save your spreadsheets that do not contain macros in .xlsx format and those with macros in .xlsm format. You can also select to save your spreadsheets in .xls format so that people who have not migrated to Excel 2007 or 2010 can open and use them.

All macros and VBA procedures developed in Excel 1997 to 2003 will work in Excel 2007 and 2010 except for a few minor changes:

* a function used only by advanced users " FileSearch" does not exist anymore. It can be easily replaced by "Dir" that runs in Excel 2007, 2010 and in earlier versions of Excel.
* Advanced users of Excel 2007 and 2010 who import external data with or without SQL might want to add these two lines of code at the end of the refresh process:
  + ActiveSheet.ListObjects(" YourQueryName" ).Unlink
  + ActiveSheet.ListObjects(" YourQueryName" ).Unlist
* Use the macro recorder to discover the new syntax to cells' background, font colors, filtering and sorting data.

Nothing else has changed in Excel 2007 and 2010 as far as VBA and macros are concerned.

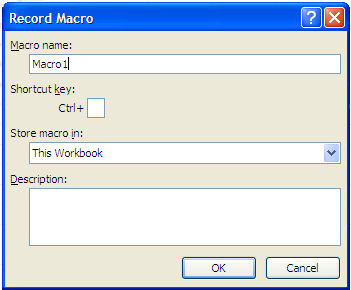
The macro recorder and the Visual Basic Editor are the same.

The Macro Recorder in Excel 2007 to 2010

One of the tools that make the programming environment in Excel unique is the **Excel Macro Recorder**. When you start the macro recorder anything you do in Excel is recorded as a new macro. That makes the macro recorder the best VBA teacher and also a great assistant who will write a lot of the words and sentences that you need without a single typo. It will also be there when you do not remember something that you do not use often. Even after many years of programming you will still use the macro recorder daily not to learn anymore but to write code (VBA words and sentences).

**Steps of recording newmMacro:**

1. Excel and a new workbook.
2. to the "Developer" ribbon to click on  http://www.excel-vba.com/zzz-record-macro.jpg
3. A small window appears titled "Record Macro". Click on "OK".

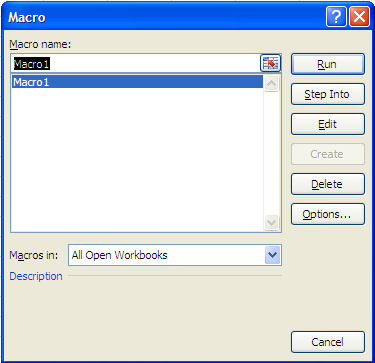


When you do so the small window disappears and in the "Developer" ribbon http://www.excel-vba.com/zzz-record-macro.jpg  is replaced by http://www.excel-vba.com/zzz-stop-recording.jpg telling you that you are going in the right direction. The macro recorder is ON.

1. In the sheet below (Sheet1) select cells B1 to B5, go to "Sheet2", select cell B6, come back to "Sheet1" and select cells D2 to D5.
2. the "Developer" ribbon click on http://www.excel-vba.com/zzz-stop-recording.jpg

**Steps of running recorded macro:**

1. Select cell "A1" of "Sheet1".
2. In the "Developer" ribbon click on http://www.excel-vba.com/zzz-excel-macros.jpg
3. In the window that appears Macro1 is selected.



1. Click "Run".
2. See how fast the macro runs. You do not even see Excel go to Sheet2 (but it does). At the end of the execution cells D2 to D5 are selected

What took you around 5 seconds to do manually only took Excel a fraction of a second. Excel can work much faster than you can. Welcome to the marvelous world of VBA for Excel (Macros).

Looking at your first recorded macro

To complete this third exercise you must have studied lessons 1 to 4.

Go to the VBE and you will see the following macro in the code window when you double click on Module in the Project Window:

Sub Macro1()

'

' Macro1 Macro

'

    Range("B1:B5").Select  
    Sheets("Sheet2").Select  
    Range("B6").Select  
    Sheets("Sheet1").Select  
    Range("D2:D5").Select

End Sub

As you can see the macro recorder recorded your instructions in a language that Excel understands (VBA). You can now use VBA's written code to have Excel perform this task.

Never forget that the Excel macro recorder is your best teacher and will remain a great assistant for the rest of your VBA developer's life.

Developing  Macros  in Excel

Most macros are developed in the code window of modules. For the purpose of this exercise double click on "Sheet1" in the project window

Enter sub proTest() without using  a capital "S" as the beginning of "sub". After entering the closing parenthesis click on "Enter". You get these two lines of code:

Sub proTest()

End Sub

VBE adds the line "End Sub" and capitalizes the "S" of "Sub" . The VBE capitalizes letters appropriately when the word is spelled correctly. This is one interesting feature that you should always use when writing macros. Make it tour habit never to use capital letters when writing code. In this way, whenever VBE unexpected fails to capitalize a letter, you will know that something is wrong.

You may now write a procedure within the two lines of code above. For example your VBA procedure could look like this. You can copy/paste the macro below from your browser to the VBE Code window, or key it in. Make sure that everything is there including all the quotation marks and periods, parentheses, equal signs, and spaces.

**Note:** Make sure that you copy/paste this code in a NEW workbook not one created in a previous exercise.

Sub proTest()

        Sheets("Sheet1").Select  
        Range("C1").Select

        Do Until Selection.Offset(0, -2).Value = ""   
                Selection.Value = Selection.Offset(0, -2).Value & " " & Selection.Offset(0, -1)  
                Selection.Offset(1, 0).Select  
        Loop

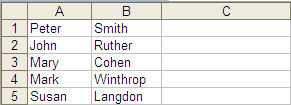
        Range("A1").Select

End Sub

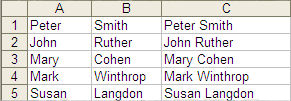
The procedure above will go down column "C" and assemble the first names of column "A" and the last names of column "B" with a space in between. It will perform this task all the way down until there are no more first names in column "A". It will then place the cursor in cell "A1".

To test this macro, follow the steps below:

1. Enter first names in cell A1 to A5.
2. Enter surnames in cells B1 to B5.



1. Come back to the VBE (Altl + F11) and click within the macro in the code window.
2. From the menu bar select "**Run**/**Run Sub**/**Userform**".
3. Go back to Excel and see the result.



You can erase everything in column C Excel and retry with more names and surnames.

Try it again removing the first name in cell A3. Notice that the macro stops on line 2.

Testing  Macros in the Visual Basic Editor for Excel

**NOTE:** While you are running the macro step by step you can stop the execution at any time by clicking on the stop button in the toolbar.   VBA for Excel reset button

Testing is the most time-consuming part of any VBA project. During the development of a project you will use 20% of your time analysing and designing, 15% programming and 65% testing.

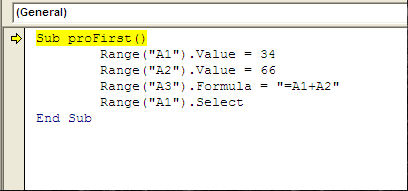
During the testing phase, you will correct bugs, typos and the logical errors. More importantly you will improve your original project, fine tune it, discover better ways to do things and add code.

Try the following:

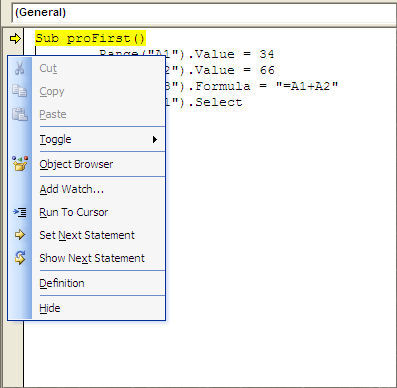
1. Go to Excel and make sure that cells A1, A2 and A3 of Sheet1 are empty.
2. In VBE go to the Code window of Sheet1 and copy/paste the following macro:

Sub proFirst()  
        Range("A1").Value = 34  
        Range("A2").Value = 66  
        Range("A3").Formula = "=A1+A2"  
        Range("A1").Select  
End Sub

1. Click anywhere within the macro and then press the F8 key at the top of your keyboard. VBE highlights the first line of code in yellow.



1. Right-click on the small yellow arrow and see a menu appear



1. Press on F8 a second time. No line has been executed yet and if you go to Excel you will see that cells A1 to A3 are still empty. The next time you press F8, VBE will execute the yellow-highlighted line.
2. Press F8 a third time. The yellow-highlighted line is now "Range("A2").Value = 66". VBE has executed the previous line "Range("A1").Value = 34" has been executed so if you go to Excel (Alt + F11) you will see 32 in cell A1.
3. Come back to VBE (Alt + F11) and press F8 again. Go to Excel and see what happened in cell A2.
4. Come back to VBE (Alt + F11) and press F8 again. Go to Excel and see that there is a formula in cell A3.
5. Come back to the VBE (Alt + F11) and press F8 again, cell A1 is now selected in Excel.
6. Press F8 again. Nothing happens in Excel but "End Sub" is highlighted in yellow
7. Press F8 again. Nothing happens in Excel no more lines in VBE are highlighted in yellow.

The macro hac been tested, the test is over.

In the code change the addresses A1, A2 and A3 respectively to B1, B2 and B3.

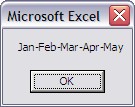
Event Handler: A special form of Sub

Event programming is a very powerful tool that you can use within your VBA code to monitor user actions, take appropriate action when a user does something, or monitor the state of the application as it changes. If you are programming with your own custom classes, you can extend the functionality of these classes by defining and raising your own custom events, broadcasting the event message to any object that is listening for events from your class.

Events and Event Procedures were introduced in Excel97. Earlier versions of Excel do not support events. Events and event procedures are the same for versions 97 through 2007. No significant changes have been made to the event model since its introduction in Excel97. A few new events have been added, but the overall structure of the event system is unchanged.

**NOTE:** We will study Event Handler extensively under the Module Document Object Model.

Module 4 – Advanced Array Handling

An array is a set of sequentially indexed elements having the same intrinsic data type. Each element of an array has a unique identifying index number. Changes made to one element of an array don't affect the other elements.    
  
Before signing values to an array, the array needs to be created.  You can declare the array by using the Dim statement.  
  
For example, to declare a one-dimensional array with 5 elements, type the following:  
  
            Dim Arr(4)  
  
The element’s index of the array starts from 0 unless Option Base 1 is specified in the public area (area outside of the sub procedure).  If Option Base 1 is specified, the index will start from 1.  
  
The following example assigns values to the array and displays all values in a message box :  
  
         Option Base 1  
         Sub assignArray( )  
                  Dim Arr(5)   
                  Arr(1) = “Jan”  
                  Arr(2) = “Feb”  
                  Arr(3) = “Mar”  
                  Arr(4) = “Apr”  
                  Arr(5) = “May”  
                  Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5)  
         End Sub  
  
  
          
  
**NOTE:** The number inside the array, i.e. Arr(1), is the index.  One (1) is the index of the first element in the  array.

LBound and UBound Functions

The largest available subscript for the indicated dimension of an array can be obtained by using the Ubound function.  In our one-dimensional array example, Ubound(arr) is 5.  
   
In our two-dimensional array example above, there are two upper bound figures - both are 2.    
UBound returns the following values for an array with these dimensions\*:  
  
         Dim A(1 To 100, 0 To 3, -3 To 4)  
  
         **Statement             Return Value**   
         UBound(A, 1)                         100                     
         UBound(A, 2)                            3   
         UBound(A, 3)                            4   
  
The UBound function is used with the LBound function to determine the size of an array. Use the LBound function to find the lower limit of an array dimension.  
  
         **Statement             Return Value**   
         LBound(A, 1)                            1                     
         LBound(A, 2)                            0   
         LBound(A, 3)                           -3   
  
To get the size of an array, use the following formula:  
  
        UBound(Arr) - LBound(Arr) + 1  
  
For example:   
  
        Ubound(A,1) - LBound(A,1) + 1   
        = 100 - 1 + 1   
        = 100  
  
        Ubound(A,2) - LBound(A,2) + 1   
        = 3 - 0 + 1   
        = 4  
  
        Ubound(A,3) - LBound(A,3) + 1   
        = 4 - (-3) + 1   
        = 8

Multi-Dimensional Array

An array can also store multiple dimensional data.  To simplify our tutorial, example on a two-dimensional array is used.  Assume you have data of a local store's yearly sale in the following table and you want to store the data in a two-dimensional array:

|  |  |  |
| --- | --- | --- |
| **Year**  **Sales** | **2003** | **2004** |
| CD | 1000 | 1500 |
| DVD | 1200 | 2000 |

First we create the array as follow:

                Dim Arr(2,2)

Then we assign the values into the array.  We treat the first dimension as the year and the second dimension as the product sale:  
  
            arr(1,1) = 1000  
            arr(1,2) = 1200  
            arr(2,1) = 1500  
            arr(2,2) = 2000  
  
We now display the values of the array with a message box:  
  
          Msgbox "Sale of CD in 2003 is " & arr(1,1) & vbCrLf  & "Sale of CD in 2004 is " \_  
                                & arr(2,1)  & vbCrLf  & "Sale of DVD in 2003 is " & arr(1,2) & vbCrLf  \_  
                        & "Sale of DVD in 2004 is " & arr(2,2)   
  
The complete precedure is as followed:  
              
         Option Base 1  
         Sub multDimArray( )              
                  Dim Arr(2,2)  
                  arr(1,1) = 1000  
                  arr(1,2) = 1200  
                  arr(2,1) = 1500  
                  arr(2,2) = 2000   
                  Msgbox "Sale of CD in 2003 is " & arr(1,1) & vbCrLf  & "Sale of CD in 2004 is " \_  
                                   & arr(2,1) & vbCrLf  & "Sale of DVD in 2003 is " & arr(1,2) & vbCrLf  \_  
                                   & "Sale of DVD in 2004 is " & arr(2,2)   
         End Sub  
  
  
              
  
**NOTE:**vbCrLf stands for VB Carriage Return Line Feed.  It puts a return and a new line as shown in the   message box above.  The underscore "\_" on the back of the first line of the message box means   "continue to the next line"

Redim

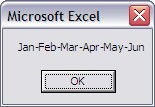
The ReDim statement is used to size or resize a dynamic array that has already been formally declared.  
  
For example, if you have already declared an array with an element value of 5 and decided to change the number of the element to 6, you can do the following to resize the array:  
  
            Redim Arr(6)  
  
We incorporate it into our last example:  
  
         Option Base 1  
         Sub assignArray( )  
                  'Dim Arr(5)   
                  Redim Arr(6)  
                  Arr(1) = “Jan”  
                  Arr(2) = “Feb”  
                  Arr(3) = “Mar”  
                  Arr(4) = “Apr”  
                  Arr(5) = “May”  
                  Arr(6) = “Jun”  
                  Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5)  
            End Sub   
  
Note that the Dim Arr(5) statement is commoned out, because leaving this original statement in the sub will causing a compile error.

Redim Preserve

A word of caution in using the Redim Statement to resize an array - resize the array can erase the elements in it.  In the following example, all the values assigned prior to resize the array are erased.  Only the value assigned to the array after resize remains.   
  
  Option Base 1  
  Sub assignArray( )                    
        Redim Arr(5)  
  
       Arr(1) = “Jan”  
       Arr(2) = “Feb”  
       Arr(3) = “Mar”  
       Arr(4) = “Apr”  
       Arr(5) = “May”  
  
       Redim Arr(6)  
  
 Arr(6) = “Jun”  
  
       Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5) & "-" & Arr(6)  
   End Sub

****

By replace the Redim Arr(6) with Redim Preserve Arr(6), all values will remain.  For example:  
  
   Option Base 1  
   Sub assignArray( )  
    Redim Arr(5)  
 Arr(1) = “Jan”  
        Arr(2) = “Feb”  
        Arr(3) = “Mar”  
        Arr(4) = “Apr”  
        Arr(5) = “May”  
 Redim Preserve Arr(6)  
 Arr(6) = “Jun”  
 Msgbox Arr(1) & "-" & Arr(2) & "-" & Arr(3) & "-" & Arr(4) & "-" & Arr(5) & "-" & Arr(6)  
   End Sub

****

Module 5 – Advanced Types

The following table summarizes variable/constant data types.

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Types** | | | |
| **Type** | **Memory** | **Type-Declaration Character** | **Description** |
| Byte | 1 byte | none | Positive whole number ranging from 0 through 255 that can be represented as a binary value. |
| Boolean | 2 bytes | none | **True** or **False** value |
| Integer | 2 bytes | % | Whole numbers ranging from -32,768 through 32,767. |
| Long (*long integer* | 4 bytes | & | Whole numbers ranging from -2,147,483,648 through 2,147,483,647. |
| Single | 4 bytes | ! | Single-precision floating-point number (with decimal points) ranging from -3.402823E38 to 3.402823E38. |
| Double | 8 bytes | # | Double-precision floating-point number (which is more precise for very large or very small numbers) ranging from -1.79769313486232E308 to 1.79769313486232E308. |
| Currency | 8 bytes | @ | Large numbers between -922,337,203,685,477.5808 and 922,337,203,685,477.5807 (15 digits to left of decimal and 4 digits to the right of the decimal). |
| Date | 8 bytes | none | Represents dates from January 1, 100 through December 31, 9999. |
| Object | 4 bytes | none | An instance of a class or object reference. |
| String | 10 bytes + 1 byte per char | $ | Series of any ASCII characters. |
| String (*fixed-length*) | length of string | none | Series of any ASCII characters, of a pre-defined length. |
| Variant | min 16 bytes | none | Any kind of data except fixed-length String data and user-defined types. |
| Sub VariableDemo()  Dim Prompt$, varUserInput As Variant  Initialize a String variable with an instruction to appear in an InputBox  Prompt$ = "Please enter something."  ' Apply your String variable as an argument for the InputBox function.  ' Use a Variant variable to capture the user's entry in an InputBox  VarUserInput = InputBox(Prompt$)  ' Display the user's entry by applying your variable within a  ' message dialog  MsgBox varUserInput End Sub  A couple of things to notice about this example are how a type-declaration character (that was the $ sign in the Prompt$ variable) can be applied to define the String variable. Also, you can see how a Variant was purposely used since the user could enter either numbers or text in the InputBox. | | | |

Variant

The VBA [language](http://www.functionx.com/vbaexcel/keywords/variant.htm) provides a universal (or vague) data type you can use for any type of value. The **Variant** data type is used to declare a variable whose type is not explicitly specified. This means that a Variant data type can hold any type of value you want.

Here are [examples](http://www.functionx.com/vbaexcel/keywords/variant.htm) of Variant-declared variables that hold different types of values:

Sub Exercise()

Dim FullName As Variant

Dim EmploymentStatus As Variant

Dim HourlySalary As Variant

Dim DateHired As Variant

FullName = "Patricia Katts"

EmploymentStatus = 2

HourlySalary = 35.65

DateHired = #6/22/2004#

End Sub

Using Type

User Defined Types (UDTs) are a convenient way to store related data in one variable. They are essentially data types that you (as the programmer) setup. You could, of course, store unrelated data in UDTs, but that’s not generally how they’re used.

I find them most useful when I have to pass information between procedures. Rather than have, say, four parameters in a procedure, I prefer to use a UDT with four elements. That way, I can pass one variable. For me, it makes the code more readable and self-documenting. And you get the benefit of Intellisense when using UDTs, which is nice.

Here’s a simple, contrived example of how to use a UDT. The Type statement is used to define the UDT and must be outside of any procedures.

Option Explicit

Private Type Applicant  
    FirstName As String  
    LastName As String  
    ResumeRecd As Boolean  
    InterviewDate As Date  
End Type  
      
Sub DefineUDT()

Dim uApp As Applicant  
      
    uApp.FirstName = “Dick”  
    uApp.LastName = “Kusleika”  
    uApp.ResumeRecd = True  
    uApp.InterviewDate = #7/15/2004#  
      
    UseUDT uApp  
End Sub

Sub UseUDT(uInput As Applicant)

    If uInput.InterviewDate = Date Then  
        Debug.Print uInput.FirstName, uInput.LastName  
        Debug.Print uInput.ResumeRecd  
    End If  
End Sub

An Example of Using a Type Structure

 Type CompanyInfo  
    SetUpID As Long  
    CompanyName As String \* 50  
    Address As String \* 255  
    City As String \* 50  
    StateProvince As String \* 20  
    PostalCode As String \* 20  
    Country As String \* 50  
    PhoneNumber As String \* 30  
    FaxNumber As String \* 30  
    DefaultPaymentTerms As String \* 255  
    DefaultInvoiceDescription As String  
End Type  
  
Public typCompanyInfo As CompanyInfo  
  
Sub GetCompanyInfo()  
    Dim strSubName As String  
    Dim rst As ADODB.Recordset  
  
    Set rst = New ADODB.Recordset  
    rst.ActiveConnection = CurrentProject.Connection  
    rst.Open "Select \* from CompanyInfo", Options:=adCmdText  
  
    With typCompanyInfo  
        .SetUpID = rst!SetUpID  
        .CompanyName = rst!CompanyName  
        .Address = rst!Address  
        .City = rst!City  
        .StateProvince = rst!StateOrProvince  
        .PostalCode = rst!PostalCode  
        .Country = rst!Country  
        .PhoneNumber = rst!PhoneNumber  
        .FaxNumber = rst!PhoneNumber  
    End With  
  
    rst.Close  
    Set rst = Nothing  
End Sub

Passing User-Defined Types

Many DLL functions require that you pass in a data structure by using a predefined format. When calling a DLL function from VBA, you pass a user-defined type that you've defined according to the function's requirements.

You can figure out when you need to pass a user-defined type and which type definition you need to include in your code by looking at the **Declare** statement for the function. An argument requiring a data structure is always declared as a *long pointer*: a 32-bit numeric value that points to the data structure in memory. The conventional prefix for a long pointer argument is "lp". In addition, the data type for the argument will be the name of the data structure.

For example, take a look at the **Declare** statements for the **GetLocalTime** and **SetLocalTime** functions:

Private Declare Sub GetLocalTime Lib "kernel32" (lpSystem As SYSTEMTIME)

Private Declare Function SetLocalTime Lib "kernel32" (lpSystem As SYSTEMTIME) As Long

Both functions take an argument of type SYSTEMTIME, a data structure that contains date and time information. Here's the definition for the SYSTEMTIME type:

Private Type SYSTEMTIME

      wYear  As Integer

      wMonth As Integer

      wDayOfWeek As Integer

      wDay As Integer

      wHour As Integer

      wMinute  As Integer

      wSecond As Integer

      wMilliseconds As Integer

End Type

To pass the data structure to a function, you must declare a variable of type SYSTEMTIME, as in the following example:

Private sysLocalTime As SYSTEMTIME

When calling **GetLocalTime**, you pass a variable of type SYSTEMTIME to the function, and it fills the data structure with numeric values indicating the current local year, month, day, day of week, hour, minute, second, and millisecond. For example, the following **Property** **Get** procedure calls **GetLocalTime** to return a value indicating the current hour:

Public Property Get Hour() As Integer

   ' Retrieve current time, then return hour.

   GetLocalTime sysLocalTime

   Hour = sysLocalTime.wHour

End Property

When calling **SetLocalTime**, you also pass a variable of type SYSTEMTIME, but you first provide values for one or more of the elements of the data structure. For example, the following **Property** **Let** procedure sets the hour value for the local system time. First, it calls **GetLocalTime** to retrieve the most current values for the local time into the data structure, **sysSystem.** Then it updates the value of the **sysLocalTime.wHour** element of the data structure with the value of the argument passed to the property procedure. Finally, it calls **SetLocalTime**, passing in the same data structure, which contains the values retrieved by **GetLocalTime** plus the new hour value.

Public Property Let Hour(intHour As Integer)

   ' Retrieve current time so that all values will be current,

   ' then set hour portion of local time.

   GetLocalTime sysLocalTime

   sysLocalTime.wHour = intHour

   SetLocalTime sysLocalTime

End Property

The **GetLocalTime** and **SetLocalTime** functions are similar to the **GetSystemTime** and **SetSystemTime** functions. The primary difference is that the **GetSystemTime** and **SetSystemTime** functions express time as Greenwich meantime. For example, if your local time is 12:00 midnight, and you live on the West Coast, Greenwich means time is 8:00 A.M., an eight-hour difference. The **GetSystemTime** function returns the current time as 8:00 A.M., while **GetLocalTime** returns 12:00 midnight.

Object

Excel has an object model which is a big hierarchy of all the objects that you can use in VBA. At the top of this hierarchy is the Application object, and all the other objects are below it.

The dot operator (.) is used to navigate through the hierarchy. You connect objects with a dot to get to lesser objects. If you want to change the font in a cell, you need to get to a Font object. For example start with the Application object, then navigates to the workbook, worksheet, range and font:

Application.ActiveWorkbook.ActiveSheet.Range(”A1″).Font

That’s a long way to go to get to a Font object. There are some shortcuts that I discuss in a moment, but whatever syntax you use, this is what’s happening under the hood.

Default Objects

Whenever you’re dealing with Excel’s object model, there are always default objects. For instance, the Application object is always assumed, regardless of where your code is. The path to the Font object could be written

ActiveWorkbook.ActiveSheet.Range(”A1″).Font

When you omit the Application object, Excel assumes that it’s there.

Standard Modules

When you’re in a standard module (like Module1), there are some more default objects at your disposal. When you omit the workbook reference, Excel assumes you want the ActiveWorkbook. This code is the same as the previous two snippets

ActiveSheet.Range(”A1″).Font

How much would you pay for these default objects. Wait, don’t answer yet, there’s more. You can omit the worksheet reference and the ActiveSheet will be assumed. Yet another equivalent code snippet

Range(”A1″).Font

That sure is shorter than the line we started with. Don’t get too excited, there’s a downside.

Class Modules

Class modules include ThisWorkbook, Sheet1 (and other sheet modules), Userforms, and class modules that you create. ThisWorkbook and the sheet modules are located in the Microsoft Excel Objects folder of the VBE’s Project Explorer. These modules are used to program event procedures. What you need to know is that the default objects rules that apply to standard modules don’t apply in these modules.

When you omit the workbook reference in the ThisWorkbook module, the workbook that contains the code is assumed, not the ActiveWorkbook. Similarly, omitted sheet references in a sheet module are assumed to be the sheet whose module the code is in, not the ActiveSheet.

The Danger of Default Object

You can pretty much rely on default objects as described. That doesn’t mean that you should. There are few exceptions. The main exception is the Application object. When you use references that navigate down the hierarchy, they’re called fully qualified references.

There are two problems with references that aren’t fully qualified. The first is programmer error. Although you can rely on the ActiveWorkbook being the default workbook reference in a standard module, you can’t rely that you know which workbook is active. Even if you get it right, changing code later, you can screw it up royally. The second problem is execution speed. For any program that works with more than one sheet, you would have to activate different sheets in order for them to be the default sheet reference.

Shortcuts to Fully Qualified References

Excel provides all these default object references for your convenience, but it is advisable not to use them. Now your code is going to be ten times longer. Anyway there are a few shortcuts you can use to keep your code managable.

The **With** keyword (commonly called a With Block) is used to access properties and methods of an object without having to type the object over and over. Take this snippet

ThisWorkbook.Worksheets(1).Range("A1").Value = 10  
ThisWorkbook.Worksheets(1).Range("A1:B1").Font.Bold = True  
ThisWorkbook.Worksheets(1).Range("A1").Interior.ColorIndex = 3

That’s a lot of typing. You can shorten it up and make it more readable with a With Block.

With ThisWorkbook.Worksheets(1)  
     .Range("A1").Value = 10  
     .Range("A1:B1").Font.Bold = True  
     .Range("A1").Interior.ColorIndex = 3  
End With

You can also nest With Blocks

With ThisWorkbook.Worksheets(1)  
     With .Range("A1")  
          .Value = 10  
          .Interior.ColorIndex = 3  
     End With  
     .Range("A1:B1").Font.Bold = True  
End With

Finally, you can get creative inside a with block

With ThisWorkbook.Worksheets(1).Range("A1")  
     .Value = 10  
     .Interior.ColorIndex = 3  
     .Resize(1,2).Font.Bold = True  
End With

The Resize method is used to get to a different range based on A1.

Another way to have fully qualified references without all the mess is object variables. You use the Set keyword to assign an object to a variable and use that variable like you would a long object reference.

Dim rMyRange As Range  
   
Set rMyRange = ThisWorkbook.Worksheets(1).Range("A1")  
   
rMyRange.Value = 10  
rMyRange.Interior.ColorIndex = 3  
rMyRange.Resize(1,2).Font.Bold = True

Predefined Constants

VB has many pre-defined constants. These have been translated into a module **vbconstants.py** which is automatically imported into the current namespace by the **vbfunctions** module. This means that the VB constants will be resolved in any module which does a from **vb2py.vbfunctions import \*.** It does also mean that you get a fair amount of namespace polution. On the positive side, the vb constants begin with the vb prefix so clashes are not very likely.

**VB Constants**

**# Key Codes**

vbKeyLButton = 0x1 # Left mouse button

vbKeyRButton = 0x2 # Right mouse button

vbKeyCancel = 0x3 # CANCEL key

vbKeyMButton = 0x4 # Middle mouse button

vbKeyBack = 0x8 # BACKSPACE key

vbKeyTab = 0x9 # TAB key

vbKeyClear = 0xC # CLEAR key

vbKeyReturn = 0xD # ENTER key

vbKeyShift = 0x10 # SHIFT key

vbKeyControl = 0x11 # CTRL key

vbKeyMenu = 0x12 # MENU key

vbKeyPause = 0x13 # PAUSE key

vbKeyCapital = 0x14 # CAPS LOCK key

vbKeyEscape = 0x1B # ESC key

vbKeySpace = 0x20 # SPACEBAR key

vbKeyPageUp = 0x21 # PAGE UP key

vbKeyPageDown = 0x22 # PAGE DOWN key

vbKeyEnd = 0x23 # END key

vbKeyHome = 0x24 # HOME key

vbKeyLeft = 0x25 # LEFT ARROW key

vbKeyUp = 0x26 # UP ARROW key

vbKeyRight = 0x27 # RIGHT ARROW key

vbKeyDown = 0x28 # DOWN ARROW key

vbKeySelect = 0x29 # SELECT key

vbKeyPrint = 0x2A # PRINT SCREEN key

vbKeyExecute = 0x2B # EXECUTE key

vbKeySnapshot = 0x2C # SNAPSHOT key

vbKeyInsert = 0x2D # INSERT key

vbKeyDelete = 0x2E # DELETE key

vbKeyHelp = 0x2F # HELP key

vbKeyNumlock = 0x90 # NUM LOCK key

**# Shell Codes**

vbHide = 0 # Window is hidden and focus is passed to the hidden window.

vbNormalFocus = 1 # Window has focus & is restored to its original size & position.

vbMinimizedFocus = 2 # Window is displayed as an icon with focus.

vbMaximizedFocus = 3 # Window is maximized with focus.

vbNormalNoFocus = 4 # Window is restored to its most recent size and position. The currently active window remains active.

vbMinimizedNoFocus = 6 # Window is displayed as an icon. The currently active window remains active.

**# Colour Codes**

vbBlack = 0x0 # Black

vbRed = 0xFF # Red

vbGreen = 0xFF00 # Green

vbYellow = 0xFFFF # Yellow

vbBlue = 0xFF0000 # Blue

vbMagenta = 0xFF00FF # Magenta

vbCyan = 0xFFFF00 # Cyan

vbWhite = 0xFFFFFF # White

**# System Colour Codes**

vbScrollBars = 0x80000000 # Scroll bar color

vbDesktop = 0x80000001 # Desktop color

vbActiveTitleBar = 0x80000002 # Color of the title bar for the active window

vbInactiveTitleBar = 0x80000003 # Color of the title bar for the inactive window

vbMenuBar = 0x80000004 # Menu background color

vbWindowBackground = 0x80000005 # Window background color

vbWindowFrame = 0x80000006 # Window frame color

vbMenuText = 0x80000007 # Color of text on menus

vbWindowText = 0x80000008 # Color of text in windows

vbTitleBarText = 0x80000009 # Color of text in caption, size box, and scroll arrow

vbActiveBorder = 0x8000000A # Border color of active window

vbInactiveBorder = 0x8000000B # Border color of inactive window

vbApplicationWorkspace = 0x8000000C # Background color of multiple-document interface (MDI) applications

vbHighlight = 0x8000000D # Background color of items selected in a control

vbHighlightText = 0x8000000E # Text color of items selected in a control

vbButtonFace = 0x8000000F # Color of shading on the face of command buttons

vbButtonShadow = 0x80000010 # Color of shading on the edge of command buttons

vbGrayText = 0x80000011 # Grayed (disabled) text

vbButtonText = 0x80000012 # Text color on push buttons

vbInactiveCaptionText = 0x80000013 # Color of text in an inactive caption

vb3DHighlight = 0x80000014 # Highlight color for 3-D display elements

vb3DDKShadow = 0x80000015 # Darkest shadow color for 3-D display elements

vb3DLight = 0x80000016 # Second lightest 3-D color after vb3DHighlight

vbInfoText = 0x80000017 # Color of text in ToolTips

vbInfoBackground = 0x80000018 # Background color of ToolTips

**# Var Type Codes**

vbEmpty = 0 # Uninitialized (default)

vbNull = 1 # Contains no valid data

vbInteger = 2 # Integer

vbLong = 3 # Long integer

vbSingle = 4 # Single-precision floating-point number

vbDouble = 5 # Double-precision floating-point number

vbCurrency = 6 # Currency

vbDate = 7 # Date

vbString = 8 # String

vbObject = 9 # Object

vbError = 10 # Error

vbBoolean = 11 # Boolean

vbVariant = 12 # Variant (used only for arrays of variants)

vbDataObject = 13 # Data access object

vbDecimal = 14 # Decimal

vbByte = 17 # Byte

vbUserDefinedType = 36 # Variants that contain user-defined types

vbArray = 8192 # Array

**# MsgBox Codes**

vbOKOnly = 0 # OK button only (default)

vbOKCancel = 1 # OK and Cancel buttons

vbAbortRetryIgnore = 2 # Abort, Retry, and Ignore buttons

vbYesNoCancel = 3 # Yes, No, and Cancel buttons

vbYesNo = 4 # Yes and No buttons

vbRetryCancel = 5 # Retry and Cancel buttons

vbCritical = 16 # Critical message

vbQuestion = 32 # Warning query

vbExclamation = 48 # Warning message

vbInformation = 64 # Information message

vbDefaultButton1 = 0 # First button is default (default)

vbDefaultButton2 = 256 # Second button is default

vbDefaultButton3 = 512 # Third button is default

vbDefaultButton4 = 768 # Fourth button is default

vbApplicationModal = 0 # Application modal message box (default)

vbSystemModal = 4096 # System modal message box

vbMsgBoxHelpButton = 16384 # Adds Help button to the message box

VbMsgBoxSetForeground = 65536 # Specifies the message box window as the foreground window

vbMsgBoxRight = 524288 # Text is right aligned

vbMsgBoxRtlReading = 1048576 # Specifies text should appear as right-to-left reading on Hebrew and Arabic systems

vbOK = 1 # OK button pressed

vbCancel = 2 # Cancel button pressed

vbAbort = 3 # Abort button pressed

vbRetry = 4 # Retry button pressed

vbIgnore = 5 # Ignore button pressed

vbYes = 6 # Yes button pressed

vbNo = 7 # No button pressed

**# Form Codes**

vbModeless = 0 # UserForm is modeless.

vbModal = 1 # UserForm is modal (default).

**# Special Folder Codes**

WindowsFolder = 0 # The Windows folder contains files installed by the Windows operating system.

SystemFolder = 1 # The System folder contains libraries, fonts, and device drivers.

TemporaryFolder = 2 # The Temp folder is used to store temporary files. Its path is found in the TMP environment variable.

**# Dir etc Codes**

vbNormal = 0 # Normal (default for Dir and SetAttr)

vbReadOnly = 1 # Read-only

vbHidden = 2 # Hidden

vbSystem = 4 # System file

vbVolume = 8 # Volume label

vbDirectory = 16 # Directory or folder

vbArchive = 32 # File has changed since last backup

**# File Attribute Codes**

Normal = 0 # Normal file. No attributes are set.

ReadOnly = 1 # Read-only file. Attribute is read/write.

Hidden = 2 # Hidden file. Attribute is read/write.

System = 4 # System file. Attribute is read/write.

Volume = 8 # Disk drive volume label. Attribute is read-only.

Directory = 16 # Folder or directory. Attribute is read-only.

Archive = 32 # File has changed since last backup. Attribute is read/write.

Alias = 64 # Link or shortcut. Attribute is read-only.

Compressed = 128 # Compressed file. Attribute is read-only.

**# StrConv Codes**

vbUpperCase = 1 # Converts the string to uppercase characters.

vbLowerCase = 2 # Converts the string to lowercase characters.

vbProperCase = 3 # Converts the first letter of every word in string to uppercase.

vbWide = 4 # Converts narrow (single-byte) characters in string to wide (double-byte) characters. Applies to Far East locales.

vbNarrow = 8 # Converts wide (double-byte) characters in string to narrow (single-byte) characters. Applies to Far East locales.

vbKatakana = 16 # Converts Hiragana characters in string to Katakana characters. Applies to Japan only.

vbHiragana = 32 # Converts Katakana characters in string to Hiragana characters. Applies to Japan only.

vbUnicode = 64 # Converts the string to Unicode using the default code page of the system.

vbFromUnicode = 128 # Converts the string from Unicode to the default code page of the system.

**# Miscellaneous Codes**

vbCrLf = "\n" # Carriage returnlinefeed combination

vbCr = chr(13) # Carriage return character

vbLf = chr(10) # Linefeed character

vbNewLine = "\n" # Platform-specific new line character; whichever is appropriate for current platform

vbNullChar = chr(0) # Character having value 0

vbNullString = chr(0) # String having value 0 Not the same as a zero-length string (""); used for calling external procedures

vbObjectError = -2147221504 # User-defined error numbers should be greater than this value. For example: Err.Raise Number = vbObjectError + 1000

vbTab = chr(9) # Tab character

vbBack = chr(8) # Backspace character

vbFormFeed = chr(12) # Not useful in Microsoft Windows

vbVerticalTab = chr(11) # Not useful in Microsoft Windows

Module 6 – Options

Option Base

Used at [module level](http://msdn.microsoft.com/en-us/library/aa171682.aspx) to declare the default lower bound for [array](http://msdn.microsoft.com/en-us/library/aa219965.aspx) subscripts. Using option base we can declare the array index starts from 1 2 or whatever.

Because the default base is **0**, the **Option Base** statement is never required. If used, the [statement](http://msdn.microsoft.com/en-us/library/aa212247.aspx) must appear in a [module](http://msdn.microsoft.com/en-us/library/aa171680.aspx) before any [procedures](http://msdn.microsoft.com/en-us/library/aa172172.aspx). **Option** **Base** can appear only once in a module and must precede array declarations that include dimensions.

**Note:**   The **To** clause in the **Dim**, **Private**, **Public**, **ReDim**, and **Static** statements provides a more flexible way to control the range of an array's subscripts. However, if you don't explicitly set the lower bound with a **To** clause, you can use **Option Base** to change the default lower bound to 1.  The base of an array created with the the **ParamArray** keyword is zero; **Option Base** does not affect **ParamArray** (or the **Array** function, when qualified with the name of its type library, for example **VBA.Array**).

The **Option Base** statement only affects the lower bound of arrays in the module where the statement is located.

Option Explicit

Used at [module level](http://msdn.microsoft.com/en-us/library/aa171682.aspx) to force explicit declaration of all [variables](http://msdn.microsoft.com/en-us/library/aa220732.aspx) in that [module](http://msdn.microsoft.com/en-us/library/aa171680.aspx).

If used, the **Option** **Explicit** statement must appear in a module before any [procedures](http://msdn.microsoft.com/en-us/library/aa172172.aspx).

When **Option Explicit** appears in a module, you must explicitly declare all variables using the **Dim**, **Private**, **Public**, **ReDim**, or **Static** statements. If you attempt to use an undeclared variable name, an error occurs at [compile time](http://msdn.microsoft.com/en-us/library/aa210361.aspx).

If you don't use the **Option Explicit** statement, all undeclared variables are of **Variant** type unless the default type is otherwise specified with a **Def***type* statement.

**Note:**   Use **Option Explicit** to avoid incorrectly typing the name of an existing variable or to avoid confusion in code where the scope of the variable is not clear.

Option Compare

Used at [**module level**](http://msdn.microsoft.com/en-us/library/aa171682.aspx)to declare the default comparison method to use when string data is compared.

If used, the **Option** **Compare** statement must appear in a [**module**](http://msdn.microsoft.com/en-us/library/aa171680.aspx) before any [**procedures**](http://msdn.microsoft.com/en-us/library/aa172172.aspx)**.**

The **Option Compare** statement specifies the [**string comparison**](http://msdn.microsoft.com/en-us/library/aa212267.aspx) method (**Binary**, **Text**, or **Database**) for a module. If a module doesn't include an **Option** **Compare** statement, the default text comparison method is **Binary**.

**Option Compare Binary** results in string comparisons based on a [**sort order**](http://msdn.microsoft.com/en-us/library/aa212196.aspx) derived from the internal binary representations of the characters. In Microsoft Windows, sort order is determined by the code page. A typical binary sort order is shown in the following example:

A < B < E < Z < a < b < e < z < < < < < <

**Option Compare Text** results in string comparisons based on a case-insensitive text sort order determined by your system's locale. When the same characters are sorted using **Option Compare Text**, the following text sort order is produced:

(A=a) < ( =) < (B=b) < (E=e) < (=) < (Z=z) < (=)

**Option** **Compare** **Database** can only be used within Microsoft Access. This results in string comparisons based on the sort order determined by the locale ID of the database where the string comparisons occur.

Stop text case sensitivity:

Option compare Text ' "A" is equal to "a".

Sub MyMacro

'Your code here.

End Sub

Make text case sensitive (Excels default)

Option compare Binary ' "A" is NOT equal to "a".

Sub MyMacro

'Your code here.

End Sub

Option Private

When used in host applications that allow references across multiple [projects](http://msdn.microsoft.com/en-us/library/aa172189.aspx), **Option Private Module** prevents a [module](http://msdn.microsoft.com/en-us/library/aa171680.aspx) contents from being referenced outside its project. In host applications that dont permit such references, for example, standalone versions of Visual Basic, **Option Private** has no effect.

If used, the **Option** **Private** statement must appear at [module level](http://msdn.microsoft.com/en-us/library/aa171682.aspx), before any [procedures](http://msdn.microsoft.com/en-us/library/aa172172.aspx).

When a module contains **Option Private Module**, the public parts, for example, [variables](http://msdn.microsoft.com/en-us/library/aa220732.aspx), [objects](http://msdn.microsoft.com/en-us/library/aa171792.aspx), and user-defined types declared at module level, are still available within the [project](http://msdn.microsoft.com/en-us/library/aa172189.aspx) containing the module, but they are not available to other applications or projects.

**Note:**   **Option Private** is only useful for host applications that support simultaneous loading of multiple projects and permit references between the loaded projects. For example, Microsoft Excel permits loading of multiple projects and **Option Private Module** can be used to restrict cross-project visibility. Although Visual Basic permits loading of multiple projects, references between projects are never permitted in Visual Basic.

Module 7 – Error Handling

Error handling refers to the programming practice of anticipating and coding for error conditions that may arise when your program runs. Errors in general come in three flavors: compiler errors such as undeclared variables that prevent your code from compiling; user data entry error such as a user entering a negative value where only a positive number is acceptable; and run time errors, that occur when VBA cannot correctly execute a program statement.  We will concern ourselves here only with run time errors.  Typical run time errors include attempting to access a non-existent worksheet or workbook, or attempting to divide by zero. The example code in this article will use the division by zero error (Error 11) when we want to deliberately raise an error.

Your application should make as many checks as possible during initialization to ensure that run time errors do not occur later.  In Excel, this includes ensuring that required workbooks and worksheets are present and that required names are defined.  The more checking you do before the real work of your application begins, the more stable your application will be. It is far better to detect potential error situations when your application starts up before data is change than to wait until later to encounter an error situation.

If you have no error handling code and a run time error occurs, VBA will display its standard run time error dialog box. While this may be acceptable, even desirable, in a development environment, it is not acceptable to the end user in a production environment. The goal of well designed error handling code is to anticipate potential errors, and correct them at run time or to terminate code execution in a controlled, graceful method.  Your goal should be to prevent unhandled errors from arising.

*A note on terminology:* Throughout this module, the term *procedure* should be taken to mean a Sub, Function, or Property procedure, and the term *exit statement* should be taken to mean Exit Sub, Exit Function, or Exit Property.  The term *end statement* should be taken to mean End Sub *,* End Function,  End Property, or just  End.

Types of Errors

No matter how well you are at writing VBA code, sooner or later you will encounter an error when running your program. For example, you may have type a keyword incorrectly or misspell the syntax. If such an error occurred, you won’t even be able to execute the procedure until you correct it. When working with VBA, you should be aware of these classes of errors.

**Design errors**

This error consists of syntax errors that occur when you mistype a statement. The statement MsgBox (“Simple Error” , produces a syntax error because you have omitted the closing parenthesis.

**Compile errors**

Compiling is the process of converting or translating the VBA code into a format that the computer can understand. This compiling process happened very fast and you are typically unaware  of it happening.

If any errors occur during the compile process, an error message pops up, and the VBE will highlights the location of the error.

**Runtime errors**

This error occurs when your code executes. VBE will display a message box informing you of the error. A common example will be the TYPE MISMATCH error when you pass the wrong data type value to a variable. If you pass a string to an expression that expects a numeric value, a runtime error occurs.

Generally to avoid your program stop due to runtime error you need to use the **On Error Resume Next** statement at the beginning of your procedure. Try not to do this as the program still run although there are errors. The bottom line is you still need to handle the errors.

**Logical errors**

A logical error do not produce any type of error message. Simple logical errors include mistyping a value, placing a decimal point in the wrong place.

Debugging your code is the process of finding and correcting run-time errors and logical errors.

Eight Bug Reduction Tips

Few tips that will help you keep them to a minimum.

1. **Use an Option Explicit at the beginning of your module.** Doing so will require that you define the data type for every variable that you use. It's a bit more work, but you'll avoid the common error of misspelling a variable name.. And there's a nice side benefit: Your routines will often run faster.
2. **Format your code with indentation.** I've found that using indentation to delineate code segments is quite helpful. If you have several nested For...Next loops, for example, consistent indentation will make it much easier to keep track of them all.
3. **Be careful with On Error Resume Next.** This statement causes Excel to ignore any errors and continue. In some cases, using this statement will cause Excel to ignore errors that shouldn't be ignored. Your may have bugs and not even realize it.
4. **Use lots of comments.** Nothing is more frustrating than revisiting code that you wrote six months ago - and not having a clue as to how it works. Adding a few comments to describe your logic can save you lots of time down the road.
5. **Keep your subroutines and functions simple.** Writing your code in smaller modules, each of which has a single, well-defined purpose, makes it much easier to debug them.
6. **Use the macro recorder to help you identify properties and methods.** If I can't remember the name or syntax of a property or method, I've found that it's often quicker to simply record a macro and look at the recorded code.
7. **Consider a different approach.** If you're having trouble getting a particular routine to work correctly, you might want to scrap the idea and try something completely different. In most cases, Excel offers several alternative methods of accomplishing the same thing.
8. **Understand Excel's debugger.** Although it can be a bit daunting at first, you'll find that Excel's debugger is an excellent tool. Invest some time and get to know it. I used VBA for quite a while before I took the time to learn how the debugger works (it's well documented in the on-line help). I spent about an hour learning the details, and I estimate that it has saved me dozens of hours in wasted time.

The Visual Basic Editor will help you avoid errors in coding in many different ways. You will not have to wait at the end to be told that there is something wrong with your macro.

Spelling Errors

You have seen in lesson 11 the VBE capitalise letters to let you know that there are no spelling errors.

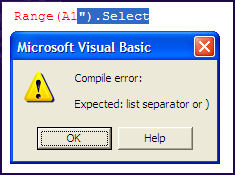
Syntax Errors

The VBE will also tell you that there is a syntax error in what you have just written by making the font red and showing you a message box.

Exercise :

1. Open a new workbook in Excel and use go to the visual basic editor (VBE).
2. In the **Code Window** of any of the sheet copy/paste the following line of code: **Range(A1").Select** and click Enter.

You get the following message box telling you that you are missing a "list separator". Look for the error before the segment highlighted in blue. We can deduce that VBA is talking about the missing quotation mark.



1. Click on the "OK" button.
2. Add the missing quotation mark, use the mouse to move the cursor to the end of the sentence and click Enter. The font is black meaning that everything is correct.

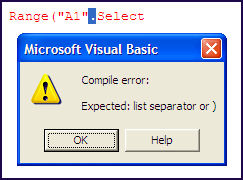
Exercise :

1. In the **Code Window** that you have used previous exercise copy/paste the following line of code:

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

and click Enter.

You get the following message box telling you that you are missing a "list separator". Look for the error before the segment highlighted in blue. We can deduce that VBE is talking about the missing parenthesis. Both the quotation marks in the exercise above and the parenthesis in this exercise are considered as "list separator" by the VBE.



1. Click on the "OK" button
2. Add the missing parenthesis, use the mouse to move the cursor to the end of the sentence and click Enter. The font is black meaning that everything is correct.

On Error

The heart of error handling in VBA is the **On Error** statement. This statement instructs VBA what to do when an run time error is encountered.  The **On Error** statement takes three forms.

On Error Goto 0  
On Error Resume Next  
On Error Goto <label>:

The first form, **On Error Goto 0**, is the default mode in VBA.  This indicates that when a run time error occurs VBA should display its standard run time error message box, allowing you to enter the code in debug mode or to terminate the VBA program. When On **On Error Goto 0** is in effect, it is the same as having no enabled error handler.  Any error will cause VBA to display its standard error message box.

On Error Resume Next

The second form, **On Error Resume Next**, is the most commonly used and misused form.  It instructs to VBA to essentially ignore the error and resume execution on the next line of code. It is very important to remember that **On Error Resume Next** does***not*** in any way "fix" the error. It simply instructs VBA to continue as if no error occured. However, the error may have side effects, such as uninitialized variables or objects set to **Nothing**.  It is the responsibility of your code to test for an error condition and take appropriate action.  You do this by testing the value of **Err.Number** and if it is not zero execute appropriate code.  For example,

    On Error Resume Next  
    N = 1 / 0    ' cause an error  
    If Err.Number <> 0 Then   
        N = 1  
    End If

This code attempts to assign the value 1 / 0 to the variable N. This is an illegal operations, so VBA will raise an error 11 -- Division By Zero -- and because we have **On Error Resume Next** in effect, code continues to the **If** statement. This statement tests the value of **On Err.Number** and assigns some other number to **N**.

On Error Goto

The third form **On Error** of is **On Error Goto *<Label>***: which tells VBA to transfer execution to the line following the specified line label. Whenever an error occurs, code execution immediately goes to the line following the line label.  None of the code between the error and the label is executed, including any loop control statements.

    On Error Goto ErrHandler:  
    N = 1 / 0    ' cause an error  
    '  
    ' more code  
    '  
    Exit Sub   
    ErrHandler:  
    ' error handling code  
    Resume Next  
    End Sub

Enabled and Active Error Handlers

An error handler is said to be **enabled** when an  **On Error** statement is executed.  Only one error handler is enabled at any given time, and VBA will behave according to the enabled error handler.  An **active** error handler is the code that executes when an error occurs and execution is transferred to another location via a **On Error Goto *<Label>***: statement.

Error Handling Blocks and On Error Goto

An error handling block, also called an error handler, is a section of code to which execution is tranferred via a **On Error Goto *<Label>***: statement. This code should be designed either to fix the problem and resume execution in the main code block or to terminate execution of the procedure. You can't use to the **On Error Goto *<Label>***:  statement merely skip over lines. For example, the following code will not work properly:

On Error GoTo Err1:  
    Debug.Print 1 / 0  
    ' more code  
Err1:  
    On Error GoTo Err2:  
    Debug.Print 1 / 0  
    ' more code  
Err2:  
  
When the first error is raised, execution transfers to the line following Err1:. The error hander is still active when the second error occurs, and therefore the second error is not trapped by the On Error statement.

Error Handling With Multiple Procedures

Every procedure need not have a error code. When an error occurs, VBA uses the last On Error statement to direct code execution. If the code causing the error is in a procedure with an On Error statement, error handling is as described in the above section. However, if the procedure in which the error occurs does not have an error handler, VBA looks backwards through the procedure calls which lead to the erroneous code. For example if procedure A calls B and B calls C, and A is the only procedure with an error handler, if an error occurs in procedure C, code execution is immediately transferred to the error handler in procedure A, skipping the remaining code in B.

A Note of Caution

It is tempting to deal with errors by placing an On **Error Resume Next** statement at the top of the procedure in order to get the code to run without raising an error.  This is very bad coding practice. Remember that using **Error Resume Next** does ***not*** fix errors.  It merely ignores them.

The Future Of Error Handling In VBA

Error handling in VB6 and VBA is based on the **On Error** statement, which leads to awkward code structure. Languages like C++ provide a code structure call **Try/Catch** that allows much more granularity and control.  At some point, Microsoft will introduce their NET framework in to Office, and when this happens, VBA programmers will have at their disposal the language features of **Try/Catch/Finally** code structure that VB.NET developers already enjoy.

Module 8 – Special Constructs

For Each

Use a **For Each...Next** loop when you want to repeat a set of statements for each element of a collection or array.

Visual Basic evaluates the collection only one time, before the loop starts. If your statement block changes ***element*** or **group**, these changes do not affect the iteration of the loop.

When all the elements in the collection have been successively assigned to ***element***, the ***For Each*** loop stops and control passes to the statement following the **Next** statement.

If **element**has not been declared outside this loop, you must declare it in the ***For Each*** statement. You can declare the type of ***element*** explicitly by using an **As** statement, or you can rely on type inference to assign the type. In either case, the scope of ***element*** is the body of the loop. However, you cannot declare **element**both outside and inside the loop.

You can optionally specify ***element*** in the ***Next*** statement. This improves the readability of your program, especially if you have nested ***For Each*** loops. You must specify the same variable as the one that appears in the corresponding ***For Each*** statement.

You might want to avoid changing the value of **element**inside a loop. Doing this can make it more difficult to read and debug your code. Changing the value of ***group*** does not affect the collection or its elements, which were determined when the loop was first entered.

For Each element [ As datatype ] In group

[ statements ]

[ Continue For ]

[ statements ]

[ Exit For ]

[ statements ]

Next [ element ]

|  |  |
| --- | --- |
| **Term** | **Definition** |
| *element* | Required in the For Each statement. Optional in the Next statement. Variable. Used to iterate through the elements of the collection. |
| *datatype* | Required if *element* is not already declared. Data type of *element*. |
| *group* | Required. Object variable. Refers to the collection over which the *statements* are to be repeated. |
| *statements* | Optional. One or more statements between For Each and Next that run on each item in *group*. |
| Continue For | Optional. Transfers control to the start of the For Each loop. |
| Exit For | Optional. Transfers control out of the For Each loop. |
| Next | Required. Terminates the definition of the For Each loop. |

A For Each Loop is used to loop through each element is a **Collection**, or an Array. A **Collection** is an object that contains a set of related objects. An **Array** is a set of sequentially indexed elements having the same intrinsic data type. To stop an endless loop, press Esc or Ctrl+ Break

To get the gist of what the example below is doing fill a few rows in Column "A" of the active sheet with some values. We could force an Exit from the For Each...Loop via the Exit For Statement. E.g. **If rCell=10 Then Exit For**

Sub For\_Each\_Collection()

Dim rRange As Range

Dim rCell As Range

Set rRange = Range("A1", Range("A65536").End(xlUp))

For Each rCell In rRange

MsgBox rCell.Value

Next rCell

End Sub

Sub For\_Each\_Array()

Dim lArray(10) As Long

Dim lArr

Dim lCount As Long

'Fill Array

For lCount = 0 To 10

lArray(lCount) = lCount

Next lCount

lCount = 0

'Show each value in Array

For Each lArr In lArray

MsgBox "The number " & lCount & " element in lArray is " & lArr

lCount = lCount + 1

Next lArr

End Sub

Abnormal termination with Exit keyword

Exits a procedure or block and transfers control immediately to the statement following the procedure call or the block definition.

Exit { Do | For | Function | Property | Select | Sub | Try | While }

|  |  |
| --- | --- |
| **PARTS** | |
| **Do** | Immediately exits the Do loop in which it appears. Execution continues with the statement following the Loop statement. Exit Do can be used only inside a Do loop. When used within nested Do loops, Exit Do exits the innermost loop and transfers control to the next higher level of nesting. |
| **For** | Immediately exits the For loop in which it appears. Execution continues with the statement following the Next statement. Exit For can be used only inside a For...Next or For Each...Next loop. When used within nested For loops, Exit For exits the innermost loop and transfers control to the next higher level of nesting |
| **Function** | Immediately exits the Function procedure in which it appears. Execution continues with the statement following the statement that called the Function procedure. Exit Function can be used only inside a Function procedure. |
| **Property** | Immediately exits the Property procedure in which it appears. Execution continues with the statement that called the Property procedure, that is, with the statement requesting or setting the property's value. Exit Property can be used only inside a property's Get or Set procedure. |
| **Select** | Immediately exits the Select Case block in which it appears. Execution continues with the statement following the End Select statement. Exit Select can be used only inside a Select Case statement. |
| **Sub** | Immediately exits the Sub procedure in which it appears. Execution continues with the statement following the statement that called the Sub procedure. Exit Sub can be used only inside a Sub procedure. |
| **Try** | Immediately exits the Try or Catch block in which it appears. Execution continues with the Finally block if there is one, or with the statement following the End Try statement otherwise. Exit Try can be used only inside a Try or Catch block, and not inside a Finally block. |
| **While** | Immediately exits the While loop in which it appears. Execution continues with the statement following the End While statement. Exit While can be used only inside a While loop. When used within nested While loops, Exit While transfers control to the loop that is one nested level above the loop where Exit While occurs. |

[[http://i.msdn.microsoft.com/Hash/030c41d9079671d09a62d8e2c1db6973.gif](javascript:void(0))Remarks](javascript:void(0))

Do not confuse **Exit** statements with **End** statements. **Exit** does not define the end of a statement.

[[http://i.msdn.microsoft.com/Hash/030c41d9079671d09a62d8e2c1db6973.gif](javascript:void(0))Example](javascript:void(0))

The following example uses the **Exit** statement to exit a **For**...**Next** loop, a **Do** loop, and a **Sub** procedure.

Sub exitStatementDemo()

Dim demoNum As Single

' Set up an infinite loop.

Do

For i As Integer = 1 To 10000000

demoNum = Int(Rnd() \* 100)

Select Case demoNum

Case 7 : Exit For

Case 29 : Exit Do

Case 54 : Exit Sub

End Select

Next i

Loop

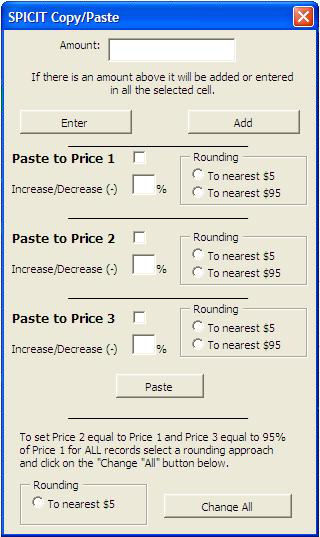
End Sub

Module 9 – User Form

This module covers how to create an **Excel VBA Userform** (also known as a dialog box).

The purpose

When the message box or the input box is not enough to communicate with the user you can create your own userforms or dialog windows like the one below.

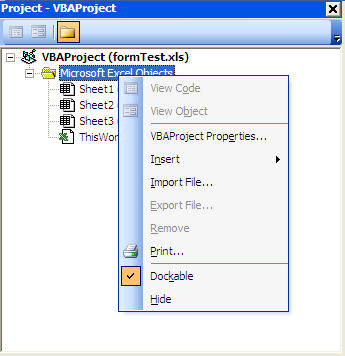


The userform is a small or large dialog window that you create and used to require information from the user to feed the VBA procedure/macros. To these userforms you will add controls (Command Buttons, Text Boxes, List Boxes, Option Buttons, Frames, Spin Buttons， images and others) and program them under event driven model.

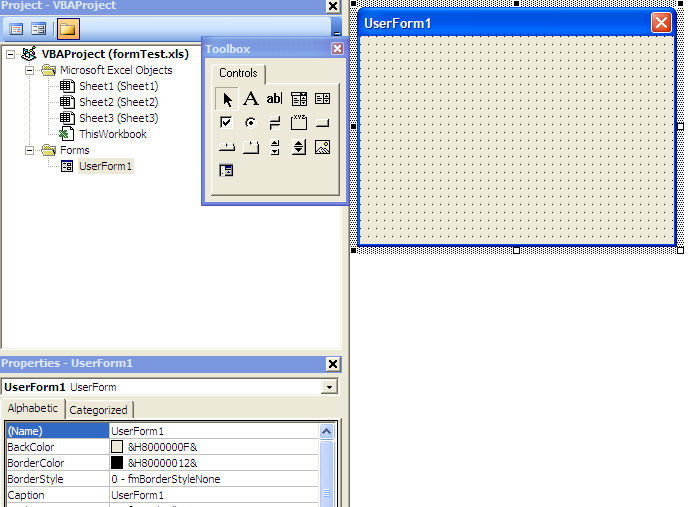
How to Declare?

Userforms are created in the **Project Window** of the Visual Basic Editor. You will also find the toolbox that allows you to add controls to your userforms in the Visual Basic Editor.

In the Visual Basic Editor you right click in the project window and you will see this menu appear:



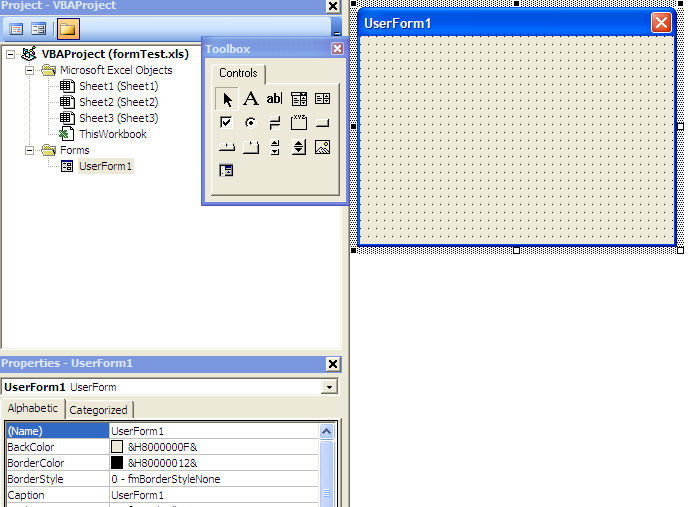
Go to "**Insert**" and select "**UserForm**". You will then see the following:



On the right you see the userform that you have just added to your workbook. On the left is the toolbox with all the controls that you can add to your userform. You can hide that toolbox by clicking on the "X" and bring it back by clicking on the toolbox icon  http://www.excel-vba.com/www-toolbox.jpg or by going to the menu bar "**View**/**Toolbox**".

Userforms Properties

When you double click on the userform name in the project window of the Visual Basic Editor the properties windows shows properties of the userform.



How to call it?

You can use the **Show** method of the userform

Userform1.Show

For example you can then, insert Shape in Excel (Ractangle, oval etc)

Right click on the shape and click the Assign Micro option.

**Note:** A new window will open asking your macro name (module name)

In that window, click the New button.

**Note:** It will create a new macro for you.

e.g.

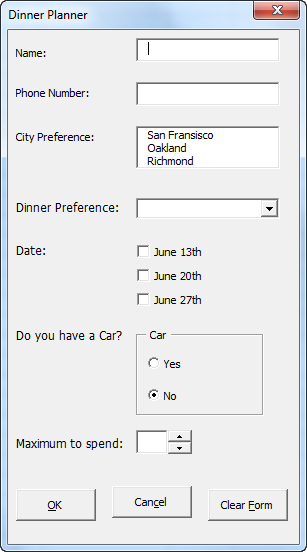
Sub Rectangle1\_Click()

Userform1.Show

End Sub

Exercise :

The **Userform** we are going to create looks as follows.



Controls

The most important Controls that can be added to an **Excel VBA Userform** are:

**Labels**

Examples of labels in our Userform are: 'Name:' , 'Phone Number:' , 'City Preference:' , 'Dinner Preference:' , etc.

**Textboxes**

The three boxes next to the Labels 'Name:', 'Phone Number:' and 'Maximum to spend:' are textboxes.

**Listboxes**

The box next to the Label 'City Preference:' is a listbox.

**Comboboxes**

The dropdown-list next to the Label 'Dinner Preference:' is a combobox.

**Note:** a combobox is the same as a listbox except that the user can now also fill in his/her own choice if his/her choice is not included in the list.

**Checkboxes and Option buttons**

'June 13th', 'June 20th' and 'June 27th' are examples of Checkboxes. 'Yes' and 'No' are examples of option buttons.

**Note:** Checkboxes and option buttons are primarily the same except that option buttons are dependent on each other while checkboxes are not. This means that when you want to check one option button the other option button will automatically uncheck.

**Frames**

The field with name 'Car' including the two option buttons is a Frame control.

**Note:** in order for option buttons to have the dependent functionality described earlier it is best to place the option buttons in a Frame control.

**Command buttons**

The three command buttons at the bottom of the Userform are examples of command buttons.

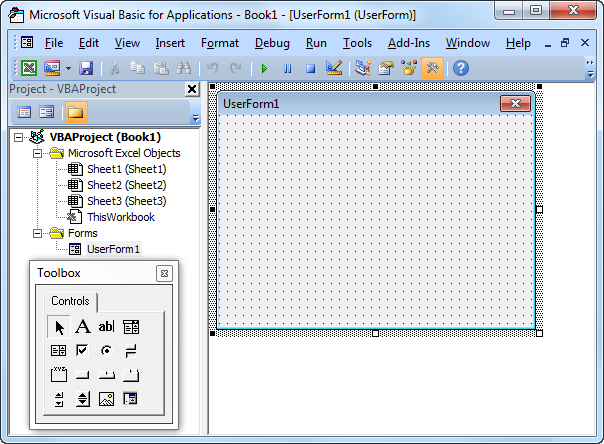
**Spin buttons**

The spin button is placed next to the textbox at the bottom of the Userform.

Create the Userform

Now it is time to create the Userform!

1. Launch the Visual Basic Editor.
2. Click on ThisWorkbook from the Project Explorer. If the Project Explorer is not visible, click on View and then Project Explorer.
3. From the Menu click on Insert and then Userform. Your screen should be set up as below:



1. If the Toolbox does not appear automatically, click on View and then Toolbox.
2. Add all the controls. Once this has been completed, the result should be consistent with the picture of the Userform shown earlier. For example, create a Label by clicking on Label from the Toolbox. Next, you can drag a Label on the Userform. When you arrive at the CarFrame, remember to draw this Frame first before you place the two option buttons in it.
3. Change the names and the captions of the controls. Right mouse click on each control. Then click on Properties. Names are used in the Excel VBA code. Captions are those that appear on your screen. Change the names and captions of the controls according to the table below.

|  |  |  |
| --- | --- | --- |
| **Control** | **Name (in VBA)** | **Caption** |
| Userform | DinnerPlannerUserForm | Dinner Planner |
| Textbox | NameTextBox | N/A |
| Textbox | PhoneTextBox | N/A |
| Listbox | CityListBox | N/A |
| Combobox | DinnerComboBox | N/A |
| Checkbox | DateCheckBox1 | June 13th |
|  | DateCheckBox2 | June 20th |
|  | DateCheckBox3 | June 27th |
| Frame | CarFrame | Car |
| Option button | CarOptionButton1 | Yes |
|  | CarOptionButton2 | No |
| Textbox | MoneyTextBox | N/A |
| Spin button | MoneySpinButton | N/A |
| Command button | OKButton | OK |
| Command button | CancelButton | Cancel |
| Command button | ClearButton | Clear Form |
| 9 Labels | No need to change | See Picture |

**Note:** it is good practice to change the names of controls. This will make your code easier to read.

Show the Userform

To show the Userform, place a command button on your worksheet and add the following code line:

Private Sub CommandButton1\_Click()  
  
 DinnerPlannerUserForm.Show  
  
End Sub

UserForm\_Initialize

The Sub UserForm\_Initialize runs automatically whenever the **Userform** is loaded. Thus, when you use the Show method for the Userform, the code will automatically be executed.

Private Sub UserForm\_Initialize()  
  
 'Empty NameTextBox  
 NameTextBox.Value = ""  
  
 'Empty PhoneTextBox  
 PhoneTextBox.Value = ""  
  
 'Empty CityListBox  
 CityListBox.Clear  
  
 'Fill CityListBox  
 With CityListBox  
    .AddItem "San Fransisco"  
    .AddItem "Oakland"  
    .AddItem "Richmond"  
 End With  
  
 'Empty DinnerComboBox  
 DinnerComboBox.Clear  
  
 'Fill DinnerComboBox  
 With DinnerComboBox  
    .AddItem "Italian"  
    .AddItem "Chinese"  
    .AddItem "Frites and Meat"  
 End With  
  
 'Uncheck DataCheckBoxes  
 DateCheckBox1.Value = False  
 DateCheckBox2.Value = False  
 DateCheckBox3.Value = False  
  
 'Set no car as default  
 CarOptionButton2.Value = True  
  
 'Empty MoneyTextBox  
 MoneyTextBox.Value = ""  
  
 'Set Focus on NameTextBox  
 NameTextBox.SetFocus  
  
End Sub

Result of the code: Fields are emptied, lists are populated and checkboxes are unchecked. CarOptionButton2 is set as default, assuming that most of the people do not have a car. Finally, the last code line sets the focus on NameTextbox as this is where we want to start when the Userform is loaded. To add this code to the Userform, execute the following steps.

* 1. Launch the Visual Basic Editor.
  2. Click on View and then Code from the Menu or right click on the Userform and then View Code.
  3. The code window will appear showing you two drop-down lists. Choose Userform from the left drop-down list. Choose Initialize from the right drop-down list.
  4. Add the code.

### Assign a macro to the Cancel button

To close the Excel VBA Userform when you click on the Cancel button, execute the following steps.

1. Launch the Visual Basic Editor.
2. Double click on DinnerPlannerUserForm from the Project Explorer.
3. Double click on the Cancel button.
4. Add the following code line:

Private Sub CancelButton\_Click()  
  
 Unload Me  
  
End Sub

Assign a macro to the Clear button

To call the **Sub UserForm\_Initialize** when you click on the Clear button, execute the following steps.

* 1. Launch the Visual Basic Editor.
  2. Double click on DinnerPlannerUserForm from the Project Explorer.
  3. Double click on the Clear button.
  4. Add the following code line:

Private Sub ClearButton\_Click()  
  
 Call UserForm\_Initialize  
  
End Sub

Assign a macro to the OK button

After clicking on the OK button, the information from the Userform will be placed on your worksheet.

Private Sub OKButton\_Click()  
  
 Dim emptyRow As Long  
  
 'Make Sheet1 Active  
 Sheets(1).Activate  
  
 'Determine emptyRow  
 emptyRow = WorksheetFunction.CountA(Range("A:A")) + 1  
  
 'Export Data to worksheet  
 Cells(emptyRow, 1).Value = NameTextBox.Value  
 Cells(emptyRow, 2).Value = PhoneTextBox.Value  
 Cells(emptyRow, 3).Value = CityListBox.Value  
 Cells(emptyRow, 4).Value = DinnerComboBox.Value  
  
 If DateCheckBox1.Value = True Then

Cells(emptyRow, 5).Value = DateCheckBox1.Caption

End If  
  
 If DateCheckBox2.Value = True Then

Cells(emptyRow, 5).Value = Cells(emptyRow, 5).Value & " " & \_ DateCheckBox2.Caption  
 End If  
  
 If DateCheckBox3.Value = True Then

Cells(emptyRow, 5).Value = Cells(emptyRow, 5).Value & " " & \_ DateCheckBox3.Caption  
 End If  
  
 If CarOptionButton1.Value = True Then  
    Cells(emptyRow, 6).Value = "Yes"  
 Else  
    Cells(emptyRow, 6).Value = "No"  
 End If  
  
 Cells(emptyRow, 7).Value = MoneyTextBox.Value  
  
End Sub

**Explanation of the code:** First, we activate Sheet1. Next, we determine emptyRow. The information from the Userform will be placed in this row. EmptyRow increases every time a record is added. Finally, we take the information from the Userform to the specific columns of emptyRow. Double click on the OK button to add the code just like we did with the Cancel and the Clear button.

Assign a macro to the Money Spin Button

By using the Money Spin Button the user can indicate how much he/she wants to spend. Execute the following steps to program this:

* 1. Launch the Visual Basic Editor.
  2. Double click on DinnerPlannerUserForm from the Project Explorer.
  3. Double click on the Money Spin Button.
  4. Add the following code line:

Private Sub MoneySpinButton\_Change()  
  
 MoneyTextBox.Text = MoneySpinButton.Value  
  
End Sub

Test the Userform

Exit the Visual Basic Editor, populate row 1, and test the Userform.

**Result:**



Module 10 – Class Module

This module covers the concepts that you need to understand to design, build, and use custom objects with their own methods and properties. As the new additions to the VBA language in Microsoft Office 2000: the ability to add custom events to your objects, and the ability to extend your objects by implementing interfaces.

The Purpose

Before getting in to classes and objects in detail, it will prove useful to examine briefly a class's logical ancestor, the **Type** declaration.

A **Type** is made up of other basic variable types. You may be familiar with Types from other programming languages, in which they are called a **struct**, **structure**, or **record**. For example, we could declare a Type that describes an employee:

Type Employee

Name As String

Address As String

Salary As Double

End Type

This defines a single type named **Employee** which has three elements: **Name, Address,** and **Salary.** You can then create variables of the Employee type and give values to the elements. For example,

Dim Manager As Employee  
Manager.Name = "Joe Smith"  
Manager.Address = "123 Main Street"  
Manager.Salary = 40000

Types are quite useful, but have three shortcomings.

1. You can't declare new instances of a **Type**. You must declare all the variables you'll need at design time or you need a dynamic array that is resized with **Redim Preserve**, an awkward and expensive operation.
2. You have no control over what values are assigned to the elements of a **Type.** For example, there is nothing to prevent the assignment of a negative value to the **Salary** element.
3. A **Type** can't do anything. It cannot carry out actions; it is simply a static data structure.

While Types have their place (they are used extensively in Windows API functions), a class module is often a better solution.

**When to use class modules**

Here are some examples on when you would want to use class modules:

* Database management Class modules make it easier to create objects that can manage a database with VBA. You can create a class that contains code for reading or writing to a database table. This class can be used in your macro without the user knowing how and where the data comes from.
* Wrapping in API procedures Class modules makes it easier to use Windows API-functions in your macros. Wrapping the API-functions in a class module makes the development easier for people that are not that familiar with the more complicated function in Windows.
* Managing reading and writing to text files This is really not that complicated, but who goes around and remember how this is done in VBA? By creating a class that manage low level reading to and writing from a text file, the properties and methods in the class can easily be used in your macros.

Classes are a powerful tool in intermediate to advanced level VBA programming.  This page is an introduction to what a class and an object are and will hopefully get you started working with classes. This is by no means a comprehensive guide.

In VBA, a *class* is defined in **Class module** and serves as a template for an **object**. The term **Object** is deliberately vague. An object can be defined to represent whatever you want. Anything that you can describe conceptually can be represented by a class. The difference between a class and an object is that a class does nothing and consumes no memory. When you have a variable of that class type and create instance of that class with the **New** keyword, a process called **instantiating**, it becomes an object and consumes memory and can carry out actions. A class is defined by its properties, which describe attributes of the class, and its methods (sub and function procedures), which carry out actions in the object. If a class is analogous to a noun, a property is like an adjective -- it describes the object. A method is like a verb -- it carries out an action.

You must instantiate a class into an object in order to do anything with it. There is nothing you can do with a class module beyond creating an object from it. An example of instantiation is shown below:

Dim C As MyClass  
Set C = New MyClass

Unlike other languages, VB/VBA allows for only one class in a class module, and the name of the class is the name of the module. You can now work with the properties and methods defined in MyClass in the C object variable.

**NOTE:** It is also possible to combine the two statements above into a single statement:

Dim C As New MyClass

This is called an **auto-instancing** variable. When the variable C is first encountered in code, a new instance is created. In general, you should avoid auto-instancing variables for two reasons:

1. It adds overhead to the code because the variable must be tested for **Nothing** every time it is encountered in code.
2. You have no way to test whether an auto-instancing variable is **Nothing** becaue the very act of using the variable name in an If Obj Is Nothing Then statement will automatically create an instance of the variable.

Programmers have managed without class modules earlier, and the reason for using class modules may not become obvious before you have used class modules for a while. Here are some of the advantages when using class modules:

* Class modules make it possible to separate complicated source code for advanced processes. This makes it easier for other to use the source code without understanding how the process is performed.
* Class modules make the development of complicated tasks easier by breaking up the code in smaller and easier manageable parts. This has been possible earlier, but a class module forces you to separate the code from the ordinary procedures, resulting in a more obvious function separation.
* Class modules let you create reusable components. Because of the obvious separation between classes and the procedures using them, the classes contain independent code components that can easily be shared between different projects.
* Class module is the foundation of other component technologies, Visual Basic can be used to create X Automation servers and ActiveX controls.

In Custom objects don't add new functionality to your code. What they can do is make complex operations appear simpler, sophisticated solutions more self-documenting and maintainable, and procedures that required hours of coding time more reusable, both for yourself and for other programmers. In fact, creating a custom object is an ideal way to package your code for other VBA programmers to use, because rather than figuring out how to call your code. They can work with your custom object much as they would with any built-in object.

As an analogy, think of a car. It's made up of complex mechanical systems working together to get you where you're going. Although there are those who like to tinker under the hood, most people don't want to think about how the car works. They're content to press on the gas pedal, without considering the fuel lines, filters, fuel-injection system, cylinders, and pistons that respond to that simple action. Even most mechanics don't want to think about those things every time they drive. The gas pedal, brake, steering wheel, and gearshift provide an **interface** for the driver, hiding the complexity that's actually at work in the system. Although it wouldn't be very practical, you can imagine a car that would run only if the driver pumped the fuel through the lines, mixed a certain ratio of gasoline and air together, and instructed the pistons to drive up and down at a certain rate.

Coding without objects doesn't make your life that difficult, of course. The point is that effort invested in simplicity can pay off for yourself and others in the long term. Once you've created and tested an object, you can treat it as a "black box" and forget about the code that makes the object behave the way it does.

You can build entire custom object models that involve complex code behind the scenes, but that present relatively simple and intuitive object syntax to the programmer. This is, in fact, what all of the Office and VBA object models do for you — they take complex operations and package them into easy-to-use objects, methods, properties, and events. When you set the **Visible** property of a Microsoft Excel object to **False**, for example, you don't have to worry about how VBA, Excel, and Windows cooperate to hide the object. You can focus on the larger-scale goals of building your solution.

Naturally, simplicity and reusability come at a price. Creating custom objects and object models requires a different, perhaps even revolutionary, way of thinking than the sort of programming you may be accustomed to doing in VBA. It can take time to get the hang of it. In the long run, though, coding with objects can make you a better programmer by increasing your efficiency, honing your design skills, and making it easier to reuse your code.

Basic Class Concepts

In case you're not yet familiar with using class modules to build custom objects, this section covers the basics of adding a class to your project, creating an instance of a class in memory, and constructing properties and methods.

Properties Declaration

As mentioned previously, public module-level variables in a class module function as properties of an object. However, they're not very sophisticated. If you need to run code in order to set or return a property's value, or you want to make a property read-only, you can create a **Property** procedure. There are three types of **Property** procedures: **Property** **Get**, **Property** **Let**, and **Property** **Set** procedures. The **Property** **Get** procedure returns the current value of a property, whereas the **Property** **Let** procedure sets the value. The **Property** **Set** procedure assigns an object to an object property.

To create a read-write property, you need to include a pair of **Property** procedures in the class module. Both procedures must have the same name. If the property stores and returns a *scalar* value, such as a numeric, text, or date value, you use a **Property** **Let** procedure to set the value and a **Property** **Get** procedure to retrieve it. If the property stores and returns a reference to an object, you use the **Property** **Set** procedure to store the reference and the **Property** **Get** procedure to return it.

You can also create read-only, write-only, and write-once properties. The following table outlines which property procedures you need for each type.

|  |  |
| --- | --- |
| Type of property | Procedures needed |
| Read-write, scalar | **Property Let, Property Get** |
| Read-write, object | **Property Set, Property Get** |
| Read-only, scalar | **Property Get** |
| Read-only, object | **Property Get** |
| Write-only, scalar | **Property Let** |
| Write-once, scalar | **Property Let**, including code to determine whether property has been set previously, **Property Get** |
| Write-once, object | **Property Set**, including code to determine whether object property has been set previously, **Property Get** |

Creating Events and Event Procedures

You already understand how to handle form and control events. If you've programmed in Word or Excel, you may also have taken advantage of events on the **ThisDocument**, **ThisWorkbook**, or **Sheet*N*** objects. These event procedures are simple to create — you just open the class module and construct the event procedure from the **Object** and **Procedure** drop-down lists in the Code window.

There are two additional ways to handle events from VBA code in an Office application. You can create event procedures for certain objects that provide built-in events but that don't have associated class modules. You can also create custom events for your own classes.

Extending Objects through Interfaces

Suppose that in the process of designing your solution, you decide that you want to create several objects that are closely related, and in fact require at least some of the same properties and methods. Also, in the future, you might need to add more objects that are related to these, and you'd like to make the process as easy as possible for yourself down the road. You can **implement an interface** that defines the properties and methods that these objects have in common.

An interface is the set of properties, methods, and events that define an object's characteristics and behavior. Every object has an interface, whether it's a built-in or custom object. When implement an interface in a class module, you take advantage of another object's interface to provide some or all of the properties and methods for that class. In this way objects can be extended, related according to their functionality, and maximize your code's reusability. Implementing interfaces can take advantage of **polymorphism**, the ability to create objects that have specific individual functionality, but that share something in common with a more general object.

**Note:**   If you've programmed in OOP languages, you may be familiar with the concepts of polymorphism and inheritance. Inheritance refers to the ability of a class to derive members (and their functionality) from other classes. By implementing an interface, you can achieve polymorphism. However, interfaces don't provide true inheritance, because a class can implement only those members that are defined within an interface that it implements. For example, if Class B implements the interface for Class A, and Class C implements the interface for Class B, Class C must implement all the members of Class B, but it cannot directly implement members of Class A. In true inheritance, objects can derive characteristics from an entire hierarchy of other objects. For example, if Class B inherits from Class A, and Class C inherits from Class B, and so on, Class E can selectively derive all or some characteristics from Classes A, B, C, and D.

As a conceptual example of polymorphism, consider **Control** objects. There are several different control classes. For example, **CommandButton** controls are created from the **CommandButton** class, and **TextBox** controls are created from the **TextBox** class. You can create a variable of type **CommandButton** and assign to it a reference to a **CommandButton** control. You can do the same with a variable of type **TextBox**. You can't, however, assign a reference to a **TextBox** control to a variable of type **CommandButton** — you'll get an error if you try it.

On the other hand, both types of controls also have the more general type **Control**. If you create a variable of type **Control**, you can assign a reference to either a command button or a text box to the variable. This flexibility is indispensable in cases where you need to enumerate through all the controls on a form and set their **Visible** properties to **True**, for example. When you know what type of control your code will work with, it's better to create a variable of the more specific type, but when you don't know ahead of time, you can create a variable of type **Control**.

The samples discussed in this section implement custom interfaces similar to the way that individual controls support the same **Control** class interface. There are a couple of ways to do this. The following sections discuss both ways in the context of examples. First, though, the next section covers some of the basics of implementing interfaces

Designing Object Models

Designing a custom object model can be a tricky business. If you dive in and start coding right away, you may find yourself realizing in the middle of your development process that your design is less than ideal. It pays to take some time to think through an object model, and even draw diagrams and make notes by hand. You may also want to study other object models to understand how they were constructed. When you design an object model, you are taking abstract processes and imposing concrete relationships upon them. In essence, you are creating artificial conceptual divisions for your code.

How to use it?

New instances of a class may be created with the **New** keyword and stored in a Collection or Dictionary object. Next, the properties of a class can be set or retrieved with **Property Let** and **Property Get** procedures, which can contain executable code. Thus, code could be written to raise an error or take other appropriate action if an invalid value is used to set a property value, such as a negative value for a Salary. Finally, classes have methods (sub and function procedures) which can carry out actions. In the example of an employee, there might be a method to print a paycheck for the employee.

Exercise :

Let's adapt the Employee Type (as previously described) into a class. Follow the steps below:

1. First, insert a class module into your VBProject (from the Insert menu in the VBA editor).
2. Name the class **CEmploye**e (it is common practice to use a 'C' as the first letter of a class).
3. There are three properties to create: Name, Address, and Salary. These values will be stored in private variables within the class. Since they are declared Private, they cannot be accessed outside the class module.

Private pName As String  
Private pAddress As String  
Private pSalary As Double

1. Declare Property procedures to allow these variables to be read from and written to. This is done with **Property Get** and **Property Let** functions (or **Property Set** for object type variables).

' Name property  
Public Property Get Name() As String   
    Name = pName   
End Property

Public Property Let Name(Value As String)   
    pName = Value   
End Property

' Address property  
Public Property Get Address() As String   
    Address = pAddress   
End Property

Public Property Let Address(Value As String)   
    pAddress = Value   
End Property   
  
' Salary property  
Public Property Get Salary() As Double   
    Salary = pSalary   
End Property

Public Property Let Salary(Value As Double)   
    pSalary = Value   
End Property

1. The **Get** procedure is used to return a value out of the class, and the **Let** procedure is to put a value into the class. Note that the return data type of the **Get** property procedure must be the same data type as the (last) parameter to the **Let** property procedure. Otherwise, you'll get a compiler error.

Because Property procedures can contain any code you like, the Let Salary procedure can be written to exclude non-positive values.

Public Property Let Salary(Value As Double)   
    If Value > 0 Then  
          pSalary = Value   
    Else  
        ' appropriate error code here  
    End If  
End Property

1. A property can be made read-only simply by omitting the Let procedure. For example, a read-only property might be withholding tax, which is calculated when it is called. E.g.,

Property Get WithholdingTax() As Double   
    WithholdingTax = calculated value   
End Property

1. Finally, the class can contain methods, such as a PrintPaycheck procedure.

Public Sub PrintPaycheck()   
    ' actual code to print check   
End Sub

Now that we have defined the class, we can create objects based on the class. In a standard code module, declare a variable of type CEmployee.

Dim Emp As CEmployee

Then, Set that variable to a new instance of the class and assign some property values.

Set Emp = New CEmployee  
Emp.Name = "Joe Smith"  
Emp.Address = "123 Main Street"  
Emp.Salary = 40000

Storing Multiple Objects in a Collection

If you need to store multiple instances of a class, such as for a group of employees, you can create mutliple objects from the class and store them in a Collection or Dictionary object, as shown below.

Dim Employees As Collection   
Dim Emp As CEmployee   
  
Set Employees = New Collection   
  
For Each Item In SomeList   
    Set Emp = New CEmployee   
    ' set properties for Emp   
    Employees.Add Emp   
Next Item

Now, you can use a simple For Each loop to loop through the collection and iterate through the collection and access each instance of CEmployee sequentailly:

Dim Emp As CEmployee  
For Each Emp In Employees  
    Debug.Print Emp.Name  
Next Emp

The Instancing Property Of A Class

The Instancing property of a class controls where that class may be used. The default value is **Private**, which means that the class can be used only in the project in which the class is defined. You can set the instancing property to **PublicNotCreatable**, which allows a variable to be declared as that class type in projects that have a reference to the project containing the class. The second class may declare a variable of the class type, but cannot create an instance of the class with the **New** keyword. See the next section for more details.

Self-Referencing An Instance Of A Class

Code within a Property or method of a class can refer to its own instance by using the **Me** reference. For example,

Private pName As String

Property Let Name(S As String)

pName = S

End Property

Public Sub SomeMethod()

' some code

Me.Name = "ABCD"

End Sub

Using Classes in Multiple Projects

If the Instancing property of the class is **PublicNotCreatable** a variable of that class type may be declared in other projects, but cannot be created in that project. You can use a function in the project containing the class to return a new instance to the caller. First, change the name of the project containing the class from the default value of **VBProject** to something meaningful like **projSourceProject**. Then, in the class that will use the class, set a reference to **projSourceProject.** Back in the project containing the class, create a procedure that will create and return a new instance of the class:

Public Function GetClass() As CEmployee   
   Set GetClass = New CEmployee   
End Function

Then call this function in the project that will use the class:

Dim NewEmp As projSourceProject.CEmployee   
Set NewEmp = projSourceProject.GetClass()

Setting the Default Value of A Class

You can specify a property to be the default property of a class. When you do this, you can omit that property name and the compiler will use the default property. For example if you made Name the default property, the following lines of code are functionally equivalent:

Emp.Name = "Joe Smith"

Emp = "Joe Smith"

If you are working with classes in VBA, it is often useful to make one member of a class the default member. For example, in the Excel **Range** object, the default member is **Value**. This allows you to omit the member name and use code like the following:

Range("A1") = 1234

' is the same as

Range("A1").Value = 1234

Because **Value** is the default member, it may be omitted in the code. Creating a default member of a class is also very useful (necessary, really) when you are working with a customized Collection class. In this case, you would likely specify the **Item** method as the default member. This allows you to use code like the following:

V = Coll(2)

' is the same as

V = Coll.Item(2)

Creating a Default Member in VBA

VBA does not directly support the creation of a default member of a class. That is, there is nothing in the VBA IDE that allows you to specify a default member. However, VBA does respect the default method if it is specified in a class. To specify a method as the default member, you need to Export the class module to a text file, edit that text file in NotePad or your favorite text editor, add an **Attribute** directive to the method, and then Import the text file back into the VBA Project.

First, export the class module to a text file. In VBA, go to the **File** menu and choose **Export File....** In the ***Save*** dialog that appears, navigate to some folder (it doesn't matter which folder), and save the class file as text with a **cls** extension. Next, select **Remove...** from the **File** menu and choose ***No*** in the **Do you want to export? *dialog.*** Next, open Notepad (C:\Windows\Notepad.exe) or another text editor, and open the cls that you saved in the Export step. In the text file, go to the method that you want to make the default, and add the following line of code.

Attribute Value.VB\_UserMemId = 0

An Attributedirective is an instruction to the c ompiler indicating various conditions for compilation. The Attribute directives are not visible in the VBA Editor and they cannot be added by the VBA Editor. You must use a text editor to add **Attribute** directives. If you are making the **Value** property the default member of your class, your code in Notepad should look similar to the following:

Property Get Value() As Long

Attribute Value.VB\_UserMemId = 0

Value = Whatever

End Property

You can make a Sub, Function, or Property the default member of the class, but only **one** procedure in the module may be the default member. Once you have added the **Attribute** directive to the text file, save the file and exit from NotePad. Now, in the VBA Editor, go to the **File** menu and **choose** Import **File...*.*** In the **Open** dialog that appears, navigate to the folder in which you saved the **cls** file and import it into VBA. Because Attributedirectives are not visible in the VBA Editor, you will not see any changes in **your code.**

Once the Attribute directive is in place, you can use code like the following:

Dim CC As CMyClassName

Set CC = New CMyClassName

CC.Value = 123

' is the same as

CC = 123

Module 11 – Document Object Model (DOM)

Dealing with events

An Event is an action initiated either by user action or by other VBA code. An Event Procedure is a Sub procedure that you write, according to the specification of the event, that is called automatically by Excel when an event occurs. For example, a Worksheet object has an event named Change. If you have properly programmed the event procedure for the Change event, Excel will automatically call that procedure, always named Worksheet\_Change and always in the code module of the worksheet, whenever the value of any cell on the worksheet is changed by user input or by other VBA code (but not if the change in value is a result of a formula calculation). You can write code in the Worksheet\_Change event procedure to take some action depending on which cell was changed or based upon the newly changed value. (The Worksheet\_Change event might more properly be called Worksheet\_AfterChange since it is called after the cell(s) has been changed. There is no way to access the previous value of the cell before it was changed.) For the Change event, the system will pass you a Range reference named Target that refers to the cell(s) being changed. You can examine that variable to determine if your code needs to carry out some action or whether it can ignore the change and get out with an Exit Sub statement. See the Sample Event Procedure section below.

The events and their procedure declarations are fixed. You must not alter the name or parameter list of an event procedure. The VBA Editor will automatically insert the correct procedure declaration. Do not alter this. You cannot create new events for an Excel object. The events are "hard coded" into the object, such as a Worksheet, and may not be changed. (You can, however, create custom events for your own classes. See the Creating Your Own Events In Your Class Modules section later in this article.

There are many events that are defined in a Worksheet object, the Workbook object, and the Excel Application object itself. On this page, we will cover Application events in only in general terms, but not in much detail since they require a different coding model. (See Application Events for a discussion of Application events.) An event is said to be raised when the action that initiates the event occurs and the application automatically sends a signal to all components connected to event-generating object (e.g, a Worksheet) indicating that the event has occured. An Event Procedure is a VBA Sub procedure that is executed automatically by Excel when the event is raised. It is important to remember that in addition to user input, events may run as the results of actions taken by other VBA code. There is no direct way to determine whether the cause of the event was user input or VBA code. You should write your events procedures such that this distinction does not matter. Since VBA can cause an event procedure to run, it is possible that your code may end up in a loop. See Preventing Event Loops later in this article for information about event loops and how to avoid them with proper coding.

For the Change event in particular, it should be noted that this is triggered when a cell is changed by user action or by other VBA code, but is not raised if the value of a cell is changed as a result of formula calculation.

EVENTS -- On this page, we will be discussing only Excel's events related to Sheets, Workbooks, and the Application, which are completely independent of the events for user forms and controls on user forms. The EnableEvents setting, discussed later, has no effect on events of user forms or controls on user forms. For information about supressing events for controls on a form, see Suppressing Events In UserForms. Events on Charts are a special case of events and need special code handling.

OBJECT MODULES -- Everything related to event handling -- the definition of event procedures, creating a WithEvents variable to receive events, and creating your own events -- takes place within Object Modules. Besides setting the EnableEvents property, there is nothing related to events in a standard code module; every thing is in an object module. An Object Module is any one of the following:

* A Class module.
* A Sheet module (either a worksheet or a chart sheet).
* The ThisWorkbook module
* The code module behind a User Form

Learning What Events Are Available For An Object

You can use the Object Browser in the VBA Editor to determine what events are avaiable for the three objects that generate events -- the Worksheet, the Workbook, and the Application. (See the notes on ChartObjects and Charts in the blue box in the Event Hierarchy section below.) Open the Object Browser in the VBA Editor (press F2 or choose Object Browser from the View menu. In the Classes list, scroll down and select Worksheet. Right-click anywhere in the primary window and choose Group Members on the pop up menu. Then scroll down in the Members Of "Worksheet" list until you see items with yellow lightening bolts next to them. These are the events for the Worksheet objects. Do the same for the Workbook and Application objects. For help on a particular object or event, select it in the Object Browser and press F1 for Help on that topic (note that not all events are documented -- you may have to access event information by going through the object to which the event belongs.

Another method to determine what events are available is to create an empty class module, and enter the following code:

Dim WithEvents App As Application

Dim WithEvents WB As Workbook

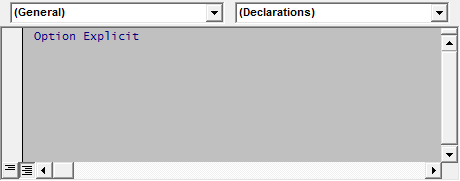
Dim WithEvents WS As Worksheet

Dim WithEvents CHT as Chart

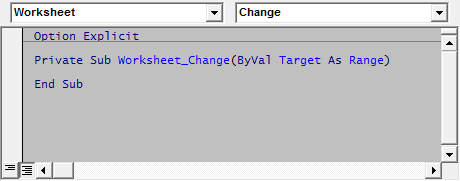
Then, select each of **App**, **WB**, **WS**, and **CHT** elements in the left side dropdown at the top of the code window. All the events for the selected item in the left dropdown will be listed in the right dropdown. If you see an item of interest, let go of the mouse button and the VBA editor will insert that event's procedure declaration in the code module. The declaration will tell you what the parameters for the event are, but you will still need to use Help to obtain a description of the meaning and usage of the parameters. Event procedures must be declared exactly as they are defined. This is why it is good practice to let the VBA Editor insert your procedure shell. Do not change any of the VBA generated code

Getting Started With Events

The easiest way to start with events and event procedures is to allow the VBA editor to build the shell code for you. In Excel, right click one of the sheet tabs at the bottom of the main window and choose View Code from the pop-up menu. This will open the VBA Editor to the code module associated with that worksheet. In that code window you will see two dropdown boxes at the top of the code window, as shown below:



Change the (General) setting to Worksheet and then change SeletionChange to Change. This will add the event procedure declaration for the Change event to the code module, as shown below:



Within the **Worksheet\_Change** procedure, you can add any code that you want to take place when a cell value is changed. This event is raised automatically by Excel when a cell's value is changed either by user input or by other VBA code. It is not raised if the value of the cell is changed by a formula calculation in that cell. The **Target** parameter is a Range type object referring to the cell(s) that were changed. To use other events of the worksheet, select the event in the right hand dropdown at the top of the code pane. When you change this setting, VBA will insert the procedure shell in the code module for you, ensuring that the procedure is properly declared. Under no circumstances should you change anything in the **Sub** statement created by VBA. Doing so may cause the event not to work.

Event Code Location

For sheet (both worksheet and chart sheet) level events, the event procedure code must be placed in the Sheet module associated with that sheet. Workbook level events must be placed in the ThisWorkbook code module. If an event procedure is not in the proper module, VBA will not be able to find it and the event code will not be executed. It is generally accepted good programming practice that only event procedures be included in the sheet modules and the ThisWorkbook modules. If you have no compelling reason to put other code in the sheet or ThisWorkbook modules (and there are a few quite legitimate reasons to do so, but they are beyond the scope of this article) you should put it in a standard code module.

There is no built in object to catch Application events. You can do either of two things, as described below, in the Application Events section of this page: use the ThisWorkbook code module or use a dedicated class module. Placing the code in the ThisWorkbook module requires slightly less code, but I prefer to use a dedicated class module for organizational purposes -- one module for each logical function group. Neither method is particularly better than the other. Use the method that is easiest for you.

Event Hierarchy

As noted above, events are generated by:

* The Application
* The Workbook
* The Worksheets
* Charts

If a Chart is a Chart Sheet, then it follows the rules of a Worksheet, except that its events are no replicated by the **Workbook** or **Application** objects. If a Chart is part of a ChartObject embedded on a worksheet, it follows its own rules. See the Charts and ChartObjects subsection, in blue, later in the article.

An object contains events for itself as well as replications of events for its subordinate objects. Since the **Worksheet** is at the bottom of the hierarchy and has no subordinate objects (at least no objects that have events, that have events), so the Worksheet contains only events for itself. For example, each worksheet has an event named **Worksheet\_Change** that is triggered when a cell on that worksheet is changed either by user input or by VBA (but not if the change is the result of a calculation). Each worksheet's Worksheet\_Change event is exclusive to that sheet.

SUBORDINATE AND SUPERIOR OBJECTS -- In the article, we will use the term Subordinate object to refer to an object below some other object in the hierarchy of event-generating objects. The term Superior object refers to an object that is higher up in the hierarchy of event-generating objects. For example, Worksheet is a subordinate object, to both the Workbook and Application object. The Workbook is both a subordinate and superior object; it is a superior object to the Worksheet object and is a subordinate object to the Application object. Though Charts (either Chart Sheets or Chart objects in ChartObject objects on a worksheet) do raise events, they don't fit into the hierarchy. As far as the event generation object model is concerned, Charts are orphans. See the CHARTS AND CHARTOBJECTS notes later in this section.

The Workbook object is higher up in the hierarchy. Worksheets are subordinate to the workbook. Therefore, the Workbook object has events for itself, such as **BeforeSave** as well has versions of all the events of the Worksheet class. For example, every worksheet has a **Worksheet\_Change** event that is called in response to changes on that worksheet. The Workbook object also has a **Workbook\_SheetChange** event that is called when any cell on any worksheet is changed. When a cell value is changed, both the worksheet's Worksheet\_Change and the workbook's Workbook\_SheetChange events are triggered, and a reference to the changed cell(s) is passed to event procedure.

Since the Application object sits at the top of the hierarchy, it contains events for itself, such as **App\_NewWorkbook** as well as events for all Workbook events and all Worksheet events. Since every event "rolls up" to the Application object, it would be possible to write all the event code within the structure of Application Events. However, this would be very cumbersome and would not take advantage of the modularization that separates event drivers (Application, Workbook, and Worksheet) provide. The code would get very complicated very quickly

.

CHARTS AND CHARTOBJECTS -- Charts do have events (although ChartObjects on a worksheet do not), but they do not fit nicely into the regular hierarchy of the event-generating objects. If you have a Chart Sheet (as oppsosed to a Chart in a ChartObject residing on a Worksheet), the chart sheet acts much the same way as a worksheet with respect to events, albeit with a different set of events. Moreover, these events do not have counterparts in the Workbook object or the Application object. Charts are kind of orphans in the grand scheme of events. A Chart object that is part of a ChartObject on a worksheet also has events, but like the Application object, there is no ready-made container for events of Charts that are part of a ChartObject on a sheet. Instead, you must use either of the techniques described later for the Application object -- just substitute "As Chart" for "As Application" and set the event class variable to **Sheet1.ChartObjects("MyChart").Chart**. ChartObjects do not have events -- it is the Chart object within the ChartObject object that has the events. Events for Charts, either Chart Sheets or Charts in embedded ChartObject do not have their events replicated in either the Workbook or the Application objects. Charts are sort of the "oddball" object of Excel's event system.

The following code may be placed in the **ThisWorkbook** object module to access events of a **Chart** in an embedded **ChartObject** object.

Public WithEvents CHT As Chart

Private Sub Workbook\_Open()

Set CHT = Worksheets(1).ChartObjects(1).Chart

End Sub

Private Sub CHT\_Activate()

MsgBox "CHT: TypeName: " & TypeName(CHT) & vbCrLf & \_

"CHT Name: '" & CHT.Name & "'" & vbCrLf & \_

"CHT Parent TypeName: " & TypeName(CHT.Parent) & vbCrLf & \_

"CHT Parent Name: " & CHT.Parent.Name

End Sub

Order Of Events

If you have event code in the sheet, the workbook, and the application classes, the event will be raised in all three of these objects. Even if a change is trapped by a sheet level **Worksheet\_Change** event, the event procedure in the Workbook and the Application will also be raised. The order of events is from the least significant object (the Sheet) upwards through the most significant object (the Application). You can stop the event from being triggered "upstream" (e.g., preventing the **Workbook\_SheetChange** and the **App\_SheetChange** event from being raised) by setting the **Appplication.EnableEvents** property to **False**. For example, in a sheet's code module:

Private Sub Worksheet\_Change(ByVal Target As Range)

Application.EnableEvents = False

' appropriate action here

Application.EnableEvents = True

End Sub

This code processes the cell change event at the Sheet level, but the line **Application.EnabledEvents = False** prevents the Worksheet and Applicaton SheetChange events from being raised. Indeed, this line of code suppresses *all* events from being raised until its value is reset to True. Note that Excel never automatically sets Application.EnabledEvents back to True (as it does do with the ScreenUpdating property). It is up to your code, including well designed error handling code, to ensure that Application.EnabledEvents is properly reset to True.

Sample Event Procedure

This section will examine a very simple event, the **Worksheet\_Change** event, and illustrate a few useful techniques you can use to determine whether your code needs to act in response to the event. The basic event code, as generated by VBA is as follows:

Private Sub Worksheet\_Change(ByVal Target As Range)

End Sub

As written, the event procedure does nothing -- there is no code within the procedure. The **Target** parameter is a Range object that refers to the cell(s) that were changed. Since **Worksheet\_Change** runs for every cell in the worksheet, you most likely will need to test whether Target is within some predefined range of interest. If it is that range, you'll do something. If Target is not in that range, you don't want to carry out the action of the procedure. The easiest way to do this is with the **Intersect** method. Intersect returns a range of cells that are included in two ranges. For example, the Intersection of the range A1:C3 and C3:F6 is the cell C3 since that cell is common to both ranges. If there are no cells in common between two ranges, Intersect returns **Nothing**. Thus, you can use intersect to see if Target is within the range of interest:

If Not Application.Intersect(Target, Me.Range("A1:C10")) Is Nothing Then

' At least one cell of Target is within the range A1:C10.

' Carry out some action.

Else

' No cell of Target in in the range A1:C10. Get Out.

Exit Sub

You could also use named ranges rather than hard coded cell references. This is the preferred approach.

There may be times you want to act only if a single cell was changed, and ignore it if multiple cells are changed. In that case, you can use

If Target.Cells.Count > 1 Then

' More than one cell. Get Out.

Exit Sub

Here, if Target contains more than one cell, get out of the procedure. In a similar fashion you can test whether Target is within a specified column or row or range of columns and rows. Any of the following code should get you started:

If Target.Cells.Count > 1 Then

Exit Sub

End If

If Target.Columns >= 3 And Target.Columns <= 10 Then

' do something

Else

Exit Sub

End If

If Target.Row >= 5 And Target.Row <= 10 Then

' do something

Else

Exit Sub

End If

Since Target is a Range object, you can perform any number of tests using the vast flexibility of a Range object to determine whether your code should act on the change or simply ignore it by calling **Exit Sub**.

Declaring WithEvents Variables

It is possible that you need to respond to the events of a specific worksheet differently than you would for other worksheets, and that the name of the special worksheet is not known until run time (e.g., it might be a sheet added by your application). You could handle this in the **Workbook\_SheetChange** event, but it would require cumbersome logic to process events for only one worksheet rather than all worksheets. A better and more elegant solution is to create a special class module and within that module declare a variable of type Worksheet using the WithEvents keyword. For example, suppose you want to handle events for a worksheet that is created at run time. In a standard code module, declare a Collection object as:

Public WSColl As Collection

Next, create a class named CWorksheetObject and insert the following code:

Public WithEvents WS As Worksheet

Private Sub WS\_Change(ByVal Target As Range)

Debug.Print "Special Code For New Worksheet"

End Sub

This code declares a variable named **WS** of type **Worksheet** using the **WithEvents** keyword. WithEvents connects the event system to the WS variable and lets you utilize the events of the object. Next, you would include the event procedures for this worksheet:

Private Sub WS\_Change(ByVal Target As Range)

Debug.Print "Special Code For New Worksheet"

End Sub

'

' more event procedures here

'

Finally, you create an instance of the **CWorksheetObject** class and set its **WS** variable to the newly created worksheet. Once the WS variable is set to a specific worksheet, the event procedures in that class will run for events on the assigned worksheet.

Sub TestProc()

Dim WSObj As CWorksheetObject

Dim WSheet As Worksheet

If WSColl Is Nothing Then

Set WSColl = New Collection

End If

Set WSObj = New CWorksheetObject

Set WSheet = Worksheets.Add()

WSheet.Name = "Some Name"

Set WSObj.WS = WSheet

WSColl.Add Item:=WSObj, key:=WSheet.Name

'

' Additional Event Procedures follow.

'

End Sub

The **TestProc** procedure first declares a variable named **WSObj** of type **CWorksheetObject**. At this point, the object exists, but its WS Worksheet object has not yet been set to any specific workbook, so no events will fire in the class. The code then creates a new worksheet, names that worksheet, and then sets the WSObj's WS object to the newly created worksheet. Now that the WS object has been set to a specific worksheet, it will respond to events generated by the newly created worksheet. Finally, it stores the **WSObj** variable in the **WSColl** Collection variable so the object is not destroyed when it goes out of scope at the end of the procedure.

Using the method above, and expanding on it to use other object types, you can simplify programming tasks that might otherwise require much more complicated logic.

Application Events

There are two common ways to declare application events (though because VBA is as versatile as it is, there are many other ways to implement Application events). The first is to declare the **App** variable (of type **Application** in the **ThisWorkbook** code module. The second method is to use a dedicated code module.

Application Events In The ThisWorkbook Module

In the **ThisWorkbook** code module, insert the following code:

Public WithEvents App As Application

Private Sub Workbook\_Open()

Set App = Application

End Sub

Then, select **App** in the left side dropdown at the top of the ThisWorkbook code pane and choose in the right side dropdown which of the available events you wish to use. VBA will automatically insert the proper declarations for that event. Remember, never change the code that VBA inserts for you. If you do change it, it is quite likely that the code will not work properly, if at all.

You can then use events for the **App** object such as:

Private Sub App\_NewWorkbook(ByVal Wb As Workbook)

MsgBox "New Workbook: " & Wb.Name

End Sub

Application Events In A Dedicated Class Module

The second approach to creating Application Events is to use a dedicated class module. Insert a class module into your project and name the class module CExcelEvents. In that class module, enter the following code:

Private WithEvents XLApp As Application

Private Sub Class\_Initialize()

Set XLApp = Application

End Sub

Then, change the left side dropdown at the top of the code pane to XLApp and choose an event from the right side dropdown. VBA will automatically insert the proper procedure shell for that event. As before, do not change the code generated by VBA.

You can then define your application event procedures in the class module. For example,

Private Sub XLApp\_NewWorkbook(ByVal Wb As Workbook)

MsgBox "NewWorkbook" & Wb.Name

End Sub

The next step is to create a variable of type CExcelEvents and initialize that variable to a new instance of CExcelEvents. In the ThisWorkbok code module, declare a variable as shown below:

Private ExcelEvents As CExcelEvents

Private Sub Workbook\_Open()

Set ExcelEvents = New CExcelEvents

End Sub

Since the **Class\_Initialize** procedure of the **CExcelEvents** class initializes the **XLApp** variable when the class is created, we do not have to worry about initializing XLApp. Any Application event procedures should be added to the **CExcelEvents** class.

Which Is Better, ThisWorkbook Or CExcelEvents?

Given that there are at least two method for creating an object to receive Application Events, you may be wondering which is better, a separate class module or the **ThisWorkbook** module, Neither is better in any significant way. As a matter of personal preference and coding style, I put my application events in a dedicated class module. In my opinion, this keeps to project better organized. However, beyond that, there is no advantage to use a dedicated class module for Application events. You should use the approach that seems most natural to your own coding style. Once you decide on a method, stick with that method across projects. Don't mix and match.

Preventing Event Loops

Without proper coding, your event procedures can end up in infinite recursive loops. Depending on your version of VBA and Excel, this may result in an non-trappable Out Of Stack Space error or VBA will simply terminate execution when some threshold (approximately 300) number of calls is met. Consider, for example, the following code:

Private Sub Worksheet\_Change(ByVal Target As Range)

Target.Value = Target.Value + 1

End Sub

At first glance, this code may seem perfectly valid. When a cell is changed to some value by the user, the code adds one the that value, so if a user enters 1, the code will change that to 2. However, this is not what will actually happen. When the user changes the cell to 1, the event procedure runs and changes the value to 2. This change, however, raises the **Change** event again and the code will run to change the 2 to a 3. This again raises the Change event, which changes the value 3 to 4. Yet again, the Change event runs, changing the 4 to a 5. This looping will continue until VBA aborts the loop or you run out of stack space.

In order to prevent this runaway looping, you can use the **EnableEvents** property of the **Application** object. When you set this property to False VBA will not raise any events, and the example **Change** event will run once only for the input by the user. It will not run when the value is changed by the VBA code. You should always be sure to set EnableEvents property back to True to enable events to be called normally. Unlike some properties (such as ScreenUpdating), Excel will not automatically change EnableEvents back to True. Your code must ensure that the value is properly reset. For example, in the code that follows, the **Target** value is incremented once, but since EnableEvents value is False, no subsequent Change event is raised.

Private Sub Worksheet\_Change(ByVal Target As Range)

Application.EnableEvents = False

Target.Value = Target.Value + 1

Application.EnableEvents = True

End Sub

In some circumstances, it may not be desirable to disable all event handling using **Application.EnableEvents = False**. Your application may rely on various events running when they should. You can work around this by creating a public Boolean variable, testing that variables in your event procedure, and exiting the procedure if that variable is True. This way, you can turn off one event handler while leaving the other event handling in place. For example, in a standard code module, declare a variable such as:

Public AbortChangeEvent As Boolean

Then, in the **Worksheet\_Change** event procedure, you test this variable. If it is true, you would immediately exit the procedure, as shown in the example below.

Private Sub Worksheet\_Change(ByVal Target As Range)

If AbortChangeEvent = True Then

Exit Sub

End If

' rest of code here

End Sub

Finally, you would disable the **Worksheet\_Change** event by setting the **AbortChangeEvent** variable to **True**. For example,

AbortChangeEvent = True

Range("A1").Value = 1234

AbortChangeEvent = False

The code above disables only the Worksheet\_Change event and only for the one line code. In general, using Application.EnableEvents = False is sufficient, but there may be circumstances in which more complex event handling is necessary.

Multiple Events Being Called

Because the event object model includes implementations of the events of subordinate objects (e.g., **Application** has events for the **Workbook** and the **Worksheet** objects), you may find that some results are different than what you may expect.

EVENT ORDER: It is important to note that the event procedures of a subordinate object (e.g., the Worksheet will run to completion before an event of a superior object (e.g., Workbook) is called. That is, the Worksheet event procedure Worksheet\_Change will run to conclusion, to the End Sub statement, before the Workbook event procedure Workbook\_SheetChange occurs. Thus, you cannot assume that the Workbook and/or Application SheetChange have been executed within your code in the Worksheet\_Change event procedure. You should assume the opposite -- the events of the superior object will not yet have run.

For example, create a class named CExcelEvents and insert the following code:

Public WithEvents App As Application

Private Sub App\_SheetChange(ByVal Sh As Object, ByVal Target As Range)

Counter=Counter + 1

Debug.Print "Counter: " & CStr(Counter)

End Sub

Then, put the following code in the ThisWorkbook module:

Private Sub Workbook\_SheetChange(ByVal Sh As Object, ByVal Target As Range)

Counter = Counter + 1

Debug.Print "Counter: " & CStr(Counter)

End Sub

Next, put the following code in the code module for Sheet1:

Private Sub Worksheet\_Change(ByVal Target As Range)

Counter = Counter + 1

Debug.Print "Counter: ", CStr(Counter)

End Sub

Finally, put the following code in a standard code module like Module1:

Public Counter As Long

Public XLApp As CExcelEvents

Sub AAA()

Set XLApp = New CExcelEvents

Counter = 0

End Sub

Sub BBB()

Debug.Print "\*\*\* COUNTER: ", CStr(Counter)

End Sub

Now run the procedure AAA to get things set up. Next, type something into any cell on Sheet1. Finally run the procedure BBB. Open the Immediate Window if it is not already open (CTRL G) and look at the messages. You should see:

Counter: 1

Counter: 2

Counter: 3

\*\*\* COUNTER: 3

The counter is incremented first by the event procedure Worksheet\_Change in the Worksheet module, then incremented again in the Workbook\_SheetChange procedure in the ThisWorkbook code module, and then incremented yet again by the App\_SheetChange event procedure. Thus, the counter gets a value of 3, not 1 as you might expect. If you are using events in a superior object, you must take care that the events in the superior object do not duplicate work done in the subordinate procedure. Not properly managing events in subordinate and superior objects can cause unexpected results, such as the result of 3 in the example above.

Creating Your Own Events In Class Modules

If you are writing your own class modules (see Introduction To Classes for an introduction to working with and creating classes), you may want a class to raise an event when a specified action or condition occurs. This is a fairly simple process. Note that only object modules (class modules, userform code modules, the sheet modules, and the ThisWorkbook code module) may declare events. You cannot define, raise, or receive events in standard code modules. Also, only object modules may declare variable WithEvents and therefore only object modules may receive event triggers.

In the class that will create the events, you must first declare the events themselves with the Public Event statement, such as:

Public Event EventName(Parameters, ByRef Cancel As Boolean)

Here, EventName is the name of the event. This should be a meaningful name. Parameters is a list of any parameters that you want to pass to the class that is receiving this event, such as

X As Long, Y As Double, Z As String

This is optional. Finally, Cancel is an optional but useful parameter. It allows the class that is receiving the event to respond back to the class that contains the event that the action in question should be cancelled. For example, the Workbook\_BeforeSave event has a Cancel parameter that allows you to cancel the Save operation. While a Cancel parameter is entirely optional, it is a nice touch that can add flexibility and a professional touch to your application.

Once you have declared your events, (one Public Event declaration for each event), you need raise the event at the appropriate location in your class. Where you raise the events depends entirely on the context of the executing code and what action or condition the event signifies. When I design commercial software, I use events extensively, using both a BeforeSomeAction and AfterSomeAction event pair to notify any listener that my code is about to carry out some action and to notify the listener than the action has been completed. If possible, I like to include a Cancel parameter to allow the event listener to cancel a pending operation. However, you can use events in any way you want.

You raise an event using the RaiseEvent statement. Once the event is declared, you trigger it with the RaiseEvent statement. Both declaration and raising of the event EventName are shown in the code below. Note that you cannot use Named Arguments when passing parameters to the RaiseEvent procedure.

Public Event EventName(IDNumber As Long, ByRef Cancel As Boolean)

Sub AAA()

Dim B As Boolean

Dim IDNumber As Long

IDNumber = 1234

Cancel = False

RaiseEvent EventName(IDNumber, Cancel)

If Cancel = False Then

' Appropriate Non-Cancel Action

Else

' Appropriate Cancel Action

End If

End Sub

Once you have created your class with events, you need to write the code that will receive the event triggers. Note that only object modules (class modules, a user form code module, a Sheet module, or the ThisWorkbook module -- standard code modules cannot receive events) can receive event messages. In a suitable object module, declare the event class using WithEvents:

Dim WithEvents XLEvents As CExcelEvents

' More code

At some point in your code, you will need to set the XLEvents variable to an instance of the CExcelEvents class, with the code:

Set XLEvents = New CExcelEvents

Exactly when and where you put the object initialization code depends on what sort of module contains the event declaration. While it is technically possible to put the initialization of the variable in another procedure, this is generally a bad programming practice: it makes the code more difficult to debug and maintain. As a general rule, the code that initializes the events variable should be in the same class as the events variable. Of course, the actual event code must reside in the same object module as the events variable declaration. In a class module, the initialization would normally be in the Class\_Initialize event. For a user form, the code would go in the UserForm\_Initialize event.

Private Sub XLEvents\_EventName(IDNumber Long, Cancel As Boolean)

Cancel = True ' or False -- your choice

End Sub

Event Parameters

Use Event Parameters to cancel an event

**Private Sub**Worksheet\_BeforeRightClick(ByVal Target **As**Range, Cancel **As**Boolean)  
    Cancel = True  
**End Sub**

Module 12 – Advanced Topics

Using Declare keyword to extend VBA ability using external DLL

Advanced Techniques

Security Concerns

There are two main aspects to security

1. Potential loss of intellectual property if people access source code without permission.
2. Potential danger from a malicious user subverting the expected behaviour for their own (possibly disastrous) purposes.

VBA security has for years revolved around security through obscurity. Any security revolved around the fact that the underlying file format was not officially documented. The VBA is not encrypted or scrambled, or otherwise protected. There is simply a flag in the file that says should Excel ask for a password before displaying the VBA. That means other file readers apart form Excel are free to respect or ignore that setting. After that the actual code is pretty much a bunch of embedded text files.

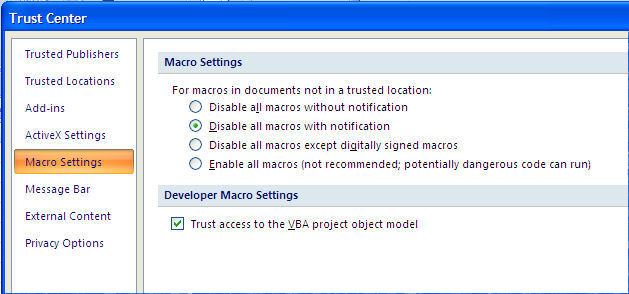
In recent times VBA file structure documentation has been published to assist interoperability, so it is even easier now for people to inspect the internals of workbooks and add-ins.

VBA security is fairly poor, the code is not compiled, and the source is available in the excel file. The password protection is pretty easy to circumvent. This reflects Excel/VBA target as an end user application. Code security can be enhanced by moving to [COM components](http://www.codematic.net/excel-development/excel-add-ins/excel-add-ins.htm) orcompiled code such as C, if there is a real need.

Our experience is that although initially clients often seek the additional intellectual property protection of something other than VBA, once they realise the cost in complexity, schedule and finance, they frequently accept VBA as the best compromise. We are always happy to quote to covert VBA system to other technologies.

The danger of having code subverted is real enough, although we don't often hear of actual incidents. The simplest solution is to sign theVBA project, any subsequent modification will then trigger a warning. Sadly code signing is still not that common so unsigned code does not worry users. Lucky then that most VBA is maintained within the organisations firewall.

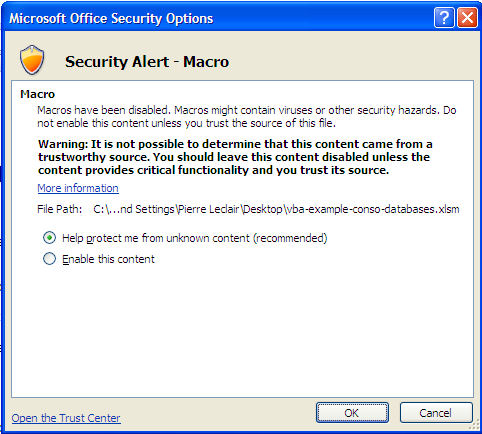
If you send a workbook with macros to a colleague and he can not get them to work it is probably because his security setting is at "High" . Tell him how to change his level of security by going to the "Developer" ribbon, clicking on "Macro Security", selecting "Macro Settings and checkink the second level "Disable all Macros with Notification" and you are set.



From then on each time you open a workbook that contains macros a temporary status bar appears above the grid in Excel:

http://www.excel-vba.com/zzz-macro-enable-2007.jpg

Click on "Options" and the following dialog window will appear.



Adopt the same attitude as you have with documents attached to Emails. If you know the origin of the file you may enable the macros if not click on "Disable Macros" and you are fully protected. You can look at the workbook but the VBA procedures (macros) are not operational. You can go to the Visual Basic Editor to take a look at the macros. If nothing looks suspicious close the workbook and re-open it enabling the macros.

## Password Protecting the code

As an Excel-VBA Developer you might want to protect your code so that nobody else may modify it. In the VBE editor go to "Tools/VBAProject Properties/Protection" . Check the box and submit a password. Make sure that you save the password somewhere that you will remember. If ever you loose the password for an important workbook you can always buy a program on the Internet that will allow you to view the code even if it is password protected.

Remember that passwords are like any locks, they only keep the honest people out.

Common Pitfalls

|  |  |
| --- | --- |
|  | A common coding oversight I see is declaring variables as follows:  Dim a, b, c as Integer  This only sets the variable c as an Integer. Variables a and b will be of type Variant. This can be validated by using VarType() in VBA.  Instead, the correct variable declaration (assuming a,b and c should be integers) should read:  Dim a as Integer, b as Integer, c as Integer |

VBA does not short circuit logical AND. For example:

Function TestShortCircuit(a As String)   
   
    If IsNumeric(a) And CInt(a) = 0 Then   
        Debug.Print "a converted to integer will be 0"   
    Else   
        Debug.Print "a is not a number or non-zero when converted to integer"   
    End If   
   
End Function

Try it in immediate window:

? TestShortCircuit("0.25")   
a converted to integer will be 0   
? TestShortCircuit("1.25")   
a is not a number or non-zero when converted to integer   
? TestShortCircuit("agilevic")

In the last call a type mismatch gets thrown because CInt gets called on a string, which cannot be interpreted as number. Most languages stop evaluating logical conjunction as soon as at least one condition evaluates to false. In VBA you have to rewrite the code as follows:

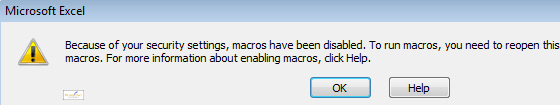
Function TestShortCircuit(a As String)   
   
    If IsNumeric(a) Then   
        If CInt(a) = 0 Then   
            Debug.Print "a converted to integer will be 0"   
        End If   
    Else   
        Debug.Print "a is not a number or non-zero when converted to integer"   
    End If   
   
End Function

# Problems When Running VBA Code

Lots of things can go wrong when you try to run a subroutine, and not all of them are down to coding errors.  This article explains a few of the common mistakes people make when running VBA code and how to get around them.

## Macros are Disabled

You might think that this is a silly mistake to make, but it's surprisingly common, especially on training courses!  All of the Microsoft Office applications have a number of macro security levels and the level that is set determines when macros are allowed to run.  If you try to run a macro when they are not enabled, you'll see a message like this:



*This is an example of the type of message you'll see when macros are disabled but you try to run one anyway.*

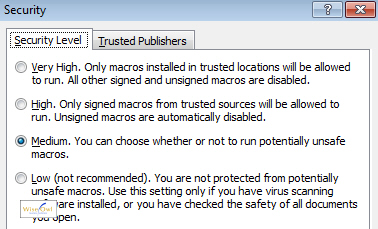
Making sure that macros are enabled works slightly differently depending on which version of Office you are using, but in essence there are two steps involved:

1. Choosing an appropriate level of security.
2. Choosing to enable macros each time you open a file that contains them.

## Changing the Macro Security Level in Office 2003

To choose the appropriate level of security in Office 2003 applications:

1. From the menu, choose: **Tools -> Macro -> Security...**
2. On the dialog box that appears, select **Medium**, and then click OK.

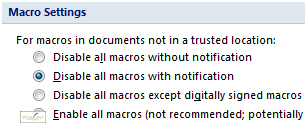


***Medium*** *is probably the most sensible option to choose. A* ***Low*** *security level means that your code will always be allowed to run, but so will everyone else's, including potentially dangerous code.*

## Changing the Macro Security Level in Office 2007 and 2010

To choose the appropriate level of security in Office 2007 or 2010 applications:

1. From the Ribbon, select: **Developer -> Macro Security**.  If you can't see the Developer tab, click [here](http://www.wiseowl.co.uk/blog/s137/recording-vba-macros-2.htm) to see how to enable it in Office 2007, or [here](http://www.wiseowl.co.uk/blog/s138/developer-rbbon-excel-2010.htm) to see how to enable it in Office 2010.
2. On the dialog box that appears, choose the option shown in the diagram below.

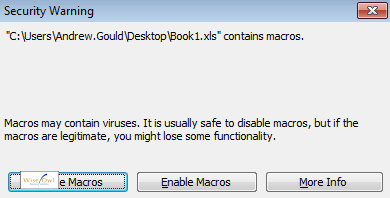


*The selected option in this diagram is the equivalent of the* ***Medium*** *security level in Office 2003.*

Once you have done this, close down the application and then reopen it.  The next time you open a file that contains macros you must choose to enable them.  This works differently in the different versions of Microsoft Office.

## Enabling Macros in Office 2003

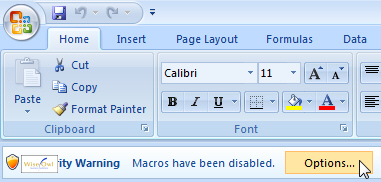
In Office 2003 applications when you open a file containing macros you will see a dialog box like this one:



*Click* ***Enable Macros*** *to make sure your code will run.*

## Enabling Macros in Office 2007

In Office 2007 you will see a small message appear below the Ribbon when you open a file containing macros:



*Click the* ***Options...*** *button to choose to enable macros.*

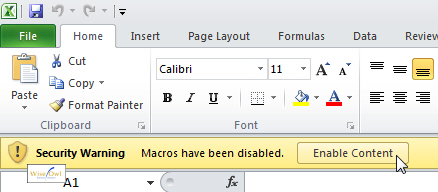
On the dialog box that appears you can then choose to enable macros:



*Choose this option and then click OK to make sure your code can run.*

## Enabling Macros in Office 2010

In Office 2010 you will see a message appear under the Ribbon - simply click the Enable Content button to make sure your macros can run.

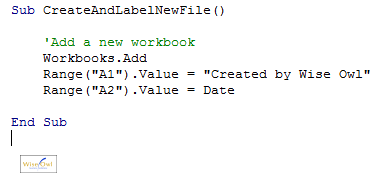


*Click the button shown here to make sure you can run your subroutines.*

When you are sure that macros are enabled there are still plenty of other potential sources of error!

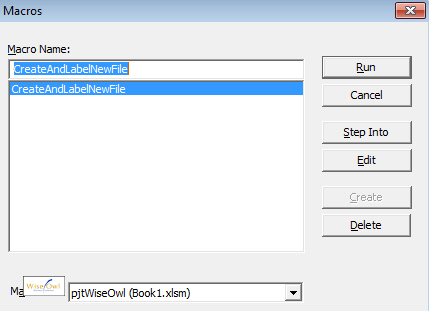
## Not Selecting the Correct Subroutine

If you are running your code from within the VB Editor, you have to select the subroutine you want to run before you try to execute it.



*If the text cursor isn't within the subroutine you want to run, the VB Editor won't know which one to run.*

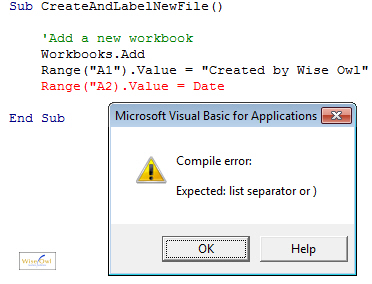
If you don't have a subroutine selected at all you will see a dialog box appear to ask you which one you want to run.



*You could always select the macro at this point and click Run, but it's much easier to just click in the macro you want before you try to run it!*

## Syntax Errors

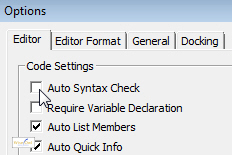
Syntax errors are mistakes in the punctuation of your VBA sentences.  These are very easy to spot because as soon as you make a syntax error and try to move the cursor to a different line of code, the offending line will be highlighted in red.



*With the default settings in the VB Editor, you will also see a dialog box attempting to explain the problem.*

To solve this issue, click OK on the message that appears and then try to amend the line of code that is flagged in red.  Here the mistake we've made is to miss the double-quotes after the cell reference **A2**.  When you have edited the line of code, move the cursor to a different line to check that it doesn't get highlighted in red again.

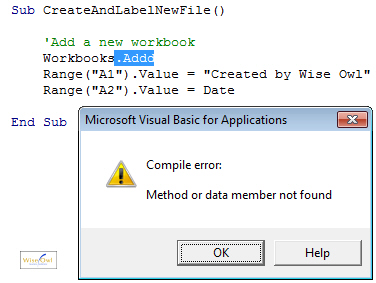
*The dialog box that pops up to warn you about syntax errors is never particularly useful, and it's quite annoying to have to click OK before you can fix the problem.  You can turn off this warning message (but still have syntax errors highlighted in red) by selecting:* ***Tools -> Options...*** *and then completing the dialog box that appears as shown below:*



*Make sure the* ***Auto Syntax Check*** *box is unchecked and then click OK to make sure you don't see the annoying pop-ups every time you make a mistake.*

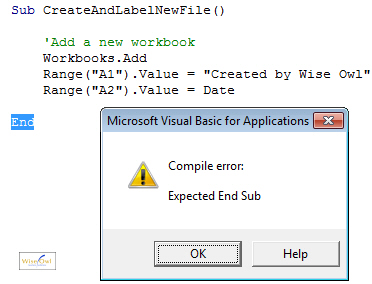
## Compile Errors

When you choose to run a program that you've written in VBA, the VB Editor first of all compiles your code to make sure that all the words make sense in the context you've used them.  A compile error is a problem that occurs at the compile stage but before your code has actually started running.



*In this example we've misspelt the name of the* ***Add*** *method. The VB Editor has helpfully highlighted this for us so our job of fixing it is as easy as possible.*

To fix a compile error, click the OK button on the message that appears and try to identify what is wrong with the part of the code that has been highlighted.  Compile error messages are often quite descriptive, like the one shown below:

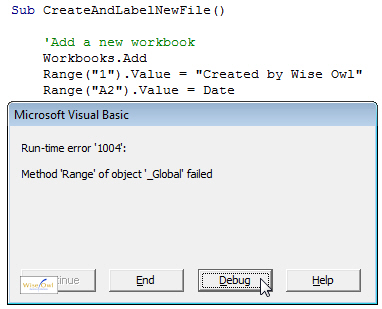


*It's fairly obvious what this error message means, and as if it wasn't obvious enough the offending part of the code is highlighted again.*

*You can ask the VB Editor to compile your project without trying to run a subroutine by selecting:* ***Debug -> Compile ProjectName***

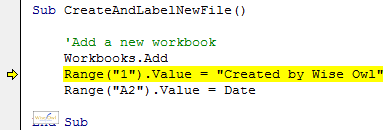
## Run-Time Errors

Run-time errors are errors that occur when your program has successfully passed all of its syntax checking and compiling and is in the process of running.  Any executable line of code has the potential to generate a run-time error and it's not always obvious why one has occurred.  The diagram below shows an example of a run-time error:



*It's not immediately obvious what's gone wrong here as run-time errors don't highlight any of your code immediately. You can click the Debug button to see which line of code has failed to run.*

When you click the Debug button on a run-time error dialog box, the line of code that has failed will be highlighted in yellow.  In the example below you can see that it's the second line of code that has gone wrong.



*Closer inspection should reveal a mistake in the cell reference we've typed in.*

This is just one of many examples of run-time errors that you'll experience when running VBA code.  Run-time errors are often the most difficult to resolve and there's really no substitute for experience when things get tricky.  The best way to learn is to try something, get it wrong, and then fix it!

## What We've Learnt

If you've been following this blog series from the beginning and you'd never even used VBA before, congratulations!  You've come a long way from where you started, but there's still an awful lot to learn before you become a fully fledged programmer.  The key now is to start experimenting with new techniques: try recording some macros to see the code that gets written for you; edit some recorded macros to see what happens; read more Wise Owl blogs to give you more ideas.  Above all, learning to program can be immensely satisfying, so have fun with it!

Pitfall-#1

Dim d as Object   
Set D = CreateObject("SomethingUseful")

instead of just

D = CreateObject("SomethingUseful")

which the IDE would not complain about, but you get a runtime error (object reference not set).

But this is basically VB.Net/VB6 difference, not especially VBA.

Pitfall-#2

Good Practices

Though VBA's error handling uses the antiquated and much-loathed "goto" statement, it's a valuable addition to most subroutines. This is especially true when changing application-level settings.

For example, a common method of speeding up code that affects a large area of cells in Excel is to disable screen updating. This is done with:

Application.ScreenUpdating = False   
' Code goes here   
Application.ScreenUpdating = True

Others might include:

Application.ShowWindowsInTaskbar = False   
Application.EnableEvents = False

These are very helpful and will greatly speed up performance under the right circumstances. The problem arises, however, when the odd error occurs and your application properties never got set back to "true". That's why if I find myself calling ANY application-level properties I reset them to True in the error handler. Otherwise you'll find your application behaving in a pretty peculiar manner.

ExampleSub\_Error:   
   
    MsgBox "Error " & Err.Number & " (" & Err.Description & ") in procedure ExampleSub of     VBA Document Sheet2"   
    Application.EnableEvents = True   
    Application.ScreenUpdating = True   
   
End Sub

### Codematic VBA Code Convention for Excel VBA

(Note this advice is specific to Excel VBA in business applications. Other languages and other applications have very different requirements).

Pretty simple really, easy to remember and easy to implement:

* Always use Option Explicit, usually use Option Private Module.
* Give Module level variables a scope qualifier prefix g\_, p\_ or m\_ (Global (all open workbooks), Public (this workbook), or Module). Scope everything as tightly as possible. Use procedure level scope where possible and pass values by parameters.
* Use meaningful names for all procedures and variables. Do not bother with convoluted data type prefixes, that is inappropriate for business level applications, makes the code harder to read and adds very little of value. The compiler will pick up any obvious data type errors. Used mixed case descriptive names - if the name is too long then probably the procedure is too.
* Procedures should fit on one screen - ie be 40-50 lines long maximum.
* Avoid most comments - make the executable code meaningful and simple instead.
* Avoid magic numbers and strings - use constants.
* Never comment what the code does - that should be crystal clear from the code, comment WHY something is done, especially if it is unusual. Add a couple of sentences to provide an overview of a module or class.
* Pass parameters ByVal (ByRef is the default) - only use ByRef where you intend to modify the parameter and pass the change back to the caller.
* Avoid Application.Run where possible as it breaks the error handling stack.
* Use additional tools. See [links](http://www.codematic.net/excel-resources/excel-links.htm) page for some suggestions.
* Be aware of other options if VBA appears inappropriate for certain aspects of the project.
* Vary any rules you the developer feel do not promote clarity, simplicity and safety.