Programming
in
Microsoft Excel VBA
An
Introduction

Table of Contents

List of Figures

Overview

WHY

Why does this book exist? I wrote this book to hopefully provide a basic introduction to learning to program using Visual Basic for Applications (VBA) as implemented in Microsoft™ Excel©. I have attempted to provide a balance of basic programming concepts and good programming practices. Along the way concepts are presented that often fall into the "advanced" category in other books. I don't believe these concepts are "advanced" in that it takes more basic teaching to understand and use them, rather if they are taught as part of that basic teaching they are no more difficult to learn than anything else in the language.

The goal is not to make you all-knowing of all things VBA in Excel, but rather to try to give you a basis for understanding what VBA for Excel is capable of, helping you put code samples you obtain from a variety of sources to work for you, to learn how to modify and adapt recorded macros to make them more generic and useful to you, and to encourage you to learn more about the language so that you can take full advantage of the worlds #1 spreadsheet application.

WHAT

What is taught in this book? The basic elements of VBA coding are covered and hopefully taught in it. The First Edition will pretty much just cover what I decide to cover. If anyone has specific things that they feel would be beneficial to the budding VBA programmer, I will certainly entertain the idea of including them in later revisions to it. You can send such suggestions via email to: HelpFrom@JLathamsite.com

The difficulty in presenting this type of material is that teaching VBA coding requires knowledge of many things that have inter-dependence on one another. This inter-dependence can be an actual physical dependence, but more often it is a dependence based on the knowledge of many different elements of the programming environment: the syntax or command structure for instructions; a knowledge of the "objects" in the application and their attributes (properties) and the things you can do to or with them (methods), along with many other things. By necessity some things must be taught before others in order to build from a basic understanding to more complex understanding as the studies continue. In discussing some of the basics, more advanced concepts may be used in the process and the reader must accept those as-yet-unexplained concepts and pieces simply on faith or with an "it is what it is" attitude for a while. Since this is an *Introduction* to VBA for Excel, many details of many subjects and areas are left to be discovered by the student on their own through experience, further study and examples from other sources in the future.

WHO

This book is for anyone desiring to learn how to extend the functionality and usefulness of Excel through added capabilities often only available through VBA.

HOW

You will learn to begin programming in Excel VBA by reading through this book and you will use your copy of Excel to 'follow along' and create procedures and observe them at work.

ANTICIPATED USER SKILL REQUIREMENTS

This book is designed to be used by those with the Excel® knowledge typical of the 'average' office user. This means that the user is expected to be familiar with general Excel® features and functions such as the use of menu and icon toolbars, selecting worksheets and cells, creating simple formulas in cells, 'navigating' within Excel®, and opening, saving and closing the Excel® application and Excel® created workbooks (.xls files).

COPYRIGHT ACKNOWLEDGMENTS

CREATING THIS GUIDE

This document was created using Microsoft Word and Excel 2003, along with the Microsoft Office 2007 provided 'publish as .pdf' feature to generate the final document.

Graphic screen capturing was performed using SnagIt from TechSmith.

The most current version of the book may be downloaded, free of charge, from:

http://www.jlathamsite.com/LearningPage.htm

Look for the link to the .pdf document just below the heading **"Introduction to VBA Programming"**. I recommend right-clicking the link and choosing "Save Target As" to get a copy of it onto your system.

What is VBA?

Visual Basic for Applications (VBA) is an extensible programming language that is made up of a core set of commands and extended on a per-application basis to be able to work directly with objects in that application. This means that VBA for Excel knows about things like workbooks, worksheets, cells and charts and more; while VBA for Access knows about tables, queries, reports and data entry forms, among other things. The core can even be licensed for use by 3rd party companies to permit it to be used with their application(s). This was the case with Visio before Microsoft bought the product for use under their banner.

VBA can probably be best described as an object-based (but not a true object oriented) language that is event driven. Let's look at the event driven side of it first.

VBA: An Event Driven Language

Event driven means that nothing happens until something happens. Rather Zen-like isn't it? Ok, once again, but with a better grasp of reality. In VBA, no code executes except in response to some event taking place (or at the command of the code once it is started by some event). An event can be any one of many things. Opening an Excel workbook creates, or *triggers*, the Open event, closing it triggers the BeforeClose event. Selecting a worksheet in the workbook will cause a Deactivate event to occur on the page that had been selected and an Activate event to happen to the new sheet you select. Many events occur that don't have code associated with them, and that's to be expected; something doesn't have to happen every time something else happens. A shape (square, button, text box) actually triggers a Click event when it is clicked on – you may or may not have code associated with one of those shapes to respond when it is clicked on.

Event driven also means that you never know exactly when code for an event will be called upon to run. For example, you may have a process that runs when a particular sheet is selected that takes a long time to complete – perhaps checking for and hiding unused rows, or refreshing the data on the sheet from another data source. While that is going on, you may click a button on the sheet to try to do something else, such as sort the data on the sheet. Excel will, for the most part, deal with the timing of when these processes are performed. You do need to be aware that it is possible to request an operation to begin before another has completed. Most of the time this does not cause any problem at all, but sometimes it can.

VBA: An Object-Based Language

Object based means that when referring to the components of the application, things like workbooks, worksheets, cells, charts, etc. are 'objects'. An object has certain attributes. Just a a person has attributes like height, weight, eye and hair color, the objects in Excel have attributes (Properties) such as value, height, width, color and more. Additionally, objects can do things or have things done to them – these actions are known as Methods. For example, a workbook can be opened or closed; a cell can have its shading altered, a worksheet can be deleted.

While you may use constants and variables in your code that seemingly don't have any direct relationship to an object, in the end the results of processing or calculations will probably be used to alter some property of an object in the workbook: the value in a cell, the range of information used as a data series on a chart, whether or not a particular sheet is visible or not at any given moment. With all of this under our belt, we can now look at how to access this power.

The Excel VBA IDE

The Excel VBA IDE (Integrated Development Environment) has not changed in quite some time. This is a good thing – the interface in Excel 2007 is the same as it was in Excel 2003, 2002 (XP), and even back to Excel 97 , and that means that no time is wasted for programmers in learning a new interface just to be able to continue to use a language they are already familiar with. There are 5 major areas of the IDE and I like to work with all of them visible.

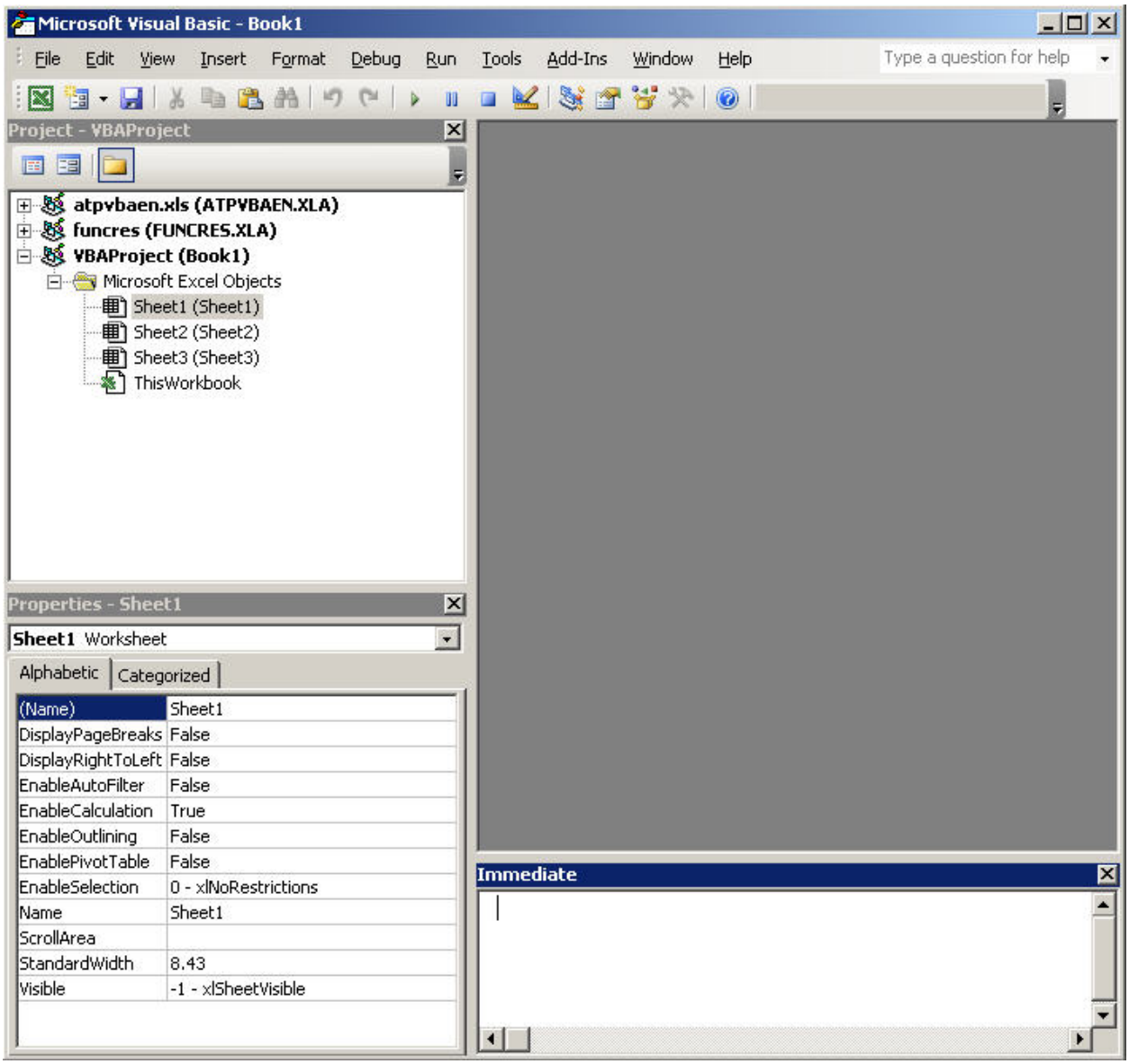


Figure 1 Excel VBA IDE - No Code Module Displayed

GETTING TO THE VBA IDE

Your first question may be "How the heck did you get there!?" The quickest way to open the VBA IDE (which I'll simply call the VBE for Visual Basic Editor for the rest of this document), is to press **[Alt]+[F11]** while in the main/normal Excel window. You can also get there from the normal Excel menu via Tools | Macro | Visual Basic Editor:

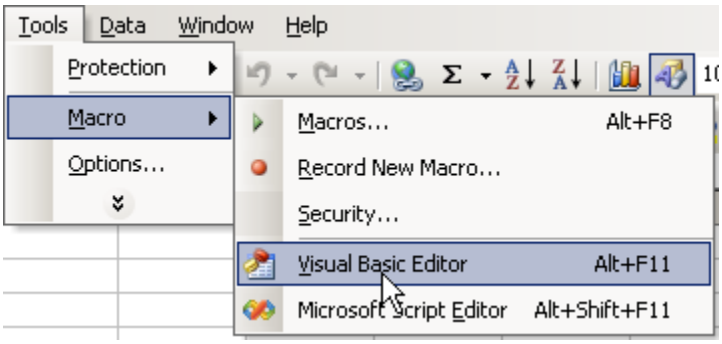


Figure 2 Open the VBE from the Excel Tools Menu

There are also other fast ways to open the VBE to specific areas without first opening the entire project as these two methods do. We will discuss those when we talk about code that deals with Workbook and Worksheet related event processing.

Not all 5 major areas of the VBE may be visible when you first open it. The [View] VBE menu option allows you to choose which of them are visible.

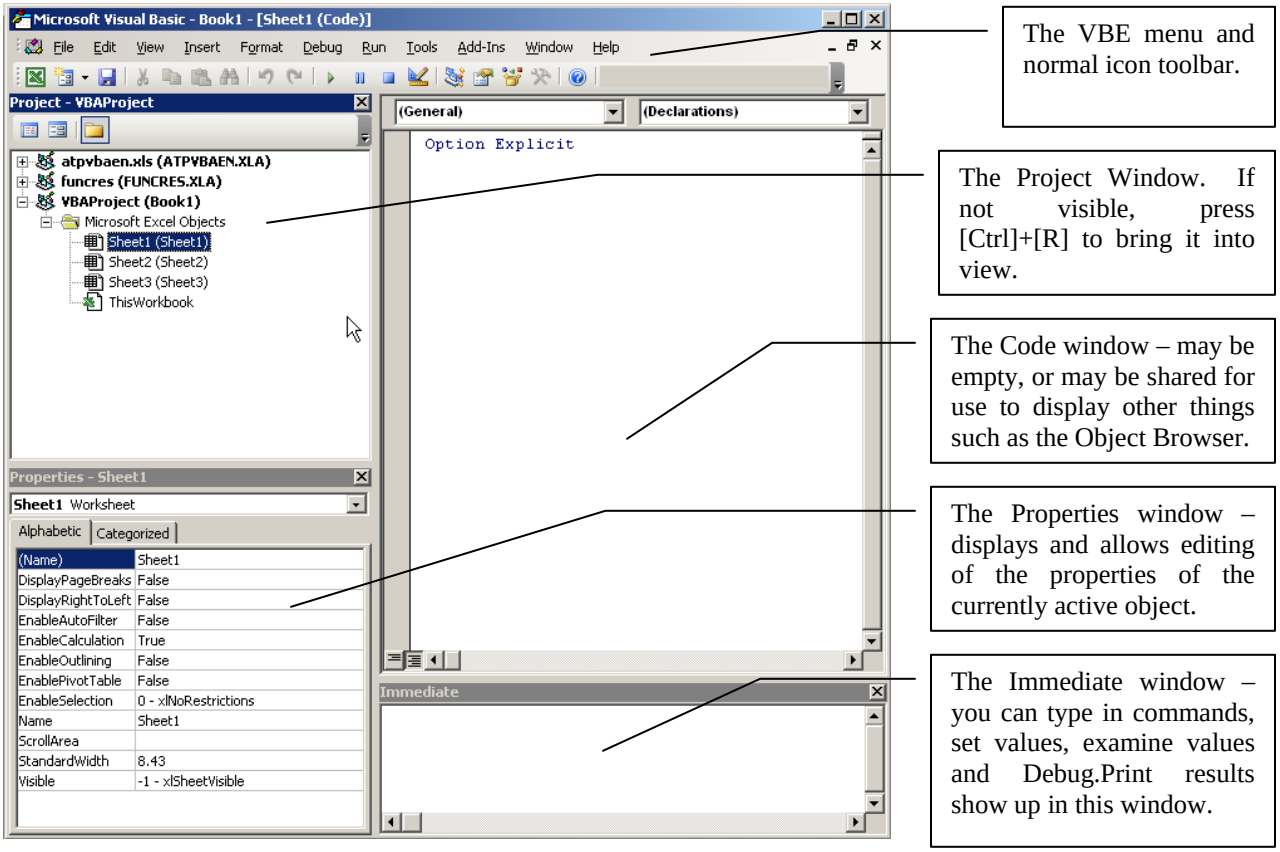
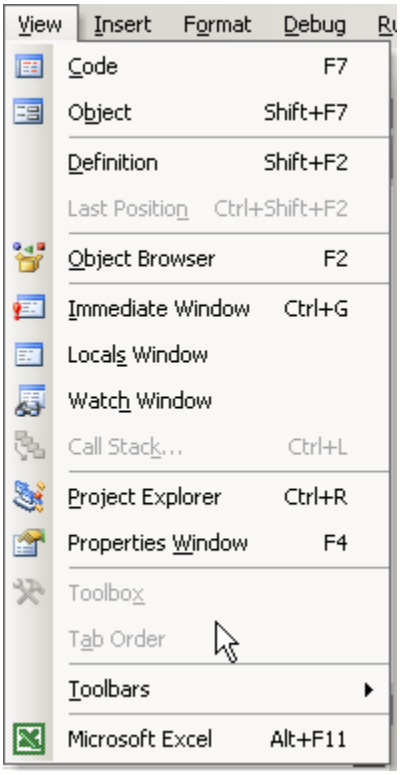


Figure 3 Excel VBA IDE Major Areas



This is the VBE [View] menu option expanded. As you can see, it permits you to display any of the 5 major areas of the IDE and even more that are useful in special circumstances such as the Object Browser and the Locals and Watch windows.

Note: To close any of these windows once you've opened them, simply click the classic "close window" [X] in the upper right corner of the window.

Figure 4 The VBE [View] Menu Item Expanded

Rather than trying to make you remember what each and every window contains, what it's used for and how to make them work for you, we will cover using them during our discussions on actually writing code and accessing objects during code development.

TO BE EXPLICIT OR NOT

Well, let's be frank about this: we are all adults (all programmers are performing an adult task and so, regardless of their physical age, we will give them adult status – and that does mean that they should act as responsible adults, i.e. no intentional malicious coding allowed). Since we are now all adults, we can be Explicit.

Figure 5 Option Explicit in Effect

Your initial view of a code module may not contain the *Option Explicit* statement at the beginning of it. It should – quite simply this is **your first step to responsible coding** through the use of accepted **Best Practices**.

Option Explicit is a directive to the compiler that says that all user defined constants and variables must be declared before actually using them in the code. The up side of using *Option Explicit* is that errors in your code due to typographic errors or reuse of a variable as the wrong type are greatly reduced and when it does happen, the problems are more easily identified. The down-side? Just that you have to take the time to go back and declare constants or variables that you find you need during code development.

To make sure that you don't forget to always use *Option Explicit*, you can tell Excel's VBE to always start new code modules with that statement. This is a 'permanent' setting and affects all projects you create in any workbook after making the setting.

Start by selecting [Tools] | Options from the VBE menu toolbar:

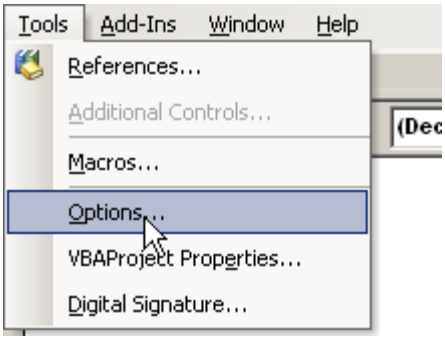
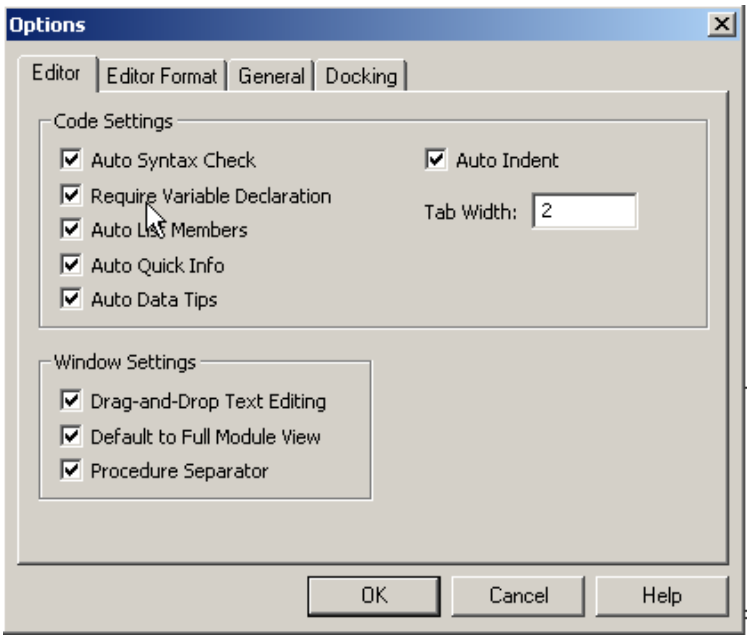


Figure 6 Setting *Option Explicit* Directive: Step 1



This is the dialog that appears once you use [Tools] | Options from the VBE menu toolbar.

Check the "Require Variable Declaration" box to set up the VBE to always place the *Option Explicit* statement at the beginning of all new code modules in the future.

Figure 7 Setting *Option Explicit* Directive: Step 2

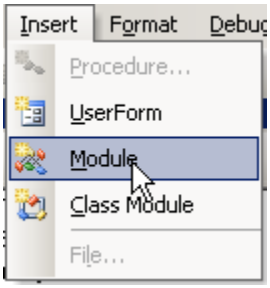
Types of Code Modules

I'll bet you thought that all code modules were created equal. Not true – code modules don't have any Constitutional Rights, although they do have to follow the rules of design requirements imposed by Microsoft and the compiler.

GENERAL PURPOSE CODE MODULES

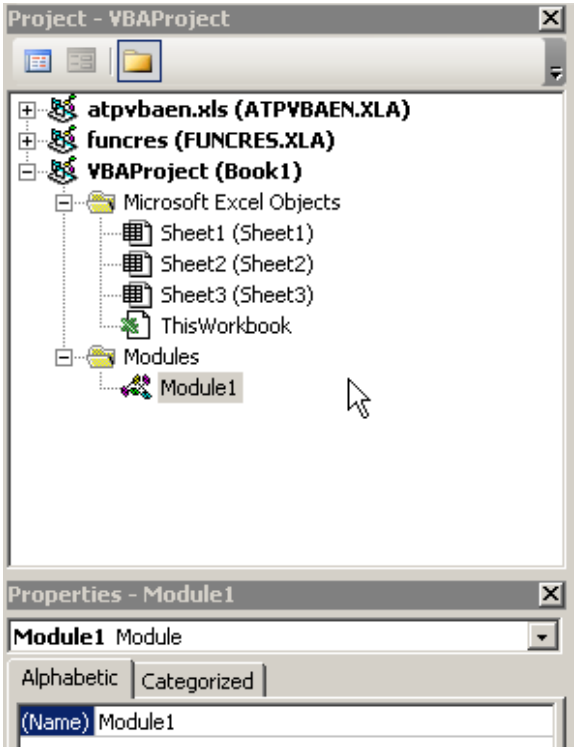
These are code modules you'll bring into existence and can contain code to do almost anything of a 'general purpose' nature. Examples of code that might appear in them would be code that responds to shapes or command buttons you might put on a worksheet; code to respond to custom menus you might develop, user defined functions (UDF) that you develop to perform actions and calculations by way of using the name of the UDF in a worksheet formula just like a built-in Excel worksheet function.

Oh, By the Way Macros you record are placed into general purpose modules. Recording macros during different sessions with the workbook results in numerous modules that may contain as few as a single procedure (macro) in it. This results in being quite wasteful of resources. All macros recorded during a single session are typically placed into a single module.



To create a new general purpose module you can use the [Insert] | *Module* option from the VBE menu toolbar:

Figure 8 Insert a New General Purpose Code Module



After inserting the first new general purpose module, you'll have a new entry in the VBAProject window.

Now you have a new collection called *Modules* and the new module you just created will be listed as one of the members of the collection. Any more modules you add will be listed as new members of the collection. You can double-click on any of them and view its contents in the Code Window.

The one property that a module has is its Name. You can give more meaningful names than just "Module1" or "Module2" by changing the name in the properties window while the module is the current ~~object of affection~~ active object.

Figure 9 VBAProject Showing the Modules Collection

There's no rule for naming modules except that they must start with an alpha character and can't contain certain special characters, I like to give mine names that start with "bas" (for BASIC) followed by some description of the use of the code within them. Examples might be names like:

basUtilities
basDeclarations
basSheet1_Operations

While there is no practical (or published) limit to the number of modules, I'm sure it's at least one of those "limited by available memory" things. The maximum size of any individual module is 64K (to the best of my recollection). Trust me, you can put a LOT of code into a single module.

WORKBOOK CODE MODULES

There is one and only one code module per workbook that is associated with **Workbook Event** handling. At the technical level, this module, along with the worksheet event handling modules are Class Modules. That need not concern you. **Just be aware that if you want to do any coding that deals with events that occur at the workbook level, you do it in this module**.

Workbook Events

Just what are the workbook events? You can get a complete list of them from the code window while the Workbook Code module content is displayed: You can display that content by double-clicking the *ThisWorkbook* object in the VBAProject window. You'll get a display similar to this

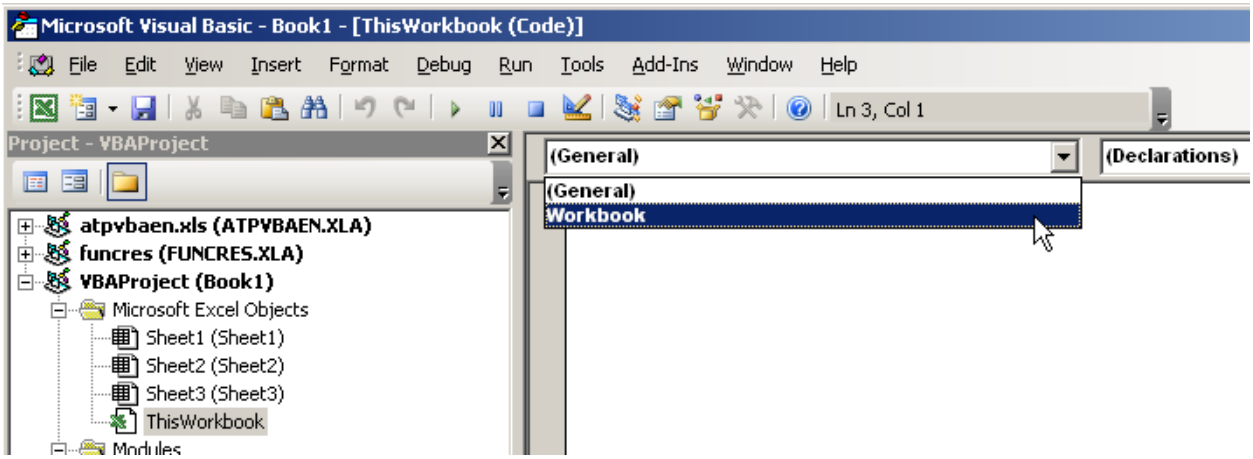


Figure 10 Working in the Workbook Code Module

If you use the left pulldown of the workbook's code module you'll see that there is a specific Workbook entry. If you choose that item, the VBE will automatically insert a stub (just the beginning declaration and end statement for the procedure) for the Workbook_Open() event. You can delete that entry if you don't need code to deal with something you want to happen when the workbook is opened.

With your cursor placed inside of any Workbook related procedure, even just a stub, you can then use the pulldown on the right to find a list of all the available event handlers for the workbook. And it is quite a list.

NOTE: If the cursor is not in a workbook event handling procedure, the list on the right will show you a list of non-workbook event procedure names in the module.

If you write code inside any of the event procedure, then when that event is triggered the code associated with that event will run; i.e., the code will execute. Some typical Workbook associated events that are often provided with code are:

Workbook_Open()
Workbook_Close()
Workbook_BeforeClose()
Workbook_BeforePrint()
Workbook_BeforeSave()
Workbook_Activate()
Workbook_Deactivate

Things you might do with some of these? Well, in the Open() event you might make certain that a particular worksheet is the one selected so that the user sees it first. Or with BeforeSave() or BeforeClose() you might examine certain cells to make sure that all required information had been entered into the workbook and even that it falls within acceptable limits. Activate and/or Deactivate? These are great for determining when to create/destroy custom menus to be used in the workbook but that you don't want available in other workbooks.

WORKSHEET CODE MODULES

There is one and only one code module per worksheet that is associated with **Worksheet Event** handling. However, each sheet has its very own code module that is separate and distinct from all of the others even though they may all have event handlers for a given event for those worksheets. At the technical level, this module, just like the event handling module for the workbook are Class Modules. **Remember that if you want to do any coding that deals with events that occur at the worksheet level, you do it in these modules**.

Worksheet Events

Just what are the worksheet events? You can get a complete list of them from the code window while any Worksheet Code module content is displayed: You can display that content by double-clicking any *worksheet* object in the VBAProject window. The code module for that sheet will be displayed. You'll get a display similar to this

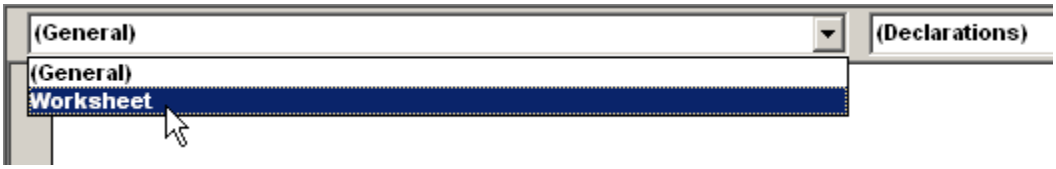


Figure 11 Viewing the Worksheet Event List

For worksheets, when you choose the **Worksheet** item in the left pulldown list, the default event is the Worksheet_SelectionChange(ByVal Target As Range) event. This even triggers any time you make a new selection on the sheet – such as simply moving to another cell. The new cell becomes the selection, and thus you've had a selection change.

As with the Workbook events, you can now get a complete list of Worksheet Events available to be programmed against by using the right-side pulldown (indicated by "(Declarations)" in the graphic). This list is much shorter than the Workbook's list, but even these 9 (from Excel 2003) provide considerable versatility in dealing with worksheets. Out of the list, the Change() event is probably the one that most often has code associated with it. A Change() occurs when a user alters the contents (value) of one or more cells on the sheet. Worksheet formula recalculations don't trigger this event, but they do trigger the Calculate() event.

The 'Target' and 'Cancel' Objects

Often in worksheet event stubs provided by the VBE you will see reference to two special objects (sometimes more or others also): Cancel and/or Target.

Target represents the Range [which is an object that represents a single cell, a group of cells, one or more rows and/or one or more columns] that is active at the time the event took place. Think of **Target** as the actual object itself. Anything you do to Target is done to the actual Range that it represents. Changes made to Target will appear on the sheet itself.

The **Cancel** object is a Boolean type object. A Boolean object can only have one of two conditions assigned to it: TRUE or FALSE. By default a Boolean object is FALSE (and has a numeric value of zero). If your code sets **Cancel = TRUE** then the underlying event action is cancelled: the DoubleClick never takes place or the RightClick never gets completed. These are handy events to use to take very special actions with – you can have someone double-click in a cell (and set Cancel = True) to begin a series of events unique to that cell. A real world example of this type of thing in one application I developed is that in a data area matrix that has dates in

the top row, a double-click on a date causes all rows with an empty cell in that column to become hidden: a kind of auto filter based on empty cells for that one column.

CLASS AND USERFORM MODULES

Class Modules

Quite frankly we're not going to cover Class Modules. That ***is*** an "advanced" topic in my considered opinion, and 99.9% of all coding needs can be met without using them. Creating a class takes much more preparation and thought than we have the time or space for in this book.

UserForms and their Modules

We will cover both UserForms and their underlying code modules separately later. Think of them much as worksheets and worksheet modules. Each UserForm has its own code module that contains the code associated with all objects on the UserForm.

Procedures: Function and Sub

Code modules contain code, and that code is placed into procedures, and procedures fall into two categories: Sub (or subroutines) and Function(s).

FUNCTIONS

The difference between a Sub and a Function is simply that a function can return a value to the procedure that called it. That procedure can be another Function, a Sub or even to a worksheet cell. When it is done using a worksheet formula, the Function is known as a User Defined Function, or UDF. Potentially all Functions are UDFs.

One other distinction between Functions and Subs is that (generally) Functions can only affect a single cell in a workbook, while Subs can do their work and affect almost any aspect of a workbook or worksheet. When it is used as a UDF, it can only affect the cell that it is called from; it cannot alter the contents of other cells.

A Function starts with its declaration:

Function functionName (argument1 As Type, argument2 As Type) As fType

Where Function is a reserved word declaring the start of the definition of the function.

functionName is the name you assign to the function.

Within the parenthesis you define the list of arguments and their types that are to be passed to the function for it to use to get its job done. You do not have to pass any arguments, but you do have to use the parenthesis, as:

```
Function noArgumentFunction() As Boolean
```

Finally, you declare the type of value that the function will return (fType). The type can be any valid type such as String, Boolean, Integer, Float, Double, Long, Variant, etc.

A Function ends with the End Function statement. Everything in between the function's declaration and the End Function statement is part of the function itself.

Here is an example of a function that calculates and returns the square of a value passed to it:

```
Function SquareOfNumber(anyInteger as Integer) As Long
  SquareOfNumber = anyInteger ^ 2
End Function
```

Here is how it might be called from another procedure:

```
Dim aNumber as Integer
Dim numberSquared as Long
aNumber = 15
numberSquared = SquareOfNumber(aNumber)
```

After all of that numberSquared will contain 225 (15 * 15, or 15^2)

The function could also be called from a worksheet in a cell like this:

```
=SquareOfNumber(15)
```

And 225 would appear in the cell. Actually, Excel would display the formula as:

=squareofnumber(15)
Excel makes UDF names all lowercase to distinguish them from built-in worksheet functions.

SUBS

Sub procedures are just like Functions, except that they do not return a value in the same way that a Function does. They can accept arguments, or not, just like a Function does.

A Sub starts with its declaration:

Sub subName (argument1 As Type, argument2 As Type)

Where Sub is a reserved word declaring the start of the definition of the procedure.

subName is the name you assign to the procedure.

Within the parenthesis you define the list of arguments and their types that are to be passed to the Sub for it to use to get its job done. You do not have to pass any arguments, but you do have to use the parenthesis, as:

Sub noArgumentProcess()
There is no declaration of the type of value that the sub will return because if there were, then it would be a Function and not a Sub.

A Sub ends with the End Sub statement. Everything in between the sub's declaration and the End Sub statement is part of the sub itself.

PROCEDURES: PUBLIC OR PRIVATE

By default all procedures are "Public". That is to say that they can pretty much be used from anywhere in the project. For Sub procedures, it also means that they show up in the Tools | Macro | Macros list as available to be run through that interface and for Functions, public functions can be used as UDFs. You can explicitly declare a procedure to be Public by preceding its declaration with the word "Public" like:

Public Sub aPublicSub()
or
Public Function aPublicFunction(arg1 As Variant) As Variant
But sometimes we don't want the user to have access to a procedure, or don't want other modules to be able to use a procedure in another module. For those times, you can make a procedure only accessible from within the code module that it exists in by preceding its declaration with the word Private. You'll notice that all of the Workbook and Worksheet built-in event procedures are declared as Private. Subs that are declared as Private do not show up in the Tools | Macro | Macros list, and private functions are not available for use as UDFs. Examples of private declarations are:

Private Sub aPrivateSub()
or
Private Function aPrivateFunction(arg1 As Variant) As Variant
Private procedures are normally only usable by other procedures in the same module with them. There is an exception to the rule; you can get around it by using Run "privateProcedureName". You can also use the **Call** command in a similar fashion. See the Excel VBA Help topic on CALL for limitations in using it. When you use **Run** or **Call** the procedure is executed and control returns to the line of code following the Run or Call statement.

Constants, Variables and Typing

Ready for your first programming lesson? Yeah, I know, it's about time!

In the process of writing code you are going to find the need to be able to remember things, compare things and change things. These *things* are values. You will also find times that you want to make sure that a value you use in several places in your code does not change. VBA, and all high level programming languages provide ways to deal with these needs. You may declare a constant or a variable several different ways and you can decide just exactly what kind of information that constant or variable may hold.

DATA TYPES

Before we jump into creating our constants and variables, let's discuss Typing. VBA has several data, well, types that it can deal with. Being specific with the type of data that a constant or variable is to hold can improve performance and reduce unexpected errors of a very strange and unexpected nature. For a complete list of data types, search the VBE Help for the topic "Data Types". For once, Help actually makes sense and offers real help to you by listing the numerous types that it recognizes. These include (refer to the VBE Help topic "data type summary" for a complete and current list for the version of Excel you are using).

Byte – whole, unsigned numbers from 0 to 255 (1 byte of memory).

Boolean – logical TRUE or False only (2 bytes of memory).

Integer – whole numbers from -32768 to 32,767 (2 bytes)

Long – whole numbers from -2,147,483,648 to 2,147,483,647 (4 bytes)

Single - floating-point numbers, ranging in value from -3.402823E38 to -1.401298E-45 for negative values and from 1.401298E-45 to 3.402823E38 for positive values (4 bytes)

Double - floating-point numbers ranging in value from -1.79769313486231E308 to -4.94065645841247E-324 for negative values and from 4.94065645841247E-324 to 1.79769313486232E308 for positive values (8 bytes)

Currency - numbers in an integer format, scaled by 10,000 to give a fixed-point number with 15 digits to the left of the decimal point and 4 digits to the right. This representation provides a range of -922,337,203,685,477.5808 to 922,337,203,685,477.5807.

Decimal – this is a special type that almost doesn't exist. The quote from Excel Help tells this tale: "*…signed integers scaled by a variable power of 10. The power of 10 scaling factor specifies the number of digits to the right of the decimal point, and ranges from 0 to 28. With a scale of 0 (no decimal places), the largest possible value is +/-79,228,162,514,264,337,593,543,950,335. With a 28 decimal places, the largest value is +/-7.9228162514264337593543950335 and the smallest, non-zero value is +/-0.0000000000000000000000000001*.

Note *At this time the* ***Decimal*** *data type can only be used within a Variant, that is, you cannot declare a variable to be of type* ***Decimal****. You can, however, create a* ***Variant*** *whose subtype is* ***Decimal*** *using the* ***CDec*** *function.*"

Date – January 1, 100 to December 31, 9999 (8 bytes)

String – Strings are Text. Strings come in two lengths:

String: Variable Length – zero (empty string) to approximately 2 billion characters. (10 bytes of memory plus the length of the string)

String: Fixed Length – length of the string when declared, 1 to approximately 65,400. (1 byte per character)

And you thought I was kidding about strings coming in two lengths. To continue and now we get to some that probably won't make quite as much sense to you as the list has so far.

Object – a reference to an object that you declare. When a variable is declared as an object it can take on the attributes of any legitimate object when you use the **Set** command to assign it to a specific type of object. An object is much like the Variant type that you are about to see. (4 bytes)

Variant – any variable that is not defined as a specific type is by default of type variant. A variant can take on the attributes of any other type depending on how values are assigned to it. Generally you should refrain from declaring variables as type Variant, however sometimes it is actually required that a Variant be used in some circumstances. Somewhat like strings, Variants come in two variations (yeah, now I am playing word games)

Variant – **with numbers** a variant can take on any value up to that of type Double. (16 bytes)

Variant – **with text characters** a variant has the same limits as a variable length String type, but it takes 22 bytes plus the length of the text in memory.

Finally, we get to the truly catch-all-nothing-else-will-do type, the **user defined type**! Yes, you can define your own type. These are special cases where you use a combination of other types to define your own. We will use at least one user defined type in our learning in time. For now picture this: you create a user type that you call EmployeeRecord and it consists of a type that can hold some text, some numbers, a date or two and even a currency value, any of which can be referenced as a property of a variable that you declare as type EmployeeRecord.

Oh crap! I've forgotten an entire application worth of types!! For each application that VBA is implemented in, any object in that application can be used as a type. So in VBA for Excel you can declare variables as specific objects such as the Application itself, a Workbook, a Worksheet, a Range, a chart, a style, and just about anything else that exists in the Excel world. We will definitely deal with this kind of assignment of type later on – Excel VBA is just a cripple if we don't make use of this incredible ability.

Alright, let's declare some variables and constants and discuss what we might do with them. But we are going to do it all for real and write some code to use the constants and variables that we define. We will start on the next page.

OUR FIRST PROCEDURE

Start by opening Excel with a new workbook. Press [Alt]+[F11] to enter the VBE. In the VBE use the menu toolbar Insert | Module options to create a module that we can put some code into.

Procedure names should be at least somewhat meaningful and hopefully will give some insight about their purpose or what they're going to do. So naturally we will call this procedure MyFirstProcedure. Go ahead and get it started – click anywhere in the code window below the *Option Explicit* statement. If you don't have an *Option Explicit* statement at the top of the code module then start reading back at page 1, please.

This procedure will show you one way of getting an entry from the user, do something with that data and both show the result on a worksheet and in a message to the user.

```
Sub MyFirstProcedure()
'this procedure accepts a numeric input from the user
'calculates that value raised to a specific power
'and places the result on a worksheet and also
'displays it in a message box to the user.
'***declare a constant to hold the name of the sheet
'that will receive the results of our calculation
'this lets you call this procedure from anywhere at
'any time and always have the result go to a
'specific sheet.  If the sheet's name changes
'you can change it here and the code will
'continue to function in the same way.
  Const dataSheetName = "Sheet1" ' name of sheet
  Const raiseToPower = 2 ' square the number
  Dim theNumber As Integer ' from user
  Dim theResult As Integer ' calculated value
  'get the number from the user.
  'InputBox accepts any input, even numbers and dates,
  'but it accepts it as text which we may have to massage.
  'so we will have to make sure that they
  'entered a number or something that looks like one
  'we will display a prompt, a title and create a default
  'value of zero
  'because theNumber has been declared as an integer, if the
  'user enters something non-numeric, a run-time error will
  'take place.  If you experiment with that and get a
  'runtime error 13 (type mismatch), just click the [END]
  'button to bring things to a safe stop.
  theNumber = InputBox("Enter a whole number", "Integer Input", 0)
  'the ^ symbol means "raise to power"
  theResult = theNumber ^ raiseToPower
  'put theResult on a worksheet in cell A1
  Worksheets(dataSheetName).Range("A1") = theResult
  'display an explanation to the user in a message box
  MsgBox theNumber & " raised to the power of " & raiseToPower & " = " & theResult
End Sub
```

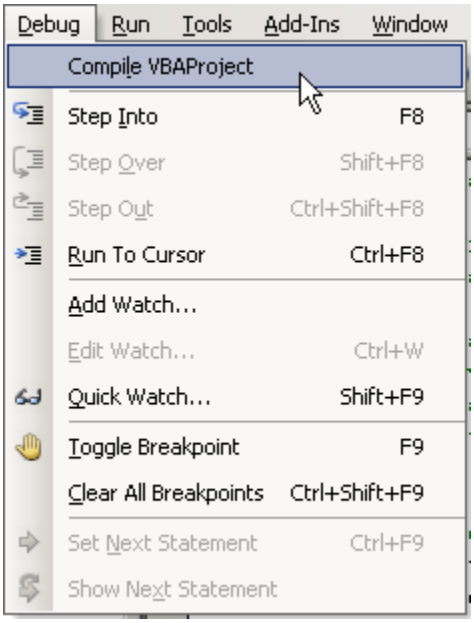


Figure 12 The VBE Debug Menu

Running your first procedure. There are a couple of ways to test your code at this point. But before just trying it out, it's a good idea to make a couple of "desk checks". Read through the code again to see if you notice any obvious errors, such as perhaps typing * (multiply) instead of ^ (raise to power). The VBE can also help you with a desk check step, and it's very critical of your code and can help find problems very early on. From the VBE menu choose **[Debug] | *Compile VBAProject.***

If it doesn't find anything wrong, it will simply blink and do nothing else – it doesn't give "all clear" message. If it does find a problem, it will highlight the first offensive line that it finds and tell you what the problem is.

Fix the problems and repeat the process until it doesn't report any more errors to you.

You can run the code directly from within the VBE itself. Click anywhere within a procedure and press [F5]. This is much the same as using **[Tools] | *Macro* | *Macros*** from the Excel menu toolbar. Here is a shot of the results for your first procedure when things go right:

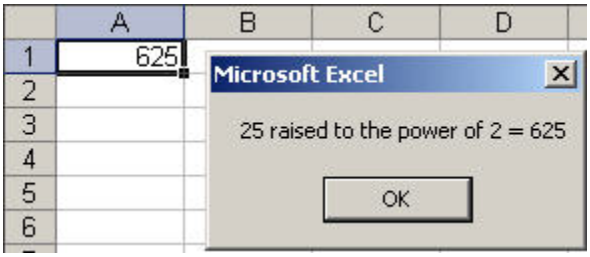


Figure 13 MyFirstProcedure Results

One thing you may notice both in the code above and in the VBE itself is that some words and phrases are in one color while others are in a different color. This is by design and is to help you read and interpret the code. VBA 'reserved' words are shown as blue text, while comments are shown as green text, while pretty much everything else is in black. Your editor may be set up to show these things in different colors, but there will be differences in colors for the different meanings of the code pieces. There is one more color that you may see from time to time – red text indicates a line of code that VBA has determined to contain one or more errors.

Reserved Words

You cannot use words unique to the VBA language as the names for your own constants and variables. The list is pretty long, you'll learn what you can use and what you cannot during your coding efforts. Words like *For, Next, Do, Loop, Until, Dim, Const, InStr* are reserved for the language and you can't use them except as the instructions that they are. It is even considered bad form to use a reserved word as part of a constant or variable name because it can confuse anyone reading the code later, so while intNext is a valid name, it is an unwise one to use; but a name like intNextNameInList would probably be a good one to use.

Comments and Remarks

It is always a good idea to add comments to your code. How many to add is a judgment call on your part. But a comment should add understanding and not just repeat what the code is doing:

```
X = X + 1 ' add one to X
```

That comment doesn't add any value to the code at all. It would be better to explain *why* one is being added to the value of X:

```
X = X + 1 ' increment the pointer into the array holding employee names
```

That would be a much more informative comment to add (assuming it's true, of course). It tells why the value of X is being increased, and informs the reader where to expect to see it used somewhere else in the code (as an index or pointer into an array [list] of names).

Comments typically begin with the single quote mark as I've used in the examples. Everything following the single quote mark is ignored by the VBA engine. Comments only add to the understanding of the code and to the size of the source code file – they do not add to the time it takes to execute the code.

I said "typically" above because you may also start a comment, or remark, with REM, as

```
REM increment the pointer into the array holding employee names
```

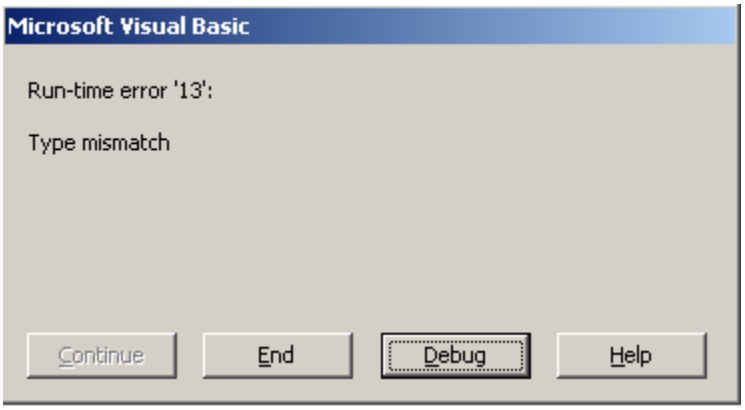
This is a holdover from earlier times and has its roots in the original interpreted BASIC language. However, there are restrictions in using it that makes it inconvenient. REM must be used as the first word in a line in the procedure, otherwise it will generate an error:

```
X = X + 1 REM increment the pointer into the array holding employee names
```

Since the single quote mark has become THE accepted VB notation for the beginning of a comment, using the word REM can actually add some difficulty in reading the code. Using the single quote in some places and REM in others would add even more confusion. **Be consistent**, and simply use the single quote mark as the start of comments in your code.

ERROR HANDLING: A BEGINNING

There isn't too much that will make your users think of you often and most unfavorably than for them to have entered a lot of important information and suddenly have the program blow up in their face with an unhandled error (also called an exception in some languages).



If you try running MyFirstProcess and supply a word or other non-numeric entry to be processed you will get a Type mismatch error because we defined variable theResult as an Integer type, and words are not integers. Clicking [End] will stop the process; clicking [Debug] will take you to the line in the code where the error took place.

Figure 14 BOOM! Unhandled Errors Are a Pain

So what can we do about such situations? VBA provides the ***On Error*** statement to help deal with both anticipated errors and those not so anticipated. Using one form of it, we can change our code just a little and keep it from failing as dramatically and allow the user to recover from the error.

Here is our code revisited, with some comments removed and others added, along with some error handing added in.

```
Sub MyFirstProcedure()
  Const dataSheetName = "Sheet1" ' name of sheet
  Const raiseToPower = 2 ' square the number
  Dim theNumber As Integer ' from user
  Dim theResult As Integer ' calculated value
  '
  ' add a test for a possible error
  ‘ this form of On Error says “if an error occurs, just ignore it for the moment”
  ‘ but do remember that it did happen.
  On Error Resume Next
  theNumber = InputBox("Enter a whole number", "Integer Input", 0)
  'now test if special system object ERR indicates something bad happened
  If Err<>0 Then
  'something bad did happen, we don’t much care what although we can
  'presume it was the anticipated Error 13 – Type Mismatch
    MsgBox “Your Input was not numeric.  Please Enter an Integer Value”, vbOKOnly, ”Error”
  'clear the error condition and exit the procedure
    Err.Clear
  'also reset error handling to let the system once again deal with problems
    On Error Goto 0
    Exit Sub
  End If
  'no error detected, continue on, but first remove
  ‘ our “error trap”
  On Error Goto 0 ‘ allows errors to be handled by the system again
  theResult = theNumber ^ raiseToPower
  'put theResult on a worksheet in cell A1
  Worksheets(dataSheetName).Range("A1") = theResult
  'display an explanation to the user in a message box
  MsgBox theNumber & " raised to the power of " & raiseToPower & " = " & theResult
End Sub
```

Now if you run the procedure again and enter a word or something other than a number, you are gently requested to correct the error of your ways and allowed to try again without the entire application crashing to the ground.

Now we will continue our interrupted discussion of declaring variables and constants.

CONSTANT AND VARIABLE DECLARATIONS REVISITED

We've written a small procedure that involved declaring and using some variables. Now we can talk about them in a little more depth. One thing we need to discuss is ***SCOPE***. Scope refers to what parts of a program can see a particular variable or not. There are three levels of variable scope in VBA:

Procedure Level Scope

A variable declared inside of a procedure has procedure level scope. The variables and constants we declared in MyFirstProcedure had procedure level scope.

Procedure level variables are created when the procedure begins to execute, they are only available to be used within the procedure and they cease to exist when the procedure ends at the procedure's Exit Sub or Exit Function statement.

As with any good rule, this one has an exception. If you declare the variable using the **Static** declaration instead of **Const** or **Dim** statements, then the variable will retain the last value assigned to it when the procedure ended as its initial value the next time the procedure executes. A trivial example: try placing this code in a module and just press [F5] several times to watch the value of myStaticCounter go up each time.

```
Sub StaticsAtWork()
  Static myStaticCounter As Integer
  'each time this procedure is called, myStaticCounter value will increase by one
  myStaticCounter = myStaticCounter + 1
  MsgBox myStaticCounter
End Sub
```

Even though myStaticCounter retains its last value, it still cannot be accessed to determine its value outside of the procedure – it retains its procedure level scope.

You cannot use the **Public** or the **Private** declarations within a procedure. For all practical purposes all declarations within a procedure are *private* to that procedure.

Module Level Scope

The next step up the scope food chain is module level scope. These are constants and variables that can be used/evaluated/modified (for variables) by any procedure in the module. Module scope variables and constants are declared in the *General Declarations Section* of a module.

The *General Declarations Section* of a module is the area ahead of any declaration of a procedure. The *Option Explicit* statement that we've already seen is in this section of the modules. Declare your module scope variables and constants after the *Option Explicit* statement and before any procedure declaration.

You can use the **Dim** and **Const** statements to make declarations in this area but it is clearer to the reader if you use the **Private** declaration statement so that readers will know later that these variables and constants are private/local to the module:

```
Private anyModuleLevelScopeVariable As Variant
Private Const anyModuleLevelScopeConstant = "The whole module can see me!"
```

Why specify Private? As you are about to see, Public (or entire VBAProject scope) objects are also declared in this section of a module, any standard module.

Public Scope

Public is a term that was previously Global. A variable or constant declared as Public in the General Declarations section of any standard module has visibility/accessibility in any procedure in any module in the entire project.

Why not just declare everything Public and be done with it? Because in more than the simplest application you will invariably change the value of a public variable at the wrong time/place causing yourself mega-headaches in debugging it all. Overall it is best to keep the scope of your declared values at the lowest level possible. You will have fewer problems and easier debugging all around by doing that.

I personally like to put all of my Public constants and variables into a single module with comments provided to explain where they are used and what they are used for. This provides a single central point of management for the Public values.

What are candidates for Public values? Look for things that you find yourself using repeatedly for the same purpose in several areas of your project – perhaps using the same worksheet name to perform operations with the sheet; definitions of the layout of those worksheets, constants that your logic depends on heavily and are used in multiple areas.

When to Use Constants and/or Variables

One question that comes up from time to time is "why use constants at all – why not just use their value(s)?". I'll answer your question with a question: which is more informative to you here?

```
If ActiveCell.Row < 4 Then
```

Or

```
If ActiveCell.Row < firstRowWithData Then
```

Not only is the second form more understandable, it keeps you from having to track down every place you've used 4 as a value and trying to figure out if you mean the first row with data on a sheet, or are comparing ages of pre-school children, or seeing if the word Mississippi has the correct number of 'i's in it.

Finally, using named variables/constants helps prevent typographic errors. [**Debug] | *Compile VBAProject*** will find errors in variable/constant name spellings rapidly, but it cannot do anything at all to determine that you typed a 4 when you really meant to type a 5 and just had your finger on the wrong key when you typed it. A common error, known as FFS (fat finger syndrome).

Good Programming Practices

We've already discussed **one good programming practice** that is beneficial: having the VBE automatically require declaration of constants and variables before there use.

Most good programming practices fall into the category of either good common sense or of following a generally accepted standard, as with the use of the single-quote/apostrophe as the beginning of a comment.

Good programming practices will improve your chances of actually writing Good Code.

WHAT IS GOOD CODE

There are probably as many definitions of "good" code as there are programmers. My definition: Good Code is code that performs the task required and does so reliably. Good Code is also maintainable.

Some examples of Good Code at work in this day? WinZip. IrfanView. Microsoft's Calculator and Notepad. They do what they're designed to do, they do it simply, and they do it reliably. I'm sure you can think of many more examples, just as you can think of programs written with "bad code" – that are far less than you expected when you acquired them.

Good Code also takes the user into consideration – making things easier for them, performing the application's tasks with a minimum of fuss, bother and annoyance. This is the human-machine interface side of code design and development. Working closely with your client or studying your intended audience can help you design an effective, usable interface for your application.

GOOD PROGRAMMING PRACTICE #2

Be consistent. If you don't follow any published standards for conventions such as commenting, constant/variable naming, source code listing formats or others, then at least be consistent within your own code in the way you do these things. This will make your code more readable and understandable to you, and will tend to making extending the code to include new features, modify old ones and simply fix bugs than if you do things one way today and some other way the next. That is not intended to keep you from changing the way you do things as you discover better ways to do them.

MORE GOOD PROGRAMMING PRACTICES

As we encounter situations in our coding examples where a Good Programming Practice can be demonstrated, they will be pointed out and labeled as **GPP #n** That may be more effective in showing them to you 'in context' than just describing them in a list here.

Looping Structures

One thing that computer code is good at doing is something dull and tedious for us human types: **repetitive actions**. A macro in itself is a way of doing something repetitive, with varying levels of complexity, over and over with ease and without boring ourselves to death doing it. Each time we run a Macro or cause a procedure to be called, we are performing some repeated process.

Within procedures we may also need to perform a particular task many times. The use of looping structures such as:

For … Next

For Each … Next

Do … Loop

Do … Until

and

Do … While

All give us slightly different ways to perform actions, calculations and other processing many times in a relatively small section of code.

GPP #3:

Keep the amount of work inside of a loop to a minimum. If there is something that can be done outside of the loop before starting it, do it outside of the loop.

For example let us assume you want to take a list of numbers you have in a range and increase those values by some percentage that you have stored. You could code it like this:

```
For Each anyValue In listOfValues
   anyValue = anyValue * (1 + percentIncrement)
Next
```

The problem with this is that each time through the loop, the value of 1 + percentIncrement must be recalculated. You can increase the efficiency of that loop by calculating that value before entering the loop, as:

```
tempValue = 1 + percentIncrement
For Each anyValue In listOfValues
   anyValue = anyValue * tempValue
Next
```

FOR … NEXT LOOPS

The simplest and oldest loop structure in Basic is the For…Next loop. The general form, or syntax, of the command is

```
For counter = startCount To endCount Step stepValue
        Executable statements and comments to be performed
Next
```

For is a reserved word that marks the beginning of the loop.

counter is a variable that is used to control how many times the code within the loop is performed.

startCount is a variable, constant, or calculated value that determines the initial value of *counter*.

To is a required reserved word that separates the starting value from the ending value

endCount is a variable, constant, or calculated value that determines the maximum value that *counter* may be assigned before the loop terminates.

Step (and *stepValue*) are optional arguments that allow you to change the way *counter* values between *startCount* and *endCount* are calculated. *stepValue* may be a variable, constant, or calculated value. The default, when Step *stepValue* are omitted from the command, is 1 (one).

Next is a reserved word that is used to mark the end of the For loop.

FOR EACH LOOPS

This is a special loop that works very much like the For…Next loop, but it loops with 'objects' within larger group of the same type of objects. The Excel engine is smart enough to figure out that part of it. That is, if your larger group is Worksheets, it knows to work with each individual worksheet in the group; or if your range is a group of cells, it knows to work with each individual cell within the group.

Typical setups for using For Each might be similar to these:

```
Dim groupOfCells As Range ' remember, a range can refer to 1 or more cells
Dim anySingleCell As Range ' remember, a range can refer to 1 or more cells
Set groupOfCells = ThisWorkbook.Worksheets("SomeSheetName").Range("A1:A500")
For Each anySingleCell In groupOfCells
… do some work within the loop
Next
```

Here is an example that would protect all sheets in the active workbook without a password.

```
Dim anySingleSheet As Worksheet
For Each anySingleSheet In ActiveWorkbook.Worksheets
   anySingleSheet.Protect
Next
```

A **For...Next** loop will always execute at least one time. Don't believe me? Try this code:

```
Sub test()
  Dim x As Integer

  For x = 0 To 0
    MsgBox "inside of the loop"
  Next
End Sub
```

You will see the message once, proving that the code inside of the loop did run.

If a **For...Next** loop runs to completion, the value of the *counter* value will be one more than the *endCount*. Example:

```
Sub test()
  Dim x As Integer
  Dim y As Integer

  For x = 1 To 10
    y = y + 1
  Next
  MsgBox "Counter x is: " & x & vbCrLf & "Value y is: " & y
End Sub
```

The message displayed should be:

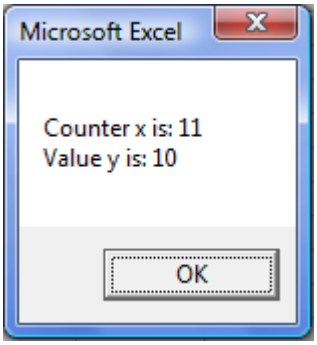


Figure 15 For...Next Loop Counting Results

What does this tell us? Simply that the test for the counter value is made at the For statement, not at the Next statement. Generally this isn't information of great interest, but it can be handy to know at times.

Note: in the MsgBox statement, ***vbCrLf*** is a built-in VBA constant that provides a newline; that is, it provides a Carriage Return and a Line Feed, thus they called it vbCrLf.

DO... LOOPS

There are several varieties of the Do loop and this variety makes them a bit more versatile than the sturdy, but rather plain vanilla For...Next loop.

In exchange for this versatility, you have to do a little more work in the form of helping to control when the loop terminates. Consider this simple loop.

```
Sub LoopForever()
  Do
    MsgBox "Pete and Repeat were in a boat.  Pete fell out.  Who was left?"
  Loop
End Sub
```

Use [Ctrl]+[End] to break into that code if you actually try running it.

There is nothing in that code to stop the loop from processing, so it pretty much runs forever. While there are times when you may actually choose to implement such a loop, you usually want a way to halt one either manually or automatically. We will rewrite the code a little to get it to halt automatically after having annoyed you just a little.

```
Sub LoopForever()
  Dim loopCount As Integer
  Dim y As Integer
  Do Until loopCount = 3
    MsgBox "Looped " & y & " times."
    loopCount = loopCount + 1
    y = y + 1
  Loop
End Sub
```

the value of loopCount is used to exit the loop once it reaches a value of 3. How many times will the message appear? No, not a trick question - the message will appear 3 times.

But what happens if we change it just a little bit?

```
Sub LoopForever()
  Dim loopCount As Integer
  Dim y As Integer
  Do While loopCount < 4
    MsgBox "Looped " & y & " times."
    loopCount = loopCount + 1
    y = y + 1
  Loop
End Sub
```

So I'll ask the question again: How many times will the message appear? In this case it becomes a trick question. You'll see the message 4 times. And yet logic tells us that 3 is less than 4, but the test must be done at the Do While statement and that means that we get an extra, sometimes unexpected pass through the loop.

```
Sub LoopForever()
  Dim loopCount As Integer
  Dim y As Integer
  Do
    MsgBox "Looped " & y & " times."
    loopCount = loopCount + 1
    y = y + 1
  Loop While loopCount < 4
End Sub
```

Again the message will be displayed 4 times because even though we've moved the test to the bottom of the loop, we still have to get some value into loopCount that equals or exceeds 4 in order to exit the loop. So be sure you know how many times your loop will execute if you are depending on it to exit after a specific number of iterations. If we rewrite the last section like this:

```
Sub LoopForever()
  Dim loopCount As Integer
  Dim y As Integer
  Do
    MsgBox "Looped " & y & " times."
    loopCount = loopCount + 1
    y = y + 1
  Loop While loopCount < 3
End Sub
```

Then the message will be displayed 3 times, presumably as you expected it to. You could also change that last statement to:

```
  Loop Until loopCount = 3
```

and get the message displayed 3 times.

What about a loop that needs to execute some undetermined number of times? Let's say that you need to pull the characters off of the front of a string of characters until you encounter a numeric character, but you don't know where in the string the number will be found. You can use a 'flag' to indicate when you have met the requirement. Here is an example:

```
Sub StripToFirstDigit()
  Dim strippedText As String
  Dim positionInString As Integer
  Dim allFinished As Boolean ' default value when declared is FALSE
  Const testPhrase = "abcdef123xyz"

  Do Until allFinished ' loop until flag allFinished becomes TRUE
    positionInString = positionInString + 1
    If Mid(testPhrase, positionInString, 1) >= "a" Then
      strippedText = strippedText & Mid(testPhrase, positionInString, 1)
    Else
      'found something that's not a letter, assume a number
      allFinished = True ' set flag to exit the loop
    End If
  Loop
  MsgBox "Stripped Text is: '" & strippedText & "'"
End Sub
```

We had to do a little more work but we got a lot of added functionality. Try changing the value of constant testPhrase and see how it works. Just make sure you have at least one non-alphabetic character in the phrase since we haven't tested to see if positionInString ends up becoming greater than the number of characters in testPhrase.

DO LOOPS CONTROL SUMMARY

For those that may have gotten confused along the way, here is a short description of how the different versions of **Do** loops work:

```
Do
        'your code within the loop will run (execute) until you take some action within the code to force it
        'to exit the loop using an Exit Do statement.
        'without such a control, it becomes an "infinite loop".
Loop
```

```
Do Until testCondition
        'your code within the loop will execute only while the testCondition is FALSE.
Loop
```

```
Do
        'the code executes at least one time and will continue to execute within the loop as long as the
        'testCondition is FALSE
Loop Until testCondition
```

```
Do While testCondition
        'your code within the loop will run (execute) only while the testCondition is TRUE.
Loop
```

```
Do
        'the code executes at least one time and will continue to execute within the loop as long as the
        'testCondition is TRUE
Loop While testCondition
```

Generally there's no accepted standard for which of the Do Loop types (While or Until) to use other than your own personal preference. Typically you can write either type to accomplish the same task and get the same result.

Decision Makers

There are two primary decision making tools in VBA: If...Then and Select Case. We can look at the loop structures as decision makers also, but they are kind of indirect decision makers. The **If...Then**, and its brothers **If...Then...Else** and **If...Then...ElseIf...Else** along with **Select Case** are very definitely there to assist you in changing the path of the program or the logic of a process; i.e., they help you make decisions about what to do next based on the result of calculations or actions at a specific point in your process.

IF...THEN

This is the most basic of the decision makers. Using it assumes there is pretty much only one test to perform and only one action to take if the result of the test is true. It can be written as a one-line statement such as:

```
If X = 2 Then Y = 5
```

Very straight forward statement: if at this point in the process X equals 2, then set Y to 5. If X does not equal 2 at this point, Y will retain whatever value it has at the moment.

A personal preference of mine is to make even this simple statement a "block" because I think it makes the code more readable and understandable. This is exactly the same statement, but in "block" form:

```
If X = 2 Then
  Y = 5
End If
```

This form also allows us to easily and clearly perform more than a single action based on the result of the decision, like this:

```
If X = 2 Then
  Y = 5
  Z = 9
  aStringVariable = "X was 2"
End If
```

So in this example, we perform three actions when we find that X has a value of 2. The End If statement also gives us a clear view of what will be done when X = 2 by defining the end of the If Then code block.

But what if we need to do one thing when X = 2 and do something else when it doesn't? Enter the If...Then...Else statement.

IF...THEN...ELSE

Taking If...Then to the next step, this decision maker lets us exercise a couple of options based on the value of something. This next snippet of code shows us how it can be used:

```
If X = 2 Then
  Y = 5
  Z = 9
  aStringVariable = "X was 2"
Else
  Y = 1
  Z = 3
  aStringVariable = "X was not 2"
End If
```

So here we are saying that when X = 2, we set Y, Z and aStringVariable to particular values, but if X is some value other than 2 then we set those same variables to a different set of values.

We can extend this decision making even farther using ElseIf along with what we've already seen.

IF...THEN...ELSEIF...ELSE

Using this combination we can test for different values of a particular item. For this example, we need to set Y, Z and aStringVariable to specific values when X is either 2 or 3, and another set of values when it is not 2 or 3.

```
If X = 2 Then
  Y = 5
  Z = 9
  aStringVariable = "X was 2"
ElseIf X = 3 Then
  Y = 0
  Z = 99
  aStringVariable = "X was 3"
Else
  Y = 1
  Z = 3
  aStringVariable = "X was neither 2 nor 3"
End If
```

You may actually have many ElseIf ... Then statements before the final Else statement, making this a multi-conditional decision maker. But for the times when you have many decisions to make based on the value of one (or more) variable value(s), the Select Case statement is more efficient.

SELECT CASE

Select Case is used much like If...Then and its variants. It's just more compact, provides improved readability and is more efficient than stringing a long series of ElseIf ... Then statements into the code.

To show how it can work, we will use the same situation that we had for the last example in the If...Then variants section.

```
Select Case X ' base our decision on the value of X
  Case Is = 2 Then
    Y = 5
    Z = 9
    aStringVariable = "X was 2"
  Case Is = 3 Then
    Y = 0
    Z = 99
    aStringVariable = "X was 3"
  Case Else
    Y = 1
    Z = 3
    aStringVariable = "X was neither 2 nor 3"
End Select
```

You can have any number of Case Is type statements, allowing you to make decisions based on a large number of possible values for a variable.

You do not HAVE to have a Case Else statement, but it is wise to have one. It doesn't even have to do anything, but having a "do nothing" section tells others reading the code later that no action is taken if a value doesn't meet one of the stated values. Here's a "do nothing" setup:

```
Select Case X ' base our decision on the value of X
  Case Is = 2 Then
    Y = 5
    Z = 9
    aStringVariable = "X was 2"
  Case Is = 3 Then
    Y = 0
    Z = 99
    aStringVariable = "X was 3"
  Case Else
     ' no action required or desired when X is not 2 or 3
End Select
```

The Select Case block always ends with the End Select statement.

Data Sources

There are lots of sources for data to work with inside of Excel:

- Cells on worksheets
- Files external to the Excel workbook (and I include things like queries to obtain data from a variety of sources such as database files or from a networked location or internet site)
- The user!

There are a couple of things you need to keep in mind when getting data from any source:

- You have to know where to find it, and what actions to take to get it into your VBA code to work with, and
- Remember that what you expect to get is not always what you actually do get. We'll cover this aspect some more in a short discussion of *data validation* later on.

DATA FROM WORKSHEETS: INTRO

Within VBA you can get data from any cell or group of cells on any worksheet in any open workbook. Later on I'll show you how to do this without ever leaving the cell that is currently active on your screen.

You will need to know where to look for the information or how to find it, and unless you are working in a fairly structured situation, you may need to perform some data validation on it before trying to use it in your code.

DATA FROM EXTERNAL SOURCES

The possible external sources and their types is so varied that we can't really cover them all here. You'll need to know how to either open any external data file such as a .txt, .dat or .csv file and read from it and you'll need to know the format of the data in the file. Usually you have an idea about these things before you begin writing the code to access the external files, so don't worry about it at this time. Sometimes finding out what's in a file and how it's all laid out requires some 'legwork'; that is, you may have to open the file and bring in the data without using Excel and simply examine it to see what's what within it.

When querying databases you will probably have some guidance from those who created the database and maintain it as to what tables and fields within those tables you are going to need to reference to get what you want from it. This takes us into the realm of SQL (Structured Query Language) and that's definitely beyond the scope of this book!

USER PROVIDED DATA

Working with data provided "on the fly" or in "real time" from the end user is almost an art. You cannot EVER be certain that they'll provide the information you've requested in the form that you need it or that it will even be the same kind/type that you asked for! User input data is almost always in dire and desperate need of data validation before using it.

I can quickly think of four typical ways of getting data from a user in Excel:

- They type it into cells on worksheets and you read it from there - which goes back to the earlier section on **DATA FROM WORKSHEETS**
- Data entered by the user in response to the use of the InputBox$() function in VBA. This is useful for getting single quick input from the user when you need it.
- Evaluating the user's response to a MsgBox$() function that uses several buttons, as [Yes], [No] and/or [Cancel] on it to allow the user to indicate their response to the prompt you have provided as part of the message displayed. Very little data validation is needed with this one.
- Inputs provided on a UserForm. A UserForm allows you to get many inputs at once from the user. This is a good way to gather lots of information at once, but you're going to need to do data validation on a lot of it of some type before actually making use of it.

We'll take some quick looks at the last 3 of these in this section, nothing in great detail, but hopefully enough to give you an idea of the abilities of each of those 3 methods of obtaining information from the user.

INPUT USING INPUTBOX$()

In VBA code the InputBox$() function is coded as shown below. For this example we are going to ask the user to enter what we plan on using as a starting balance for a worksheet that is set up to act as a checking account program of some type. So we are expecting a numeric input that we will want to use as money (Currency).

```
Sub GetACurrencyEntry()
  Dim dataAccepted As Boolean ' a flag to tell us when we think the input is good
  Dim userInput As Variant ' use variant to accept any type of entry the user may provide
  Dim acceptedInput As Currency ' we will store the validated/accepted amount in this variable

  dataAccepted = False ' initialize to remain in the loop until a good entry is made
  Do Until dataAccepted ' implied test of dataAccepted = True
    userInput = InputBox$("Enter the Starting Balance for the account:", "Starting Balance", 0)
    If IsNumeric(userInput) Then
      'looks ok, at least it starts with numbers
      dataAccepted = True ' so that we will exit this loop
    Else
      'oops, not looking very good
      MsgBox "Please enter a dollar amount to continue..."
    End If
  Loop
  acceptedInput = Val(userInput) ' get the numeric value of the validated/accepted entry
'... continue on to use acceptedInput in your code
End Sub
```

As you can see, we've set up a loop to keep asking the user for some numeric entry until we get one from them. We use the Boolean flag, dataAccepted, to tell us when we think it is alright to use what they entered later on in our processing.

Let's quickly look at the line of code that gets the input from the user:

```
userInput = InputBox$("Enter the Starting Balance for the account:", "Starting Balance", 0)
```

InputBox$() can also be written as InputBox() but I use the $ with it as a reminder that if something is entered, it is going to be a string/text even if it looks like something else such as a number, currency amount, time or date.

InputBox$() takes 3 basic parameters:

- A prompt to be shown to the user,
- some text to use as a title in the dialog and
- a default value to use if the user just clicks the [OK] button.

Here's what this looks like at runtime:

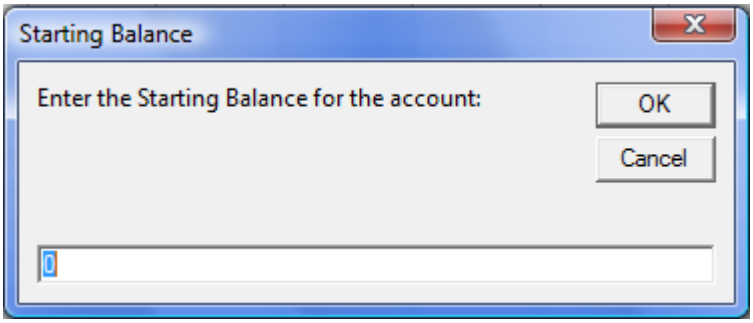


Figure 16 InputBox$() Example

You can see where the three pieces of information were used when the line of code was executed. If the user just presses [Enter] or clicks the [OK] button at this point, we get zero (the default value we provided) as the starting balance.

But if they don't enter something that looks like numbers, they will get a reminder message and the dialog will be shown to them again:

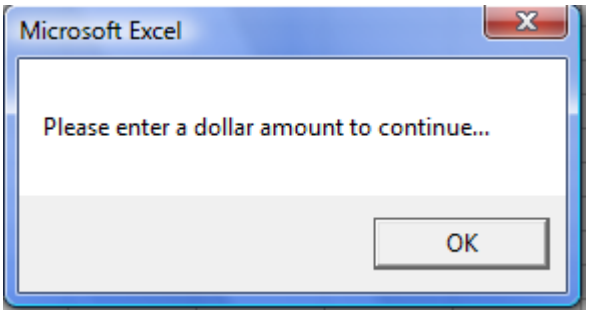


Figure 17 InputBox$() Validation Failed Message

But there are 2 situations that can come up here that we haven't taken into consideration in our data validation: The user clicks the [Cancel] button or the user clicks on the close dialog X.

In both of those cases we get a zero length string back into variable userInput, not a zero or anything else. We have to expect this to happen, look for it, and decide what to do if it happens. That's more of the data validation process. Here is the code segment with a test for this situation added to it:

```
Sub GetACurrencyEntry()
  Dim dataAccepted As Boolean ' a flag to tell us when we think the input is good
  Dim userInput As Variant ' use variant to accept any type of entry the user may provide
  Dim acceptedInput As Currency ' we will store the validated/accepted amount in this variable

  dataAccepted = False ' initialize to remain in the loop until a good entry is made
  Do Until dataAccepted ' implied test of dataAccepted = True
    userInput = InputBox$("Enter the Starting Balance for the account:", "Starting Balance", 0)

    If userInput = "" Then
      'user either clicked [Cancel] or closed the dialog window
      'we have to decide what to do in this case and code it
      'into this section
    End If

    If IsNumeric(userInput) Then
      'looks ok, at least it starts with numbers
      dataAccepted = True ' so that we will exit this loop
    Else
      'oops, not looking very good
      MsgBox "Please enter a dollar amount to continue..."
    End If
  Loop
  acceptedInput = Val(userInput) ' get the numeric value of the validated/accepted entry
'... continue on to use acceptedInput in your code
End Sub
```

You're probably going to ask "Well, teach, what do we do if userInput = ""?" My answer is "that depends". It depends on how you want to handle it. You could toss up a prompt asking if they wish to continue and put up the dialog again or you could put the default value of 0 into variable userInput just as if they'd clicked [OK] or you might even ask them if they want to quit futzing around with their checkbook for now and if so, shut down so they can restart everything later on.

USING MSGBOX$ AS USER INPUT

You can use the MsgBox$() [which can also be coded as MsgBox()] as a fast, reasonably accurate method of getting a short Yes/No answer from your user during your processing.

Normally MsgBox() just puts up a message with an [OK] prompt and continues to process after the user hits the [Enter] key or clicks the [OK] button. Code for such a thing might look like this:

```
MsgBox "We are on page 34 of the tutorial.  Press [Enter] or click [OK] to continue", vbOkOnly,"Checkpoint"
```

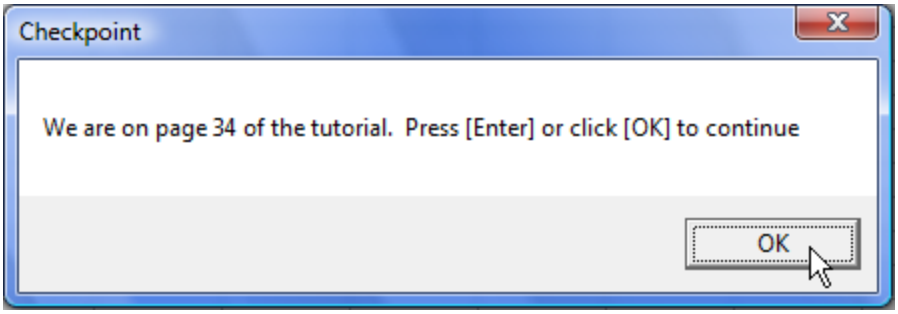


Figure 18 Plain Vanilla MsgBox$() Displayed

But what if we want the user to make a choice right now? We could change it a little bit and ask if they want to continue reading or take a break. You can 'capture' and evaluate the user's response to a MsgBox() by forming it as a function and using it as a test, like this:

```
Sub Checkpoint()
  If MsgBox("We are on page 34 of the tutorial.  Would you like to continue", vbYesNo, "Checkpoint") = vbNo Then
    Application.Quit ' close Excel!!
  End If
  'just continue on here...

End Sub
```

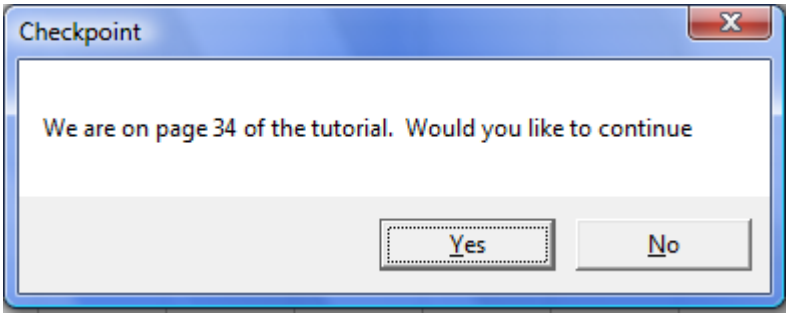


Figure 19 MsgBox Used to Obtain User Input

By enclosing the prompt, button choices and title within () we have turned it into a function that returns a code associated with the button they click. It would look like this on screen:

vbOkOnly, vbYesNo and vbNo are constants automatically available within VBA - you don't have to predefine them anywhere in your own code. There are others that can be used with MsgBox also, such as vbExclamation, vbCritical, vbYesNoCancel and some that let you determine which button on a multi-button message is the default (is used if they just hit the [Enter] key).

USERFORM AS A DATA SOURCE

I'm going to show you a big form from an actual project I'm working on when I'm not trying to finish up this book. We won't discuss all of it, but we will take one or two of the controls on it and discuss them to show things like how to validate the data and how to get it from the form out into a cell on a worksheet.

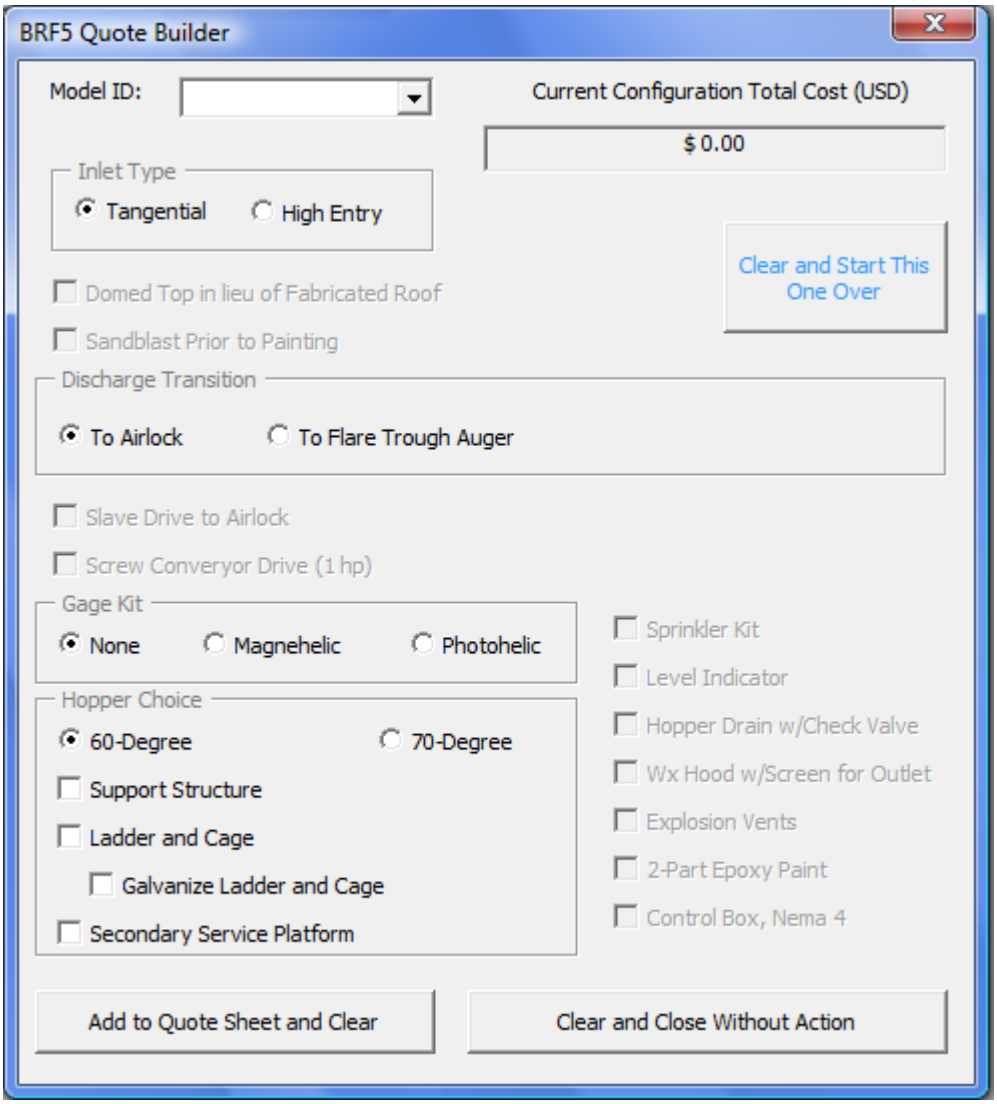


Figure 20 Multi-Control UserForm

I chose this one because it uses many of the possible controls you can use on a user form. It starts off using a ComboBox control to present a list of possible models of a piece of equipment that the user can select from to begin to build up a cost for the item. The astute observer will notice that everything except that ComboBox is disabled right now. That means that the user can only choose a Model ID at this point. That is part of my own data validation here: they can't choose pieces of an equipment item without first telling what equipment item they are going to be working with. Once they choose from the list, everything else gets set up to hold and accept legitimate values from the user. All of the other controls on this form really need no further data validation because the form itself and the way option buttons and check boxes within groups work does it for me automatically. What I want you to see here is the variety of controls you can

put on a user form. One control that is definitely missing from this one is a plain text entry area such as you might use to get someone's name, address or other information. Here is another form from the same project that has lots of those.

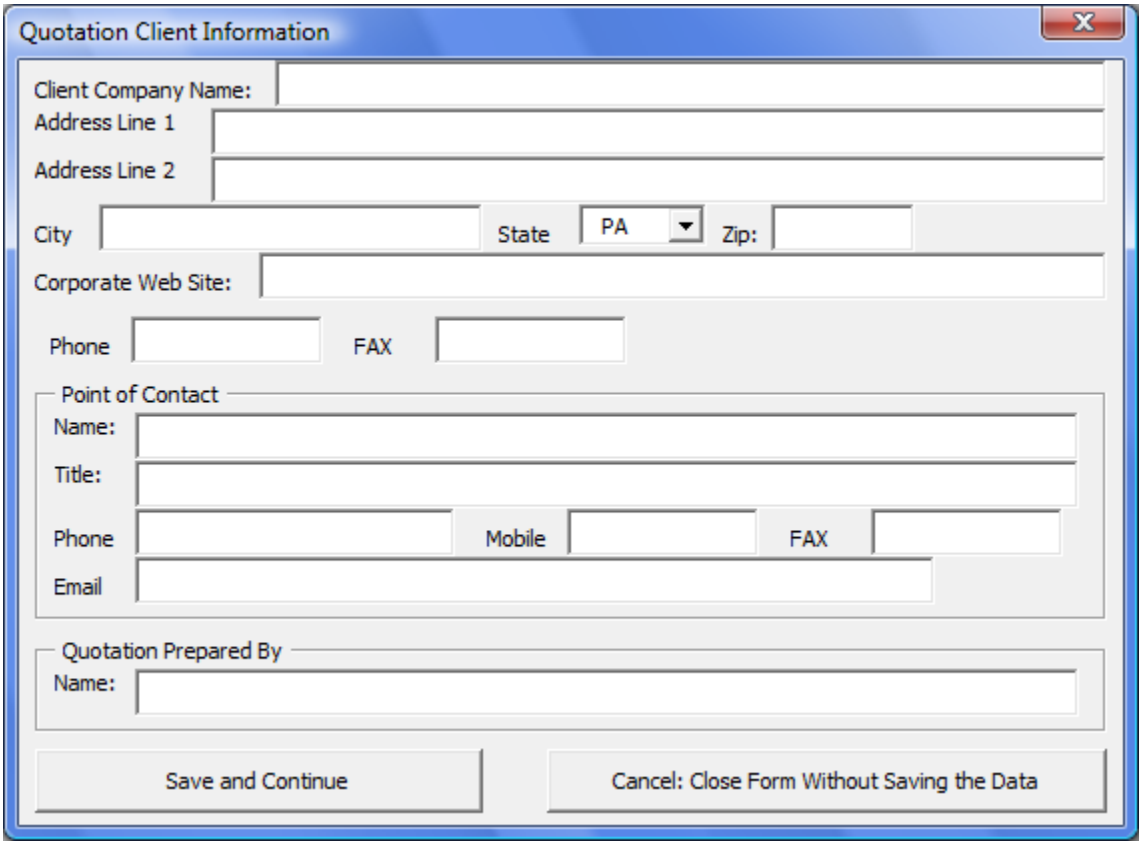


Figure 21 UserForm With Text Entry Boxes

Each of the text entry boxes on this form has a unique name, just as each of the controls on the other form do. That allows us to work with them in the VBA code. Actually on this form there's not much done in the way of serious data validation because most of the entries are things that we pretty much don't have an idea of what they should look like. But we can check to make sure that something was entered into any areas that we consider mandatory information, and we might check that the zip code looks like a zip code or that phone numbers look like what we expect a phone number to look like ... or not. But even that can get almost out of hand. Unless we tell the user how to enter a phone number somewhere, it might come to us several ways, like:

(800) 555-1212 or 800.555.1212 or 800 555-1212

or others. So I'm just taking it on faith that the user has enough common sense to enter phone numbers in some fashion that is acceptable and understandable to others that may look at the information later on.

On this form nothing much happens until you click one of the two buttons at the bottom of the form. If you click the [Cancel: ...] button, well, I just close the form and that's that. But if the user clicks the [Save and Continue] button, then there is work to be done: we have **to take the information from this user form and put it someplace more permanent**. In this case it is going to be moved to a sheet in the workbook.

There is a sheet in the workbook named ***SysSheet*** that is used to store information like this until the user finally tells the program to build a quote for a customer out of all the information that's been entered. We start putting the client's information from this form onto that sheet at cells A2 and B2, with column A being a description of what's in column B. Hopefully the names I gave the text boxes on the userform are as informative as I hoped they would be when I created it and you'll be able to see which ones are being moved onto the worksheet in the code. I've not shown all of the code here because there are a lot of text boxes...

```
Private Sub cmd_SaveData_Click()

  ThisWorkbook.Worksheets("SysSheet").Range("A2") = "Client:"
  ThisWorkbook.Worksheets("SysSheet").Range("B2") = Me!txt_ClientCompany

  ThisWorkbook.Worksheets("SysSheet").Range("A3") = "Address 1:"
  ThisWorkbook.Worksheets("SysSheet").Range("B3") = Me!txt_Address1
'
'and it goes on and on through all of the text boxes until it gets finished and then
'it tells the user that things seem to have worked out well and after that
'it removes itself from memory with the Unload Me statement below

  MsgBox "The information has been saved."
  Unload Me

End Sub
```

Although I did not have to be as specific as I have been in this code, it definitely tells exactly what to do with what:

ThisWorkbook. is optional usually. However since the person may be working on several quotations in several workbooks, I use *ThisWorkbook* to tell VBA that I mean the worksheet named SysSheet that exists in the same workbook that this code is being executed in.

Similarly, the *Me!* associated with the names of the two text boxes used in the example tells VBA not to be confused by any other forms that may be open or any text boxes that it may see laying around that have those same names.

Some of you may ask to please explain a little about how controls like the ComboBox, checkboxes and option buttons in the first userform are referenced or tested in code. So I'll hit them each quickly and then we'll move on.

Checkboxes and Option Buttons usually have one of two possible conditions: TRUE (has an x or check in it or the button has a dot in the middle) or FALSE (checkbox is empty and same for the circle of the option button). So you can write code like this:

If checkboxIncludeLadder = True Then

or

If optionButtonChoose1HPEngine = True Then

and take appropriate action based on the results of those kinds of tests.

As for the ComboBox, it has a couple of properties that can be used. You can use its .Text property to get whatever selection was made in it verbatim. Or you can use its .ListIndex

property to find out which item in the list was selected. The ListIndex values start at zero, so if ListIndex = 0 it means that they chose the first item in the list. If they did not choose an item in the list, the ListIndex value is a negative 1 (-1).

So How Do I Display or Remove a UserForm from the Screen?

To present a userform on the screen, you .Show it. Somewhere in your code you'll need a line that uses the name of the form that you give it during design along with the .Show method, as:

UserForm1.Show

or

GetCustomerInformationForm.Show

You can simply write a macro to do it if you need to:

```
Sub ShowCustomerInfoForm()
        GetCustomerInformationForm.Show
End Sub
```

There are two ways to remove a form from display. The code I presented earlier uses

```
Unload Me
```

which completely removes the form from memory. This has the side effect of also removing all information that was entered into it at that time. I could write that line as Unload Me because it was executed from within the form's code module. If I had needed to do that from some other section of code I could have written it as:

```
Unload GetCustomerInformationForm
```

But you can simply hide the form from view which keeps it in memory and retains the information that was last placed on it. Two ways of doing that:

First, from within the form's own code segment:

```
Me.Hide
```

Second, in some other code segment

```
GetCustomerInformationForm.Hide
```

And that's how you deal with UserForms. That is not to say that it is all that can be done with forms and controls on them. Remember that this is an *Introduction* to things, not a definitive bible covering every aspect of every possible command, object, function and feature in VBA or Excel. I actually used different code in my project to move the data from the form onto the worksheet, but what I wrote above will work and hopefully was easy for you to understand at this point in the tutorial.

DATA FROM WORKSHEETS: A STUDY

Within VBA you can get data from any cell or group of cells on any worksheet in any open workbook. The studies here show how to access that kind of data using user defined objects that represent the other workbooks, worksheets and ranges of cells on them. This method has some distinct advantages:

- It's FAST! You're working with in-memory representations of those objects and there simply isn't anything faster going on in your computer than memory accessing.
- It is neat. Because you are working in memory, there's no need to actually jump around in within Excel selecting various workbooks, worksheets and cells. Because you can get to these directly in memory, there's no distracting (and slow) flickering of the screen as you manipulate the data.

Project 1: Copy Between Workbooks

Ok, this one is more my project than yours – there's no work for you to do except examine the code and observe the results. The project consists of two workbooks:

Project01_WB01.xls and **Project01_WB02.xls**

They are available by clicking the appropriate link (right-click and choose Save Target As) on this page:

http://www.jlathamsite.com/LearningPage.htm

The code is all in Project01_WB01.xls (WB01) while the other workbook, WB02, contains data that we want to move into WB01. There are three text boxes on the first sheet of WB01 that are associated with VBA code. The first one activates a macro that was recorded while individually copying each of the data items from WB02 into the 3rd worksheet in WB01.

To observe the difference in performance between the recorded macro and the custom code in WB01, first try clicking the "Step 2" button. The object at this time is to simply see how long it takes to copy a total of 167 entries from the WB02 into WB01. Obviously, both workbooks must be open for this to take place.

The Step 2a and Step 2b 'buttons' each run a version of custom written VBA code to achieve the same results. The only difference between the two procedures is that one of them has a slight, but probably not humanly noticeable speed advantage over the other. But they definitely have a visible speed advantage over the recorded macro.

Other things to notice when examining the code in WB01 is that the recorded macro is absolutely not robust or versatile: add another item of information in WB02 and it won't get copied over into WB01; delete an item that's already in WB02 and you end up with a 'hole' in the information transferred over into WB01.

So now we see that not only is the custom code faster and more compact than the recorded macro, but it is also more robust. It needs no further attention or maintenance no matter what changes you make to the information in WB02.

The custom written VBA code demonstrates how to use objects in VBA to reference information in an entirely separate workbook, and can easily be adapted to work for you with the same workbook or across more than just two workbooks.

DATA FROM TEXT FILES: A STUDY

Project 2: Importing Data from a Text file

Often you don't have the luxury of working with another Excel file. But many applications have the ability to either be saved as, or to export their information to what is known as an ASCII text file. You're probably used to seeing them as .TXT files and they can be opened and read easily with a program such as Microsoft's Notepad. Sometimes they are rather specially formatted ASCII files that you see as .CSV files. CSV stands for Comma Separated Values. Actually several different characters may be used besides a comma to separate groups of values, but the name from the original use of the comma has stuck with them. Excel has built-in features to import data from .CSV files, but other text files may not conform to those standards and you may want to import those into Excel and nothing but custom code will do the trick for you.

Ok, this one is more my project than yours – there's no work for you to do except examine the code and observe the results. The project consists of one Excel workbook and a text file with sample data in it:

Project02.xls and **Project02DataFile.txt**

They are available by clicking the appropriate link (right-click and choose Save Target As) on this page:

http://www.jlathamsite.com/LearningPage.htm

There are some useful snippets of code to take note of and possibly save for reuse. The module named GetFilenameCode contains a routine that opens up the file browser window and will return the full path and name of a file you select to the calling routine. This is definitely handy, reusable code. I blatantly plagiarized that code from

http://www.cpearson.com/excel/GetFileName.aspx

No sense in reinventing the wheel unless you figure a way to make it turn faster and easier. Chip's website, and others, provide fantastic resources like this at no cost to the user. But I do believe that credit is always due to the benefactor, so Chip gets the plug from me along with my gratitude for providing the code.

Within the ReadTextFileCode module, in the ReadATextFile process, there is definitely one line that deserves some detailed discussion:

```
ActiveSheet.Range("A" & Rows.Count).End(xlUp).Offset(1, 0) = oneTextLine
```

The ActiveSheet is the one sheet that is currently selected in Excel. There can only be one ActiveSheet at any given time.

The .Range("A" & Rows.Count).End(xlUp) portion of the command says look in column A beginning at the last possible row and go up the column until you find the end of cells that match the general character of that cell. The 'general character' being either empty or not empty. The assumption here is that the last cell in the column is empty, so the command is going to find the last cell in the column that is not empty. If the entire column is empty, it will return 1 for "the first row is the end of this section".

The .Offset(1, 0) portion says that once you've found the end of the list, move down 1 row in the same column. So this points at the next empty cell in the column, or if the column is entirely empty, it points to row 2 of the column.

This is a very handy function that is used often to find the next available cell in a column or even the next available row on a worksheet. It's fast and it's effective.

The ReadATextFile() process gives you the basic tools for identifying, opening and reading a text file. You can add more code within the loop that tests for EOF to further process the lines of data read from the file and process it as needed.

Programming With Excel Objects

Somehow I've got to give you a good understanding of what's going on when you program using references to objects rather than with the objects themselves. This method of working with the objects in Excel such as Worksheets and cells, or even with multiple workbooks, is much faster than working with them directly, and offers a lot more flexibility for you.

Let's try a couple of examples and hope I get the idea across.

Example 1: Telephone numbers. You're going to run errands today and you know that along the way you need to make several phone calls. You have some choices on how to 'remember' the phone numbers you need:

You can open the phone book, look up each number and enter the information into your cell phone contacts. You've just created a 'reference' to the numbers in the phone book.

Not wanting to take the time to punch in names and phone numbers into your cell phone contacts list just so you can delete them later, you grab the same phone book, a sheet of paper and a pen and write the information down on the sheet of paper. Again, you've created a reference to the numbers you'll need.

You can drag the entire phone book with you and go through the process of looking up each number as you need it later on. A bit cumbersome and someone back at the house might want the phone book for some other reason anyhow.

Hopefully it's obvious that either one of the reference lists you've created will provide you easier and faster access to the information you need than going to the phone book and looking each one up later.

Example 2: Credit Card Info. You're on your way to apply for a loan and know that they're going to ask for current account information such as account numbers, monthly payments and balance. Again you have some choices:

You can gather up a big stack of most recent statements and scurry off to fill out the application with them, or you could grab the trusty not-very-high-tech sheet of paper and pencil again and just write down the simple facts you know you will need. Voilá!, a reference to the actual data.

In Excel, your sheet of paper is the computer's memory and your pen or pencil is the **Set** command.

ADVANTAGES OF USING OBJECT REFERENCES

I've already mentioned a big one: speed. Performance improves dramatically when you reference these 'in-memory' objects than if you use more direct methods of coding to work with them.

By working with the objects in memory, you often prevent having to select different sheets and cells on them and display the updated data - this alone is a big time saver because updating the displayed workbook/worksheets/cells is a big time user.

You don't have to actually physically "select" an object to work with it! You may even make reference to worksheets that are hidden and to the data on them without having to unhide it and select it and then start selecting cells one by one or in groups on that sheet. Consider the following code:

PERFORMANCE IMPROVEMENTS USING OBJECT REFERENCES

The following is actually an example of the way I once worked through columns of data on worksheets in my earlier days of programming Excel. Before I learned of the wonders of using Objects. This code simply looks in column A of a sheet until it finds an "X" (or "x") in that column, or until it encounters an empty cell. It presumes that there are no empty cells in column A until the end of the list.

```
Sub Find_X_InColumnA_OnSheet1()
  Const SeekValue = "X"
  Dim startTime As Date
  Dim endTime As Date

  ThisWorkbook.Worksheets("Sheet1").Select   ' wasted time, screen flickers, and
                                             ' you won't return to where your user was
  Range("A1").Select ' a little more wasted time
  startTime = Now() ' for timing the test
  'now we really annoy the user by 'scrolling down the worksheet in column A
  Do Until IsEmpty(ActiveCell)
    If Trim(UCase(ActiveCell)) = SeekValue Then
      Exit Do ' we found first "X" or "x"
    End If
    ActiveCell.Offset(1, 0).Activate ' move to next row
  Loop
  endTime = Now()
  MsgBox "It took " & Format(endTime - startTime, "ss") & " seconds to find X"
End Sub
```

It's difficult to see the difference in performance of that method than with others without data to test with. So you can use the **Project03_ObjectReferenceBenefits.xls** file to get some test data into a workbook and run the above code (already in that workbook), along with the variations of it I'm about to present to you to actually see the differences in performance. But if you want to tough it out on your own, I'll provide all of the code here and you can copy and paste into your own workbook, just make sure that there's a "Sheet1" in it.

As usual, the file is available by clicking the appropriate link (right-click and choose Save Target As) on this page:

http://www.jlathamsite.com/LearningPage.htm

Here is some code to fill Sheet1 with lots of entries to test with:

```
Sub FillSheet1()
  Dim LC As Integer
  Dim IL As Integer
  Dim myTestSheet As Worksheet
  Dim baseCell As Range
  Dim rowOffset As Long

  Application.ScreenUpdating = False
  Set myTestSheet = ThisWorkbook.Worksheets("Sheet1")
  myTestSheet.Cells.Clear ' delete any info on the sheet
  Set baseCell = myTestSheet.Range("A1")
  For LC = 65 To 90 ' values for A to Z
    For IL = 1 To 1000 ' 1000 rows worth of each letter
      baseCell.Offset(rowOffset, 0) = Chr$(LC)
      rowOffset = rowOffset + 1
    Next
  Next
  Set baseCell = Nothing
  Set myTestSheet = Nothing
End Sub
```

GPP #2: Use Application.ScreenUpdating = False to improve performance. This command tells Excel to hold off on actually sending updated data/changes to the screen. I've seen the use of this command improve performance as much as 10 times without any other changes to the code at all. Think about it: work done in 1 second instead of 10, or even 1 minute instead of 10 minutes.

Application.ScreenUpdating = False is almost a 'set it and forget it' command: When the end of your Sub is encountered, Excel will automatically turn screen updating back on without any action or code from you at all. Within a Sub it will remain in effect until you either exit the Sub or you give a Application.ScreenUpdating = True command.

There's one catch to that automatic reset of screen updating - if your Sub calls other Subs, then it will be turned back on when one of the other Subs exits unless you remember to set it back to False after making the call(s).

Now let me prove to you that this really works. Here's our first search for X done just the same way, but without updating the screen as Excel works through all of the cells on the sheet.

```
Sub FindWithoutScreenUpdating()
'probably 4 to 10 times faster than Find_X_InColumnA_OnSheet1() was
  Const SeekValue = "X"
  Dim startTime As Date
  Dim endTime As Date

  ThisWorkbook.Worksheets("Sheet1").Select ' wasted time, screen flickers, and
                                           ' you won't return to where your user was
  Range("A1").Select ' a little more wasted time

  Application.ScreenUpdating = False ' THE time saver here!
  startTime = Now() ' for timing the test
  'now no more annoying the user by
  'scrolling down the worksheet in column A
  Do Until IsEmpty(ActiveCell)
    If Trim(UCase(ActiveCell)) = SeekValue Then
      Exit Do ' we found first "X" or "x"
    End If
    ActiveCell.Offset(1, 0).Activate ' move to next row
  Loop
  endTime = Now()
  'because we didn't update the screen, the current active
  'cell is not 'visible', so we need to pull it up into view
  Application.Goto Range(ActiveCell.Address), True
  MsgBox "It took " & Format(endTime - startTime, "ss") & " seconds to find X"
End Sub
```

The time displayed in the message box should be MUCH! less than in our first attempt, even though the only real changes we made were to turn off screen updating, which in turn 'forced' us to use the Application.Goto command to bring the cell we found up into view. But think about it -- we have added instructions to the code and yet we still got a rather impressive performance improvement.

It just doesn't get much better than that. However, "much" is not "any", and we can improve the performance more by using Object references rather than directly moving from cell to cell. We'll see the code for that on the next page.

Here's the almost magical code that manages to improve performance even more than we've seen with Application.Screenupdating = False alone. We have to add some variables and do some setup that we haven't yet done, but just as with FindWithoutScreenUpdating(), even though we've added code, we've still improved performance.

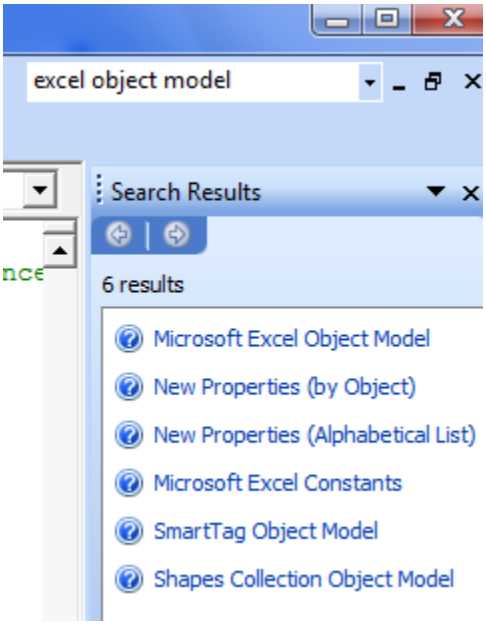
```
Sub FindUsingObjects()
'Just slightly faster than FindWithoutScreenUpdating(), but because
'the timer only has 1 second resolution, you may not 'see' the difference
'but when doing complex operations, this method definitely pays off.
  Const SeekValue = "X"
  Dim myTestSheet As Worksheet will represent 'Sheet1' but in memory.
  Dim seekInRange As Range ' will be used are of column A
  Dim anyCellInSeekInRange As Range ' individual cells in the range
  Dim foundAtCell As String ' to remember where we found the match at.
  Dim startTime As Date
  Dim endTime As Date

'this Set creates an in-memory reference to sheet 'Sheet1'.
'We actually don't even have to choose/select/activate 'Sheet1'
'for this code to work -- except at the very end where we use
'Application.Goto - that requires that we be on Sheet1 if we
'want things to work right for that part of the test.
  Set myTestSheet = ThisWorkbook.Worksheets("Sheet1")
  Application.ScreenUpdating = False
  startTime = Now() ' for timing the test

'this Set will assign the range from A1 down to the last row in column A
'that has any entry in it.  We don't have to worry about empty cells in
'the middle of the list any more!
  Set seekInRange = myTestSheet.Range("A1:" & _
   myTestSheet.Range("A" & Rows.Count).End(xlUp).Address)
  'now we work through the individual cells in memory
  For Each anyCellInSeekInRange In seekInRange
    If Trim(UCase(anyCellInSeekInRange)) = SeekValue Then
      'remember the address of the cell with the X in it
      foundAtCell = anyCellInSeekInRange.Address
      Exit For ' we found first "X" or "x"
    End If
  Next
  endTime = Now()
  'because we didn't update the screen, the current active
  'cell is not 'visible', so we need to pull it up into view
  Application.Goto Range(foundAtCell), True
  MsgBox "It took " & Format(endTime - startTime, "ss") & " seconds to find X at " & foundAtCell
End Sub
```

I hope that all of this convinces you of the advantages of working with Object references to objects in Excel. If it doesn't, then you probably might as well stop reading right here, because you are going to see a lot more of them as we continue onward.

Keep in mind that any object that you can work with directly in VBA can be referenced by one of these "in memory object references". This includes things like various shapes, controls, queries, charts, chart elements, etc. But the end result of it all is that modifications you make to those in-memory object representations are applied to the actual "see it on the screen" object they represent, or to the unseen object (as hyperlinks or queries) in the workbook.

THE EXCEL OBJECT MODEL AS A REFERENCE

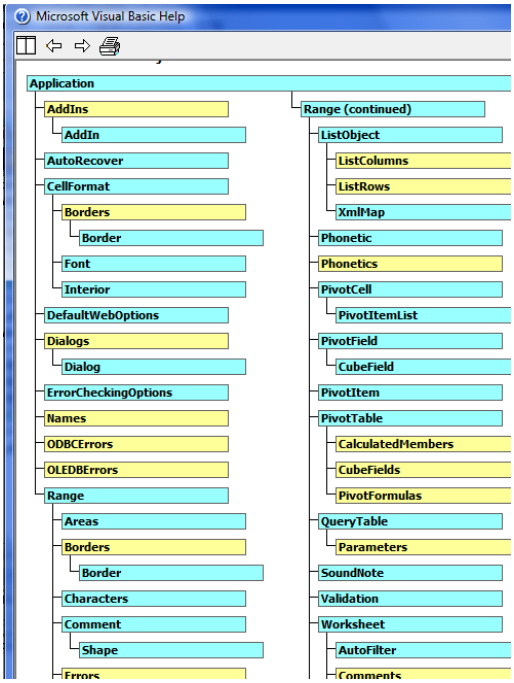
So how do you learn to reference these objects or what their 'family' (technically a 'Collection' in Excel) is? There are a couple or three of ways to do that. One of the more direct methods is to use the Excel Object Model as a reference. So, where the heck is it at?



Once again, Excel Help actually comes to the rescue. If you open the VBE and use it's Help feature and type in Excel Object Model as the search criteria, it will (should?) provide you with a link to the Object Model for your version of Excel.

Remember that you must be in the VB Editor and use its Help/search feature. You won't find this with the regular Excel help/search tool.

Below is just a small example of what the Excel Object model looks like.



But even the Object Model is not always intuitive to use. If you look at the one from Excel 2003 (as depicted here), it's not intuitive that a Worksheet is a member of Worksheets. So sometimes a little stabbing around in the dark has to be done. Or asking for help in one of the Microsoft discussion forums!

Another way to get a quick 'skeleton' for the code you need to piece together is to simply record a macro while doing what you plan on getting done in your code. You can then adapt the code to be more versatile and robust (and more efficient) than the macro you recorded. It is also a good way to find out which Methods (actions) and Properties (attributes) of the objects you are going to work with that you will need to use in your own code.

GPP #3: Don't be too proud to ask for help. If you find yourself in deep waters and can't seem to figure out what it is you need to work with or what to do with what you have found, then by all means ask for help.

Help sources can range from Excel's own "Help" tool, either in Excel or in the VBE to asking for assistance in any number of very good on-line discussion communities, or just searching for examples of code on the internet. I often find myself recording a macro to perform an operation that may have lots of parameters that I refuse to commit to memory just to refresh my memory on how the command should look. Sort is one of those.

Recently (as of this writing) I was faced with the problem of needing to build some user forms that could reference an indeterminate number of selections by the user. And each new group of controls needed to take certain action based on the user's choices on the form. I was kind of lost - in my experience you set up a user form with controls that had a pretty definite or finite count and you built the code for each of them as part of the user form itself. I finally threw my hands up in the air, shouted "I Surrender" and asked if anyone had any ideas of how to get the job done. One of my fellow Excel MVPs came to my rescue almost immediately and showed me how to use Class Module coding to overcome the problem. Nobody knows everything, but given a large enough group then almost everything is known by someone. All you need to do is be able to get in touch with that someone!

Programming with Named Ranges

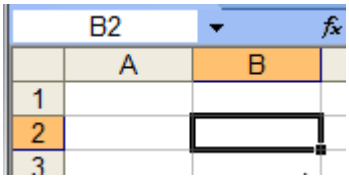
DEFINING A NAME

Naming Directly on a Worksheet

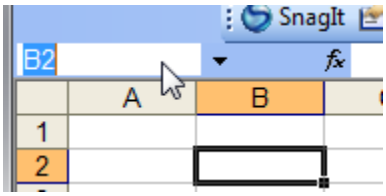
Excel gives you the ability to assign a 'meaningful name' to a cell or group of cells (each of which is known as a *range* in VBA). Assigning a name is easy to do:

Step 1: choose the cell or group of cells to be named,

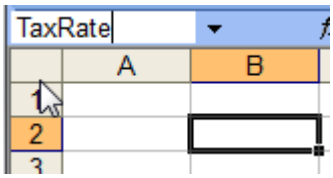
Step 2: enter the name for the range in the 'Name Box' and press [Enter]



Here we have chosen cell B2 on a worksheet. It's address, B2 is shown in the Name Box.



Next simply click inside of the Name Box to get ready to assign a name to cell B2.



Finally, type in the name 'TaxRate' and press the [Enter] key. **NOTE:** you must terminate the name entry with the [Enter] key. If you don't, then the name is not accepted by Excel as a name.

After this is done, you can now refer to the contents of cell B2 on this sheet using its *TaxRate* name. For example, you could refer to it in a formula in the workbook like this:

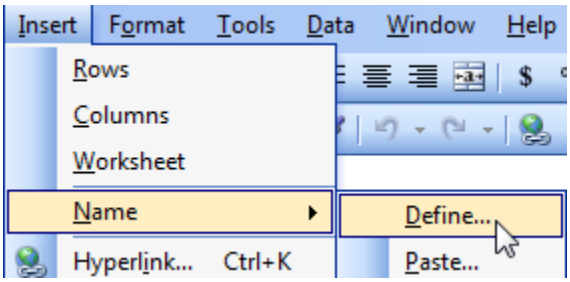
=1.99 * TaxRate

and the cell would show the result. Assuming there is a value of 0.875 in B2/TaxRate, then the formula would return $0.17 (also assuming the cell with the formula in it formatted to display currency).

You can also assign a name to a range of cells to make referencing the group easier and 'maintenance free' in your formulas and other worksheet functions.

You may define names using the Name Manager.

In Excel 2003 and earlier, you get to the name manager through Insert --> Name --> Define



This brings up the Define Name dialog in Excel 2003:

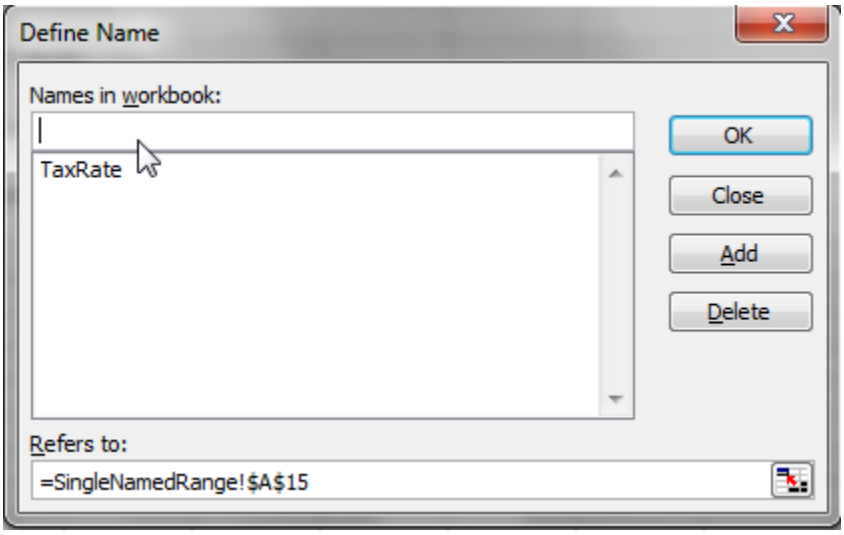
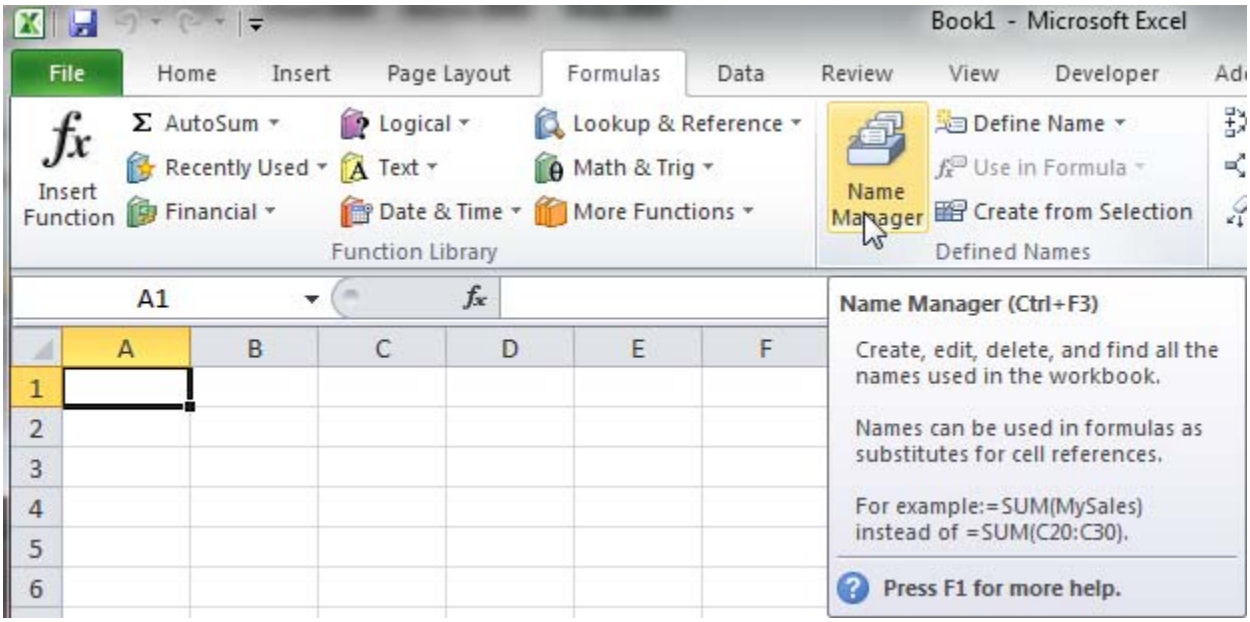


Figure 22 - Define Name Dialog: Excel 2003

You can manage existing names or create new ones using this dialog.

Excel 2010 gives us a more versatile tool, the Name Manager to perform these functions.

In Excel 2010, you access the Name Manager from the Defined Names group on the [Formulas] tab:



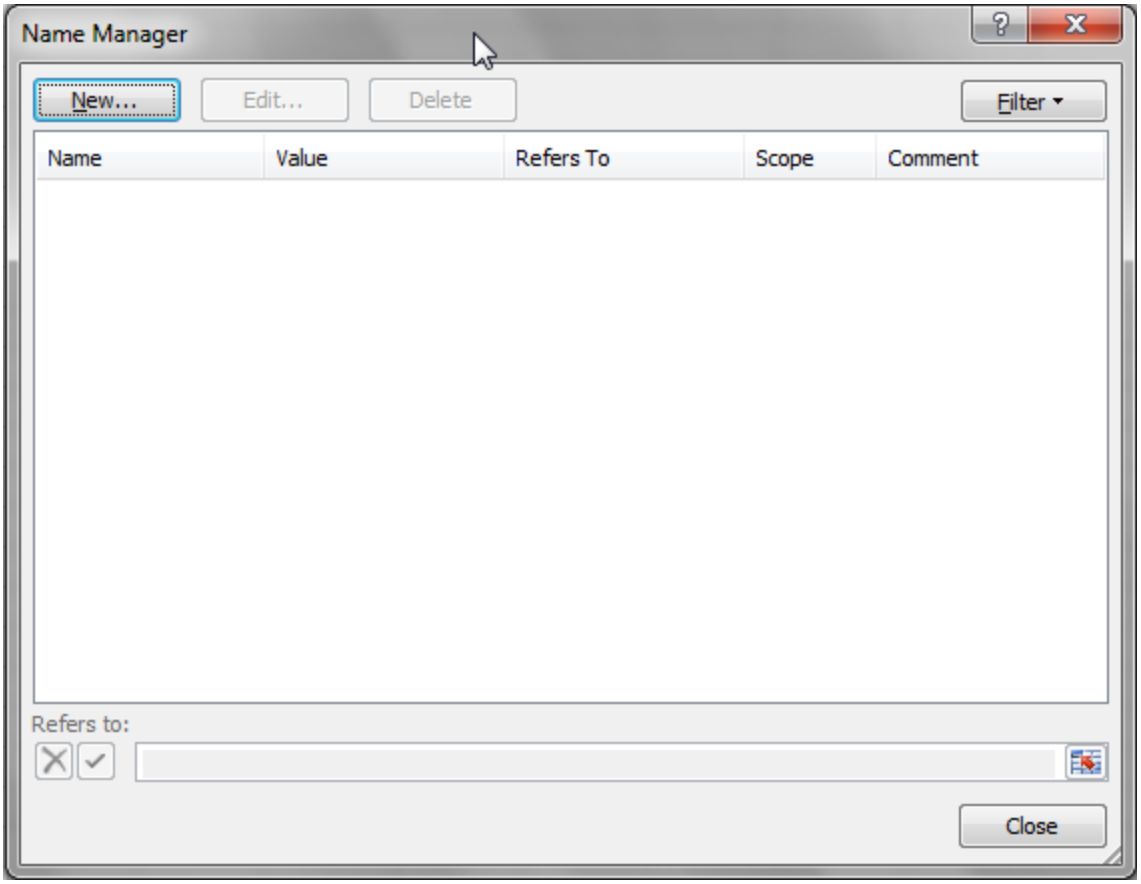


Figure 23 Name Manager: Excel 2010

With the Define Name or Name Manager, you can remove old name definitions, add new ones and even change the address of the cell(s) that a name refers to.

Using a Named Range for a List

Normally when you use a list of entries in a group of cells as the source for Data Validation controlled cell entry, the list must be on the same worksheet with the cell using it as its list. But if you have given a name to a list of cells, that list can be on another sheet in the workbook. This gives you the ability to put several lists on a worksheet that you can even keep hidden.

Code Snippets and Examples

In this section I'll try to present some useful routines that can be used and reused in your own coding efforts. The sources for these range from web sites, help forums and my own experiences. Sorry, but they're really not organized in any specific way, just as I have come across them in trying to gather up content for this.

SORTING A RANGE

See the VB Help topic for SORT for all the details.

In setting up for a sort you need to keep in mind that you will be sorting a range of cells, and that the various sort keys must also be ranges or references to ranges.

The setup – the variables we will need (or just want) get declared first:

Setup for single field (column) sort, which will sort an area that includes A1 to F1000 on Sheet1. Row 1 contains labels (headers) and we will sort in ascending order based on column C.

```
Dim myWorksheet As Worksheet
Dim theSortRange As Range
Dim theSortKey as Range
Set myWorksheet = ThisWorkbook.Worksheets("Sheet1")
Set theSortRange = myWorksheet.Range("A1:F1000")
Set theSortKey = myWorksheet.Range("C1")

theSortRange.Sort Key1:=theSortKey, Order1:=xlAscending, Header:=xlYes, _
    OrderCustom:=1, MatchCase:=False, Orientation:=xlTopToBottom, _
    DataOption1:=xlSortNormal
```

Note: the 'DataOption#" parameter is only valid in Excel 2003 and later versions. Leave it off of the command if the sort will be used in an earlier version [it will still work in 2003 and later without it]

Setup for three field (column) sort, which will sort an area that includes A1 to F1000 on Sheet1 where row 1 contains labels (headers) and we will sort in ascending order based first on column C, then on column A and finally in descending order of column F.

```
Dim myWorksheet As Worksheet
Dim theSortRange As Range
Dim theSortKey1 as Range
Dim theSortKey2 as Range
Dim theSortKey3 as Range
Set myWorksheet = ThisWorkbook.Worksheets("Sheet1")
Set theSortRange = myWorksheet.Range("A1:F1000")
Set theSortKey1 = myWorksheet.Range("C1")
Set theSortKey2 = myWorksheet.Range("A1")
Set theSortKey3 = myWorksheet.Range("F1")

theSortRange.Sort Key1:=theSortKey, Order1:=xlAscending, Order2:=xlAscending, _
    Order3:=xlDescending, Header:=xlYes, OrderCustom:=1, OrderCustom2:=1, _
    OrderCustom3:=1, MatchCase:=False, Orientation:=xlTopToBottom, _
    DataOption1:=xlSortNormal, DataOption2:=xlSortNormal, DataOption3:=xlSortNormal
```

Note: the 'DataOption#" parameter is only valid in Excel 2003 and later versions. Leave it off of the command if the sort will be used in an earlier version [it will still work in 2003 and later without it]

FIND THE LAST USED CELL IN A COLUMN

This is an operation you'll probably use over and over in your coding to find the end of a range of data. It assumes that your data starts at the top of the sheet (row 1) or another designated row and continues down the sheet but may include empty cells in the rows.

It's useful when you have data on a sheet that is dynamic and that may have a different number of rows in it at any given time.

First thing to do is to identify a column that will have some information in it in the last possible row of data. For this example, we will assume that column A fills that need.

Identify the Last Used Row

```
Dim lastRow As Long
lastRow = ThisWorkbook.Worksheets("SheetName").Range("A" & Rows.Count).End(xlUp).Row
```

Now, that wasn't all that difficult, was it?

You've probably noticed that I keep specifying "ThisWorkbook" – you don't have to do that if you know what workbook will be active when your code runs – the currently active workbook will be the one used to determine where everything else is. But if there is any doubt, then specify the workbook either by name or by using ThisWorkbook. 'ThisWorkbook' means the workbook that the code is contained in.

So that line of code says "look in the same workbook that this code is located in, on a sheet named SheetName, and in column A --- and here's the trick: start looking at the cell that is on the last possible row on the sheet (Rows.Count returns a number that is the maximum number of rows permitted on a sheet for the version of Excel that is being used). **Assumption** is also made that there is not a value in that last cell – that it is empty. The End(xlUp) part of says to look from the referenced cell ("A" and the last possible row) upward until a cell is found that marks the end of the section that's like that referenced cell. So if A65536 (Excel 2003) is empty, it will look up until it finds a cell with something in it – even just a formula that returns "" (empty string).

So that command puts the row number of the "last used cell" in the specified column into variable lastRow.

Identify the Next Available Row

There are a couple of ways to modify what we just did to find the first empty cell in the column of data.

Method #1: Take the value of lastRow and add 1 to it. That would be the row number of the next available empty cell in the column.

Method #2: make the addition part of the statement itself.

```
lastRow = ThisWorkbook.Worksheets("SheetName").Range("A" & Rows.Count).End(xlUp).Row + 1
```

You can even modify the statement with the .Offset() option to do it:

```
lastRow = ThisWorkbook.Worksheets("SheetName").Range("A" & Rows.Count).End(xlUp).Offset(1, 0).Row
```

FIND THE FIRST EMPTY CELL IN A COLUMN

In this case you may have a column of data that has empty cells in it and you want to find the first row with an empty cell in it. Hopefully it is obvious that if there are no empty cells in the data list that the first empty cell would be the one just below the last entry in the list. The command looks much like our previous command except that it looks *down* from row 1 in the column:

```
Dim firstEmptyCell As Long
firstEmptyCell = Worksheets("SheetName").Range("A1").End(xlDown).Row
```

GET THE ADDRESS INSTEAD OF THE ROW

You don't have to settle for just the row, you can actually return the address of the cell you find. Simply change the type of the variable that will receive the information from Long to String and change .Row to .Address:

```
Dim cellAddress As String   'the address is returned as text
cellAddress = Worksheets("SheetName").Range("A1").End(xlDown).Address
```

Enough about finding those particular unique entries in a column. Now we'll look at doing much the same thing for rows.

FIND THE LAST USED CELL IN A ROW

This is an operation that you probably won't use as often as the ones for columns, but there are still times you may need to find the last (right-most) used cell on a row.

This is complicated a bit due to the fact that it's a little difficult to find the reference to the column that is the last column in the version of Excel you are using. But we can work around it using the Columns.Count property of the sheet.

```
Sub LastColumnInOneRow()
'Find the last used column in a Row: row 1 in this example
    Dim LastCol As Integer
    With ActiveSheet
        LastCol = .Cells(1, .Columns.Count).End(xlToLeft).Column
    End With
    MsgBox LastCol
End Sub
```

That comes directly from Ron De Bruin's site at: http://www.rondebruin.nl/last.htm which leads me to pretty much stop providing code examples, scratching my head trying to think up code that you might find useful. Instead, the remainder of this document consists of links to various absolutely excellent examples of VBA for Excel coding. In some cases, such as Chip Pearson's site, I've copied the table of contents with links to various Excel solutions on the sites. You can follow those links to those code solutions.

CONSOLIDATING DATA IN A WORKBOOK

One of the best sources of code that may be used to combine data from several worksheets and even workbooks can be found at Ron De Bruin's site:

http://www.rondebruin.nl/copy2.htm

USING A TEXTBOX TO ACCESS A MACRO

Before going on to the next section, I think would be a good thing for you to know of at least one easy way of getting rapid access to macros you create for the user to accomplish tasks with.

Using a TextBox or other shape from the Drawing Toolbar offers a good deal of flexibility. You can set the font size and format and add color to the 'button'. Plus the code can reside in any module in the workbook.

You don't even have to have the macro written in order to create the button in anticipation of associating a macro with it later.

Select a shape from the Drawing toolbar (Excel 2003 and earlier) or from the Insert tab (Excel 2007 and later) and place it on the sheet and size it, add text to it, and format its colors. I also recommend setting its properties to NOT move or resize with cells.

At any point after creating the 'button' you can assign a macro to run when you click it. Simply right click near the edge of the shape and choose [Assign Macro] and then pick the macro to run from the list presented to you. And that's all there is to it.

DOING THE IMPOSSIBLE

There are some things that simply cannot be done using Excel worksheet formulas. Hiding either rows or columns and unhiding them again comes to mind right away. Some other things that you can do from the keyboard such as auto-filtering and removing such a filter can't be controlled through formulas, but they can be done using VBA code.

Providing the user with a 'button' to access these features is a nice touch because it reduces the need to use the keyboard and the 4-click sequence of

Tools --> Macro --> Macros, select a macro and click [Run] to make it all work.

The user will definitely appreciate being able to run the macro with one click, and of being sure to use the right macro instead of clicking the wrong one in what may be a long list of macros to choose from.

Hiding Rows

While you can use Data Auto-Filter to remove rows from view, sometimes it's easier to just hide the rows with code. You need to know what condition you want to use to choose rows to be hidden and what column values meeting the condition can be found.

Let us say that you want to hide all rows that do not have a value entered into column R. Perhaps a row's data represents values for quantities of a product produced on certain dates and you want to see all products that were produced on the date contained at the start of column R. Hiding rows with no entry in column R would clean up the list for you.

Here is the code that would do the trick for you.

Start by UNHIDING all rows! This will make sure that only the rows you mean to hide will be hidden at the end of the process, with no left over hidden rows from other similar actions taken on other columns:

```
ActiveSheet.Columns("R:R").EntireRow.Hidden = False
```

Actually you can pick any column! I just chose to use R since R is the column we are about to examine for empty cells to determine whether to hide a row or not.

Now you need code to work through all possible rows. You need a column that has an entry for every possible row used on the sheet. That is not going to be column R. For argument's sake, assume that column A has product identifiers in it, one for all possible products. So we could examine column A to determine what rows in column R are unused. Here is the code to set a reference to the cells in column R that may have entries in them based on the total number of rows used in column A.

```
  Set testRange = ActiveSheet.Range("R1:R" & _
   ActiveSheet.Range("A" & Rows.Count).End(xlUp).Row)
  For Each anyCell in testRange
    If IsEmpty(anyCell) Then
      anyCell.EntireRow.Hidden = True ' Hide this row!!
    End If
  Next ' end of anyCell loop
```

The complete procedure might look like this:

```
Sub HideColumnRRows()
  Dim testRange As Range
  Dim anyCell As Range
  ActiveSheet.Columns("R:R").EntireRow.Hidden = False
  Set testRange = ActiveSheet.Range("R1:R" & _
   ActiveSheet.Range("A" & Rows.Count).End(xlUp).Row)
  For Each anyCell in testRange
    If IsEmpty(anyCell) Then
      anyCell.EntireRow.Hidden = True ' Hide this row!!
    End If
  Next ' end of anyCell loop
  Set testRange = Nothing ' release resource back to the system
End Sub
```

Unhiding Rows

I've already given away this secret because we started off our Hide Rows routine by unhiding all rows. But we will put it in a sub by itself anyhow. Just remember that this will NOT unhide rows that are hidden because of data filtering - they are hidden in a different manner and will not be unhidden with this code.

You can pick any column on the sheet, since any column includes all rows. For convenience sake, it's probably easiest to simply use column A.

```
Sub UnhideAllRows()
  ActiveSheet.Columns("A:A").EntireRow.Hidden = False
End Sub
```

An Introduction to Debugging

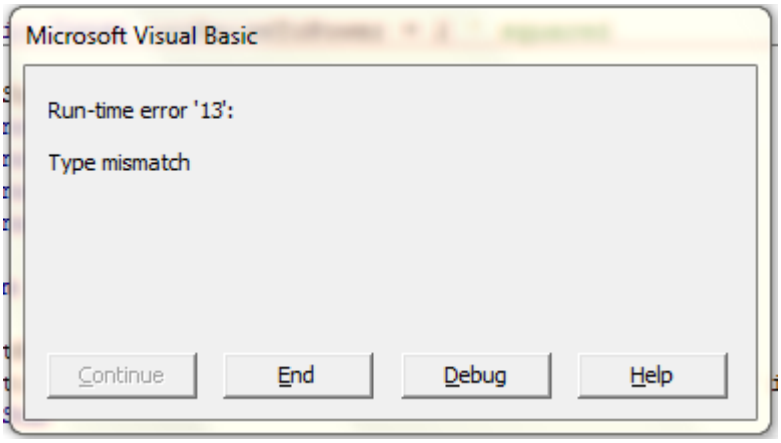
Debugging a project can be a simple "mechanical" process, sometimes it is almost an art. In this section I can only present a few suggestions and at least introduce you to some of the tools available to you to help examine code, determine values during execution and hunt down and squash bugs. We take a simple routine that is giving us problems and see how some of the tools can help us determine what the problem is. How to fix this particular one is left as a thought exercise for the reader: validate information before trying to use it? put in code to ignore the error? repremand the user for entering the wrong kind of information in the first place?

THE PROBLEM EXAMPLE

We have a simple looking routine but it isn't working for us. Unfortunately the programmer wasn't very generous with comments, and now we need to try to figure out just why it isn't working for us.

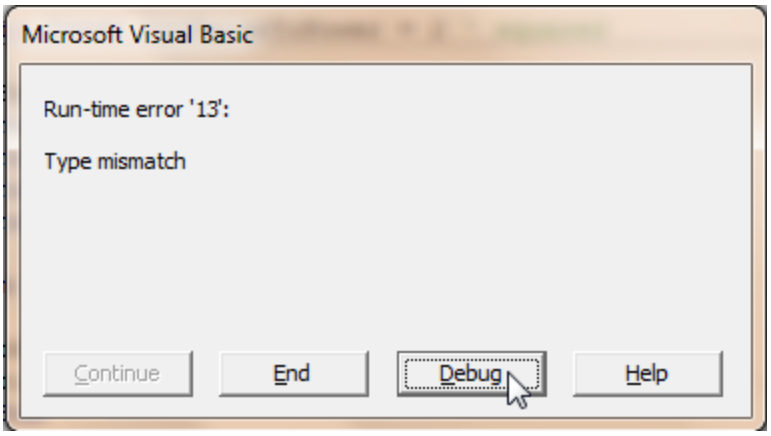
```
Option Explicit
Public Const intRaiseToPower = 2 ' squared

Sub SquareANumber()
  Const getRow = 2
  Const getCol = 1
  Const putRow = 2
  Const putCol = 2

  Dim intMyNumber As Integer

  intMyNumber = ActiveSheet.Cells(getRow, getCol)
  ActiveSheet.Cells(putRow, putCol) = intMyNumber ^ intRaiseToPower
End Sub
```

When we try to use the Sub we get this error:



This tells us that we are trying to perform some operation that requires some specific 'Type' of constant or variable, but what we are trying to use is not one of them and Excel could not trick

the constant/variable into being the proper type. But it doesn't tell us what the problem value is, or even where it is in the code. So what do we do?



What you want to do at this point is to click the [Debug] button. Excel will then automatically open the VB Editor (VBE) and even show you the line of code that it thinks is causing the problem. NOTE: For some problems, Excel can get confused and point to the wrong line of code. But that's something to cover on another day.

When we click the [Debug] button, we get this display:

```
Sub SquareANumber()
  Const getRow = 2
  Const getCol = 1
  Const putRow = 2
  Const putCol = 2

  Dim intMyNumber As Integer

  intMyNumber = ActiveSheet.Cells(getRow, getCol)
  ActiveSheet.Cells(putRow, putCol) = intMyNumber ^ intRaiseToPower
End Sub
```

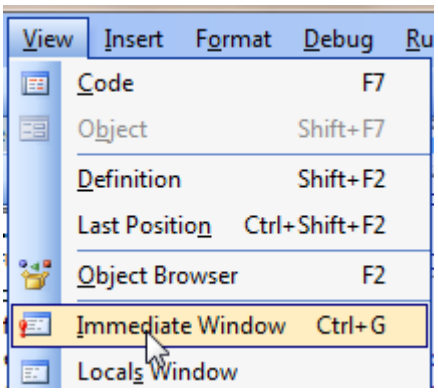
Excel has pointed out the line of code that it feels there is a problem with, now it is up to us to figure out the details.

We probably need to answer 2 or 3 questions on the way to determining the exact problem:

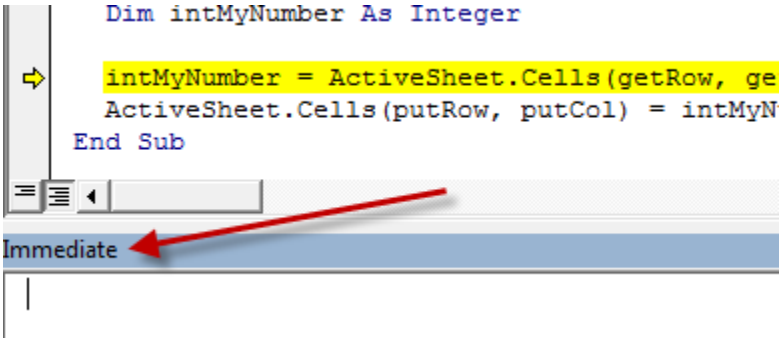
#1 - what sheet is the "ActiveSheet". For experienced users, this is an easy question to answer, but if it was some object set to represent a sheet that might be hidden from view or not currently selected, we need to know which sheet we should be examining.

An important tool you'll need to help pin the problem down and come up with a fix is the ***Immediate window*** of the VBE.

If you do not see a window in the VBE with the title "**Immediate**", then you can bring it into view from the View menu option, or just press [Ctrl]+[G] to make it visible:



Normally it appears right underneath the main window in the VBE:



You can do a lot in the Immediate window: examine values, determine addresses, set values and even issue some commands to Excel itself. For now we want to find out what sheet we are getting a value from, so we type the following into the Immediate window and press the [Enter] key at the end of it:

? ActiveSheet.Name

The **?** is a shortcut entry for "Print" or "Show me..." and is a holdover from the very early days of the original BASIC interpreters.

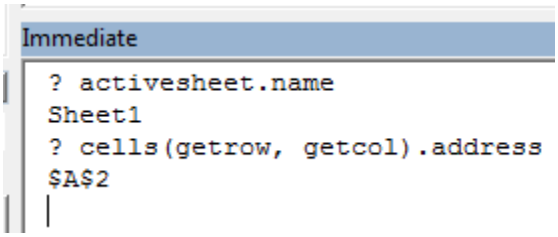
The name of the ActiveSheet is revealed to us as: **Sheet1**. So at least now we know what sheet to look at, but *where* on that sheet do we need to be looking? We know that we should be looking at Cells(getRow, getCol). But where the heck is that? Once again, it is the Immediate window to the rescue. We can get it to show us the address of that cell by typing:

? Cells(getRow, getCol).Address

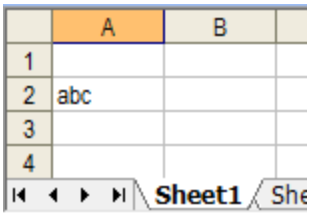
and pressing the [Enter] key.

By the way, you can copy from the code window into the Immediate window, which can help prevent typos from interfering with your debugging efforts.

The Immediate window now tells us that we are trying to get some value from cell A2 on Sheet1:



So we can go look there and see what is in that cell.

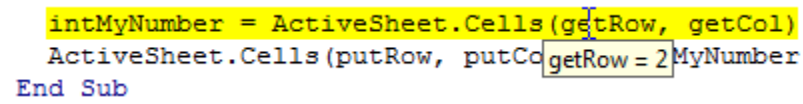


And there we see a problem: Cell A2 holds some text, and since we declared intMyNumber to be type Integer, and so a text value cannot be assigned to a numerically typed variable - and there lies the answer to why we got "Error 13: Type Mismatch" when trying to use the Sub.

Now it is up to you to determine why someone typed text into a cell you were expecting to find a number into (or realize that you should not type text into a cell that is going to be used for some math processing). The Debugger has done about all it can for you, what you do to prevent the problem in the future is a decision you have to make yourself.

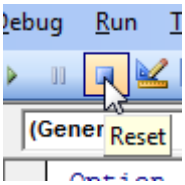
OTHER DEBUGGING TIPS:

Tip #1: You can quickly determine certain values right in the code window by simply moving the mouse pointer/cursor over a constant or variable name and its value will be displayed as a popup tip:



Tip #2: You can end the process by either:

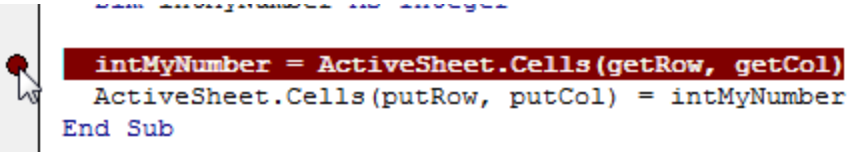
Clicking the Reset button up in the menu area of the VBE:



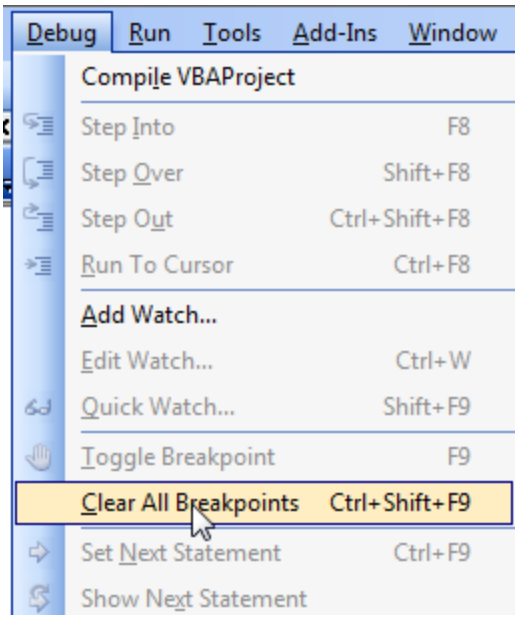
Or you can simply hit the [F5] key which will bring up the error message again and you can click the [End] button on it.

Tip #3: If you run into a rather complicated situation where some values are being calculated or retrieved in the code and you are getting an error on down the line because one of them is incorrect, you can examine the entire process step by step.

You can force code execution to stop by setting one or more "breakpoint"s in the code:



You do that by clicking in the area to the left of the beginning of a line of code. The dark red dot and highlighting will indicate that a breakpoint has been set. You can remove it later by clicking the dark red dot again, or use the [Debug] option in the menu to clear them all:



Another way to achieve much the same thing is to insert the Stop statement into the code where you want it to break into debug mode. Simply type the word Stop on a line by itself in the code - just don't forget to delete it after your debugging is completed.

Once you have halted the code where you want to, you can press the **[F8]** button to then single step through your code and examine values and processes along the way to the problem line of code. When you are ready to let the code run normally again during your debug session, press the **[F5]** key - or you can terminate the execution using the [Reset] button and start all over again.

You'll notice in the last screen shot that there are several ways of controlling what pieces of code are executed associated with the [F8] and [F9] keys. They can all help in your effort to determine the source of a problem, or just to go through code to see how it works.

Additional Excel VBA Resources

This section provides a list of other sites that are simply filled with useful Excel and Excel VBA help. I've copied their table of contents pages where they are available, however, the contents of any site may change at any time – it is the Internet, after all.

Excel MVP Websites

You can find a more complete list of many Excel MVP websites here:

http://www.mvps.org/links.html#Excel

And if you go to the top of that page, you can look through the list for websites belonging to MVPs in many areas:

http://www.mvps.org/links.html

Within these sites is a veritable library of knowledge of Excel; its operation,

RON DEBRUIN'S EXCEL TIPS:

http://www.rondebruin.nl/tips.htm

Excel Add-ins and code for Mailing from Excel

Excel Add-ins page
Example Code for sending mail from Excel

Excel 2007

Where can I find the menu commands in Excel 2007
Create and mail PDF files with Excel 2007
Use VBA SaveAs in Excel 2007
Copy sheet security dialog in Excel 2007
Sheet Direction in Excel 2007
Reverse compatibility problem of the old ATP functions
Macros are disabled when you open password protected workbooks
Shapes and VBA code in Excel 2007
Filtering by the Active Cell's Value, Font Color or Fill Color in Excel 2007
Help: Different Excel file formats and Excel versions
Disable Excel 2003 Menu Accelerators keys in Excel 2007
Table Tools in Excel 2007
VBA code examples for Tables in Excel 2007 or a List in Excel 2003

Excel 2007 Ribbon and QAT pages

Menu for favorite macros in Excel 2007 (for all workbooks)
Menu for favorite macros in Excel 2007 (for one workbook)
Change the ribbon in Excel 2007
Change built-in groups in the Ribbon
Add missing built-in commands to the QAT or Ribbon
Add buttons to the QAT and customize the images of the buttons
Menu in the ribbon with different languages
Dealing with Ribbons and Menus - Avoiding Two Versions
Images on custom Ribbon controls
Galleries in the Ribbon
Hide or Display Custom Ribbon Tab/Group/Control with getVisible

Copy/Paste/Merge examples

Copy to a database sheet on the next empty row
Merge cells from all or some worksheets into one Master worksheet
Create a summary worksheet from all worksheets (formulas with VBA)
Create a link to or Sum a cell in all worksheets (worksheet functions)
Merge data from all workbooks in a folder (1)
Merge data from all workbooks in a folder (2)
Merge data from all workbooks in a folder: Add-in
Create a summary worksheet from different workbooks (formulas with VBA)
Merge data from all workbooks in a folder to a txt file
Copy every TXT or CSV file in a new worksheet of a newly created workbook
Merge all CSV or TXT files in a folder
Copy a range from closed workbooks (ADO)
Copy a range from closed workbook (Local, Network and on the internet)
Copy data from an Access database into Excel with ADO
Change cells or range in all workbooks in a folder
Copy records with the same value in a column to a new sheet or workbook
VBA code examples for Tables in Excel 2007 or a List in Excel 2003
Create a workbook from every worksheet in your workbook
Create separate sheet for each horizontal PageBreak
Copy, Move and Delete files and folders
How do I create/use a sheet template

Delete/Hide/Disable examples

Delete row if a specific value exist
Delete or Hide Objects/Controls on a worksheet
Disable command bars and controls
SpecialCells limit problem
Disable key or key combination or run a macro if you use it

Zip (compress) ActiveWorkbook, Folder, File or Files with VBA code

7-zip : Zip Activeworkbook, Folder, File or Files(VBA)
7-zip : Unzip a zip file (VBA)
Zip file or files with the default Windows zip program (VBA)
Unzip zip file or files with the default Windows zip program(VBA)
WinZip : Zip Activeworkbook, Folder, File or Files (VBA)
WinZip : Unzip a zip file(VBA)

Weeknumber/Dates

Use the Calendar control to fill in dates
Week numbers
ISO Date Representatation and Week Numbering

Help information

Help Context IDs for Excel 2000, 2002, 2003 and 2007
Where do I paste the code that I want to use in my workbook
How do I create a PERSONAL.XLS(B) or Add-in

Other pages

Print tips for Excel
Test if Folder, File or Sheet exists or File is open
Find last row, column or last cell
Change formulas to values
Find value in Range, Sheet or Sheets with VBA

Cleaning "Dirty" Data
Lotus Transition Formula Evaluation Errors
Analysis ToolPak Translator 7.0

DEBRA DALGLEISH'S EXCEL TIPS

WWW.CONTEXTURES.COM

HTTP://WWW.CONTEXTURES.COM/TIPTECH.HTML

A

Advanced Filter - Basics
Advanced Filter - Criteria
Advanced Filter - Different Sheet video
Advanced Filter - Unique Items video

AutoFilter - Basics
AutoFilter - Filter Text in Long String
AutoFilter - Limits to Dropdown Lists
AutoFilter - Programming
AutoFilter - Protected Sheet
AutoFilter - Status Bar Record Count
AutoFilter - Sum a Filtered List

B

Bar over character 31-Oct-06

Beyond the Keyboard

Blank Cells, Fill Video

Blog, Contextures

Book List, Excel 10-Jun-07
Books -- on my bookshelf
Books -- e-books, Microsoft Office 10-Feb-06

C

Charting - Jon Peltier's Site Index
Charting Links

Christmas Planner 21-Nov-08

Code, Copy to a workbook video

Coderre, Ron - Sample Workbooks 17-Jul-07

Column headers show numbers (FAQ)

Combining Data

Comments - Add a Picture video
Comments - Basics
Comments - Change Indicator Colour

F (CONT'D)

Form, Create a UserForm
Form, Print Selected Items 23-Sep-06
Form, Survey 29-Oct-05 updated 11-Oct-06
Form, Worksheet Data Entry 22-Sep-06
Formatting Tips - Move Toolbar Palettes 29-Apr-08
http://www.contextures.com/xlfaqFun.html - FormulaShowFormulas visible on Worksheet 16-Jun-07

Functions
Functions -- Count cells
Functions -- IFERROR 31-Dec-08
Functions -- INDEX 23-Nov-05
Functions -- INDIRECT 11-Nov-06
Functions -- MATCH Video
Functions -- SUBTOTAL
Functions -- Sum cells
Functions -- VLOOKUP Video

G-M

GetPivotData 23-Jan-09

Gift Ideas for Excel Users updated 27-May-09

Govier, Roger - Sample Workbooks 18-May-09

Grades, Convert Percentages to Letter 30-May-09 Video

INDEX Function

INDIRECT Function 11-Nov-06

Keyboard Shortcuts

Macros, Copy to a workbook video
Macros Prompt, Enable or Disable (FAQ) video
Macro Toolbar 24-Dec-05

MATCH Function Video

N-O

Names -- Naming Ranges video
Names -- Naming Dynamic Ranges with a macro 22-Feb-09
Names -- Use Names in Formulas 17-Jun-05
Navigation Command for Sheets 2007 17-Jul-08
Navigation Toolbar for Sheets 2003 21-Dec-05

Comments - Change Shape
Comments - Copy Text to Adjacent Cell 22-Sep-05
Comments - Extract Text to Word
Comments - Format All 09-Mar-06
Comments - Format Text 29-Sep-08
http://www.contextures.com/xlcomments03.html - PictureComments - Insert Selected Picture 28-Jan-06
Comments - Number and List 22-Jan-06
Comments - Printing
Comments - Programming
Comments - Resize
Comments - Show in Centre 22-Jul-06

Conditional Formatting - Basics
Conditional Formatting - Documentation 07-Mar-05
Conditional Formatting - Hide Cells to Print
Conditional Formatting - Hide Duplicates
Conditional Formatting - Hide Errors
Conditional Formatting - Lottery Numbers
Conditional Formatting - Row video
Conditional Formatting - Shade Alt Rows 14-Feb-07
Conditional Formatting - Shade Bands 14-Feb-07
Conditional Formatting - Filtered Bands 23-Feb-07
Conditional Formatting - Coloured Shapes 22-Jun-07

Contextures Blog

Count Cells

D

Data Entry - Tips
Data Entry - Fill Blank Cells Video
Data Entry - Convert Text to Numbers video
Data Entry - Increase Numbers by Set Amount video
Data Entry - Excel Videos

Data Validation - Basics video
Data Validation - Combo box 15-Jan-07
Data Validation - Combo box Named Range 15-Jan-07
Data Validation - Combo box - Click 18-Oct-08
Data Validation - Custom Criteria
Data Validation - Dependent Dropdown- Sorted List 15-Jul-05
Data Validation - Dependent Lists
Data Validation - Dependent Lists INDEX 17-May-09 NEW
Data Validation - Documentation
Data Validation - Font Size, List Length
Data Validation - Hide Used Items
Data Validation - Input Message in Text Box 05Jun-05
Data Validation - Invalid Entries Allowed updated 11-

Newsgroup Posting Statistics -- Annual 01-Jan-09

Numbers, Convert Text to video
Numbers, Increase by Set Amount videohttp://www.contextures.com/xlDataEntry04.html 18-May-08

Order Form 30-Jul-05

P

Paste Values Mouse Shortcut video 04-Feb-09
PeltierTech - Charting Site Index

http://www.contextures.com/Pubn03.htmlPivot Tables, Beginning (Excel 2007)
Pivot Tables, Recipe Book (Excel 2003)
Pivot Tables, Recipe Book (Excel 2007)

Pivot Tables - Add-in -- Pivot Power 29-Apr-05
Pivot Tables - Add-in - Pivot Play PLUS 15-Mar-08
Pivot Tables - Clear Old Items video updated 27-Jun-08
Pivot Tables - Create in Excel 2007 video 19-Feb-09
http://www.contextures.com/xlPivot06.html
http://www.contextures.com/xlPivot10.htmlPivot Tables - Custom Calculations 07-Mar-05
Pivot Tables - Data Field Layout video
Pivot Tables - Dynamic Data Source
Pivot Tables - Field Settings
Pivot Tables - FAQs 09-Oct-06
Pivot Tables - Filter Source Data 19-Jan-09
Pivot Tables - GetPivotData
Pivot Tables - Grand Total at Top 15-May-08
Pivot Tables - Grouping Data
Pivot Tables - Layout, Excel 2007 video 04-Jul-08
Pivot Tables - Multiple Consolidation Ranges
Pivot Tables - Pivot Cache 22-Mar-05
Pivot Tables - Printing
Pivot Tables - Protection 23-Apr-05
Pivot Tables - Running Totals **video** 13-Sep-08
Pivot Tables - Select Sections video 31-Aug-08
Pivot Tables - Show and Hide Items
Pivot Tables - Unique Items
Pivot Tables and Pivot Chart Intro

Q-R-S

Queries - Add-in - Pivot Play PLUS 15-Mar-08

Ranges, Naming
Ribbon -- Navigation Command for Sheets 2007 17-Jul-08
Running Totals, Pivot Tables **video** 13-Sep-08

Oct-06
Data Validation - List from Other Workbook
Data Validation Dropdowns are Too Wide
Data Validation - Make List Appear Larger
Data Validation - Make List Wider
Data Validation - Messages
Data Validation - Missing Arrows updated 11-Oct-06
Data Validation - Order Form 11-May-05
Data Validation - Tips and Quirks updated 11-Oct-06

Dynamic Ranges, Naming
Dynamic Ranges, Naming with a Macro

E

Excel 2007 -- Articles List updated 16-Nov-07
Excel 2010 -- Articles List 01-Jun-09 NEW
Excel Events updated 28-Mar-09 UPDATED
Excel Links
Excel Sites
Excel Store
Excel Table 21-Aug-08
Excel Conference, Advanced

F

FAQs, Excel - Application and Files
FAQs, Excel - Dates and Times
FAQs, Excel - Index
FAQs, Excel - Macros, VBA
FAQs, Excel - Pivot Tables and Pivot Charts 09-Oct-06
http://www.contextures.com/xlfaqFun.htmlFAQs, Excel - Worksheet Functions and Formats

File with that name is already open (FAQ)
File size, large (FAQ)
Fill colour doesn't work (FAQ)
Fill pattern doesn't print (FAQ)

Filter, Advanced
Filter, AutoFilter

Sample Data
Sample Workbooks updated 10-Mar-08
Sample Workbooks (Ron Coderre) updated 20-Sep-07
Sample Workbooks (Roger Govier) updated 18-May-09

Scenarios -- Automatically Show 10-Apr-05
Scenarios -- Create and Show 03-Apr-05
Scenarios -- Programming 12-Apr-05
Scenarios -- Scenario Summaries 03-Apr-05

Shortcuts, Keyboard

Sorting a List
Sorting Data -- Programming 06-Aug-06
Store, The Excel

Subtotal
Sum cells
Sum a Filtered List

Survey Form 29-Oct-05 updated 11-Oct-06

T-Z

Table, Excel 21-Aug-08

http://www.contextures.com/xlToolbar02.html
Toolbar -- Macros 24-Dec-05
Toolbar -- Navigate Workbook Sheets updated 30-Oct-07
http://www.contextures.com/xlToolbar01.html

Topics Index

Trailing Minus Signs

Used Range, Reset (FAQ)

UserForm, Create a Video
UserForm with ComboBoxes 23-Jan-06

VBA Code, Copy to a workbook
Video Index
Video Instruction Clips 01 10-May-08
Video Instruction Clips 02 10-May-08
Video Instruction Clips 03 13-May-08
Video Instruction Clips 04 23-May-08
Video Instruction Clips 05 01-Jun-08
Video Instruction Clips 06 updated 25-Jul-08
Video Instruction Clips 07 updated 31-Aug-08
Video Instruction Clips 08 updated 30-May-09 UPDATED
Video Instruction Clips 09 05-Feb-09
Video Instruction Clips 10 19-Feb-09

	http://www.contextures.com/xlFunctions02.html VLOOKUP function Video

CHIP PEARSON'S EXCEL TIPS:

http://www.cpearson.com/excel/MainPage.aspx

http://www.cpearson.com/excel/topic.aspx This Tips Topics Index

A

Absolute And Relative Cell References
Activating Excel From Other Applications
ActiveCell, Highlighting
Add-Ins, Automation, Creating
Add-Ins, COM, Creating With VB6
Add-Ins, Creating
Add-Ins, Installing And Loading
Add-Ins And Utilities, Third Party
Age, Calculating
ALT, SHIFT, and CTRL Testing State Of Key
Analysis Tool Pack (ATP), Installing
Analysis Tool Pack, Calling Function From VBA
AnyXML, Allowing optional and arbitrary XML content with an XSD Schema
API Functions, Getting Error Information
Application Events
Application Shutdown, Detecting And Taking Action
Application-Level Names
Arguments, Passing ByVal And ByRef
Array Formulas, Described
Array, Converting To Columns
Array, Testing If Allocated
Array, Testing If Sorted
Arrays, Determining Data Type Of
Arrays, Number Of Dimensions

M

Macro-Sheet Function, Calling From Worksheet Cell
Macros, Adding or Deleting With VBA Code
Macros, As Opposed To Functions
Macros, Ensuring Macros Are Enabled, Technique 1 (Sheet Visibility)
Macros, Ensuring Macros Are Enabled, Technique 2 (Calculations With Errors)
Macros, Running From Worksheet Cell
Matrix To Vector Formulas
Maximum Values, Persistent
Me Reference, Self-referencing an instance of a class
Menu Items, Creating Manually
Menu Items, Creating With VBA Code
Menu Items, Creating For The VBA Editor
Merging Lists Without Duplicates
Minimum And Maximum Values
Minimum And Maximum Values, Persistent
Missing References In A VBA Project
Modified File, Returning The Most Or Least Recently Modified File In A Folder
Modules, Adding And Deleting With Code
Modules, Adding descriptions for the Object Browser
Months, Calculating Fractional Months
Most Or Least Common Entry In A List
Moving A Form With The Window
Multiple Monitors

B

C

N

O

P

Q

R

D

S

E

T

G

V

W

L

OZGRIDS FORMULAS W/DOWNLOADS:

http://www.ozgrid.com/forum/index.phps=26c4d4689355798111b17e605a0d4eb6&

Look for the two links to downloadable Formulas and the one for downloadable Code.

JON PELTIER'S CHART TUTORIALS

Jon Peltier has premium quality knowledge of Excel Charting and Graphing and you would be hard pressed to get better starting help on Charting with Excel from someplace other than

http://peltiertech.com/Excel/Charts/index.html

CHARLES WILLIAMS DECISIONMODELS.COM SITE

Long overlooked and underappreciated, Charles Williams was finally awarded Microsoft Excel MVP status. Long overdue. His site has some really informative information, good help and great 'inside' information about the way that Excel works. For the really serious, his Fast Excel analysis tool is definitely one to have around.

This page has many links at the top regarding how Excel's (re)Calculation engine works, and how to make it work for you:

http://www.decisionmodels.com/calcsecrets.htm

This page has lots of tips and information on how to speed up the performance of your workbooks:

http://www.decisionmodels.com/optspeedh.htm

Need to find out more detail about the memory requirements or usage in your version of Excel? Then check out this page:

http://www.decisionmodels.com/memlimits.htm

TOOLS AND DOWNLOADS BY JAN KAREL PIETERSE

Here you will find some really useful, and FREE tools for working with Excel.

http://jkp-ads.com/Download.asp

JOHN WALKENBACH FREE EXCEL TIPS

You may have seen books in the Computers section at your local Barnes & Noble, Borders, Waldenbooks or your very local Ma & Pa Smith's Books and Antiques shop. I own several of his books myself, and his Excel 2007 Bible (ISBN: 0470044039) is highly respected. But you can get excellent information from his site without leaving your keyboard or spending any extra $.

http://spreadsheetpage.com/ John Walkenbach site's main page.

Here is the Table of Contents for his Excel Tips at the site, with some having companion files that can be downloaded.

General

- Getting A List Of Files Names - Another Method
- Clearing The Text To Columns Parameters
- Making An Exact Copy Of A Range Of Formulas, Take 2
- Create A Drop-Down List Of Possible Input Values
- Excel 2007 Upgrade FAQ: Charts And Graphics
- Excel 2007 Upgrade FAQ: Formatting And Printing
- Excel 2007 Upgrade FAQ: General
- Excel 2007 Upgrade FAQ: User Interface
- Using Custom Number Formats
- Navigating Excel's Sheets
- Override Excel's Text Import Wizard
- Sharing Autocorrect Shortcuts
- Making A Worksheet Very Hidden
- Importing A Text File Into A Worksheet
- Using A Workspace File
- Protecting Cells, Sheets, Workbooks, And Files
- Resize Excel's Sheet Tabs
- Changing The Number Of Sheets In A New Workbook
- Close All Workbooks Quickly
- Restrict Cursor Movement To Unprotected Cells
- Change The Color Of Worksheet Tabs
- Making An Exact Copy Of A Range Of Formulas
- Creating A Database Table From A Summary Table
- Solving Common Setup Problems
- Getting A List Of File Names
- CommandBar Calculator
- Spreadsheet Protection FAQ
- Extended Date Functions

Formatting

- Quantifying Color Choices
- Excel 2007 Upgrade FAQ: Formatting And Printing
- Comparing Two Lists With Conditional Formatting
- Alternate Row Shading Using Conditional Formatting
- Duplicate Repeated Entries In A List
- Removing Or Avoiding Automatic Hyperlinks
- Working With Fractions
- Using Conditional Formatting
- Fix Incorrect Decimal Places During Data Entry
- Display Text In Multiple Lines
- Changing The Default Cell Comment Formatting
- Change The Formatting Of Your Subtotal Rows

Formulas

- Is A Particular Word Contained In A Text String?
- Formulas To Perform Day Of Month Calculations
- Making An Exact Copy Of A Range Of Formulas, Take 2
- Calculating Easter
- Converting Unix Timestamps
- Naming Techniques
- Creating A List Of Formulas
- Cell Counting Techniques
- Summing And Counting Using Multiple Criteria
- Chart Trendline Formulas
- Making An Exact Copy Of A Range Of Formulas
- Comparing Two Lists With Conditional Formatting
- Locate Phantom Links In A Workbook
- Dealing With Negative Time Values
- Converting Non-numbers To Actual Values
- Compare Ranges By Using An Array Formula
- Calculate The Number Of Days In A Month
- Identify Formulas By Using Conditional Formatting
- Displaying Autofilter Criteria
- Calculating A Conditional Average
- Display Text And A Value In One Cell
- Automatic List Numbering
- Calculate The Day Of The Year And Days Remaining
- Rounding To “n” Significant Digits
- Working With Pre-1900 Dates
- Using Data Validation To Check For Repeated Values
- Sum The Largest Values In A Range

- Count Autofiltered Rows
- Perform Two-Way Table Lookups
- Referencing A Sheet Indirectly
- Delete All Input Cells, But Keep The Formulas
- Round Values To The Nearest Fraction
- Avoid Error Displays In Formulas
- Change Cell Values Using Paste Special
- Hiding Your Formulas
- Counting Distinct Entries In A Range
- Force A Global Recalculation
- Summing Times That Exceed 24 Hours
- Transforming Data With Formulas
- Creating A "Megaformula"
- Alternatives To Nested IF Functions
- A Formula To Calculate A Ratio

Charts & Graphics

- Saving A Range As A Graphic File
- A Quick And Dirty Slideshow Macro
- Excel 2007 Upgrade FAQ: Charts And Graphics
- Pasting An Image To A UserForm Control
- Interactive Chart With No Macros
- Creating A Splash Screen For An Excel Workbook
- Creating A Clickable Image Map
- A Class Module To Manipulate A Chart Series
- Chart Trendline Formulas
- Removing Lines From A Surface Chart
- Update Charts Automatically When You Enter New Data
- Creating A Non-Graphic Chart Directly In A Range
- Creating A Linked Picture Of A Range
- Creating A Thermometer Style Chart
- Displaying A value in an AutoShape
- Handle Missing Data In A Line Chart
- Format Cells To Display In Thousands
- Unlink A Chart Series From Its Data Range
- Display Multiple Charts On A Single Chart Sheet
- Layouts For Column Charts
- Saving A Chart As A GIF FIle
- Rotating Text With An AutoShape
- Creating A Transparent Chart Series
- Creating Combination Charts

- Animated Hypocycloid Charts

Printing

- Excel 2007 Upgrade FAQ: Formatting And Printing
- Determining The Number Of Printed Pages
- Mail Merge - Without Word
- Displaying A Menu Of Worksheets To Print
- Copy Page Setup Settings To Other Sheets
- Printing Just A Portion Of Your Worksheet
- Avoid Printing Specific Rows

Developer Tips by Category

General VBA

- Is A Particular Word Contained In A Text String?
- The Value, Formula, and Text Properties
- Clearing The Text To Columns Parameters
- A Macro To Count Word Frequencies
- Saving A Range As A Graphic File
- A Quick And Dirty Slideshow Macro
- Maximize Excel Across All Monitors
- Understanding The IsDate Function
- Excel 2007 Upgrade FAQ: Macros
- Controlling User Scrolling

CommandBars & Menus

- Add The Speech Controls To The Ribbon
- Identifying CommandBar Images
- Creating Custom Menus
- Developer FAQ - CommandBars
- CommandBar Calculator

UserForms

- Pasting An Image To A UserForm Control
- Displaying Help
- General Userform Tips
- Selecting A Directory
- Displaying A Progress Indicator
- Importing And Exporting Userforms

- Handle Multiple Userform Buttons With One Subroutine
- Filling A Listbox With Unique Items
- Displaying A Menu Of Worksheets To Print
- Creating A Color Picker Dialog Box

VBA Functions

- Extracting An Email Address From Text
- Quantifying Color Choices
- Determining The User's Video Resolution
- Identifying Unique Values In An Array Or Range
- Getting A List Of File Names Using VBA
- Looping Through Ranges Efficiently In Custom Worksheet Functions
- Undoing A VBA Subroutine
- Determining The Last Non-empty Cell In A Column Or Row
- Multifunctional Functions
- Some Useful VBA Functions
    - File Exists
    - FileNameOnly
    - RangeNameExists
    - SheetExists
    - WorkbookIsOpen