Excel Training - VBA

http://www.spreadsheetml.com

Table of Contents

1. INTRODUCTION

In this first lesson we will discuss the basics of VBA for Excel. I purposely do not include screen shots or pictures, as I believe by omitting them it forces the student to explore and become more familiar with the environment in which they will be working. This may seem a bit inconvenient at first, but the long-term gain will far outweigh the short-term pain. The basics are probably not very exciting, but definitely necessary so you have a good understanding of the Application (Excel).

VBA is short for **Visual Basics for Applications**. This is the standard Macro language used in most Microsoft Office products. The word "Applications" can represent any one of the Office products it is used within eg; Excel, Access etc. The main ones being Excel, Word, Access, Powerpoint and is now, in Office 2000, moving into Outlook. The VBA language is a derivative of **Visual Basic** (VB), which in turn is a derivative of the language **Basic**. The fundamental difference with VBA from VB is that VBA is (as the name implies) used within an Application. By far the most mature of these Applications when it comes to VBA is **Excel**. You will find as we delve deeper into VBA for Excel that we can modify the Application so it will behave in almost any way possible.

The purpose of VBA is to enable programmers to customise and extend the functionality of the Application in which it is used. The VBA we will be talking about in all lessons will be VBA for Excel. While the VBA language is generally the same throughout other Applications, their **Object model** can differ significantly. Unless you are already familiar with a programming language I would recommend not trying to learn too much too soon. I believe it is far better to gain a good understanding of one small aspect of VBA for Excel than to gain a superficial knowledge of a broad aspect. I can honestly say (without hesitation) that I may not know every single aspect of VBA for Excel, but what I do know, I know well. Please at anytime throughout the course never feel that any question is a silly question and if you do not understand an answer I supply for a particular question then it is important you say so. Trust me I have immense patience! One of the most important things about the Basic VBA for Excel course is that you have an understanding of each lesson before moving on to the next lesson.

1.1 Getting Started - The Look of the Visual Basic Editor

To **write** any VBA code (not record), we need to go into the VBE (b>Visual Basic Editor). There are many ways to do this, but by far the easiest is by pushing the shortcut key **Alt + F11** (hold down the Alt key and push F11). You can also access the Visual Basic Editor by going to **Tools > Macro > Visual Basic Editor** from within Excel.

Below is a summary of arguably the most important components of the Visual Basic Editor. I would like you to work through this, so open a **new** workbook within Excel. Open the Visual Basic Editor using one of the ways described in the previous paragraph (**Alt + F11** is the fastest). *It is important that you use a new Workbook so that all names and terms we refer to are the same.* The **positioning** of the windows I refer to may be different in your view, but I will use the headings also so you shouldn't get lost.

At the very top of the VBE you have what is known as the **Menu Bar**. From this one menu bar it is possible to access **most functions** of the VBE. If you right click on the grey area just to the right of **Help** on the Menu Bar a shortcut menu should appear. Select **Customize...**. Here you will see the names of all the Toolbars that can be available to you. As a general default you will have the

Menu Bar and the **Standard** Toolbar showing. To make sure we are looking at the same view, go to **View > Project Explorer**, then back to **View > Properties Window**. Or to use the shortcut keys, **Ctrl + R** for **Project Explorer**, and **F4** for the Properties Window. You should now have visible the **Project Explorer** and the **Properties Window**. If you go up to the top Menu bar and click on any menu item you will notice that many of the functions available have their associated short cut keys written next to them. Get to know these well and working from within the VBE will become mush easier.

1.2 Project Explorer and Properties Window

Within the **Project Explorer** (the small window with the heading **Project - VBAProject**) you will see at the top **VBAProject (Book1).** This window and it's folders are very similar to the folders you would see in **Windows Explorer** in that you will be able to expand and collapse folders by clicking on the + or - signs to expand and collapse them. Expand VBAProject (Book1)! Once the VBAProject (Book1) is expanded, you should see a folder called **Microsoft Excel Objects**. Expand this folder and you will see the **Objects** that the Workbook contains. In most cases this will be **Sheet1** (Sheet1), **Sheet2** (Sheet2), **Sheet3** (Sheet3) and **ThisWorkbook**. It is not necessary to know anything more about these at this stage, but I will be showing you how they can be used in a later lesson. But as you have no doubt guessed, the **Sheet1**, **Sheet2** etc refer to the Worksheets in the Workbook, while **ThisWorkbook** refers to the Workbook itself. If you now click on **Sheet1** in the **Project Explorer** you will see in the **Properties Window** (the window probably directly beneath) a list of all the Properties for a standard Worksheet. Don't be too concerned with what **Properties** are at this stage, we will go into the detail later.

1.3 Common Terms used in Excel VBA:

1.3.1 Modules

The grey area of your screen is where your **modules** (where code is written) are located. As this is a new Workbook we are looking at, there will not be any Modules visible. To insert a Module we go to **Insert > Module**. You should then see a white background where it used to be grey, this is what's know as a Module, sometimes referred to as a **standard Module** or **code Module** in earlier versions. It is within these Modules that we can write VBA code. The code we write is written within a **Procedure**, a Procedure is a series of statements giving Excel instructions on what to do (a macro). We can have as many Procedures within a Module as we like and we can also have as many Modules within a Workbook as we like, the only restriction is the PC's available memory.

There are 2 types of Modules in the Excel VBE, these are **Module** and **Class Module**. The Module (standard Module) is the one we will be using throughout this course. The standard Module can also have what is known as **Private** or **Public** modules. By default all standard modules are Public. The Private Module is used in some of Excels other Objects, ie UserForms, Sheets and ThisWorkbook. Again we wont go into any more detail on this yet as we will be covering these later.

One thing you will find yourself doing many times over when writing VBA code is toggling between the VBE and Excels user interface. The simplest way to do this is to either again push Alt + F11 or click on your Microsoft Excel Workbook located on the **Task Bar**. Let's now look at the Fundamentals of VBA for Excel.

1.3.2 Procedures

Procedures are a named set of statements that are executed as a whole. They tell Excel how to perform a specific task. The task performed can be very simple or very complicated. It is good practice to break long or complicated procedures into smaller sized logically ordered procedures. The two main types of Procedures are **Sub** and **Function**. A Sub procedure is what results from recording a macro, while Function procedure must be written manually. *We will be looking at Function procedures in a later lesson.* All Sub procedures must begin with the word **Sub** followed by a chosen name and then a set of parenthesis. All Subs must end with **End Sub**. See example below:

```
Sub MyMacro()

'<Procedure Code>

End Sub
```

1.3.3 Keywords

Keywords in Excel VBA are words that Excel has set aside to use in the execution of code. This means we cannot use them for any other purpose. For example, Select, Active, Sub, End, Function etc are all Keywords that we can only use for their indented purpose.

While working in the Visual Basic Environment we can get help on any keyword by simply placing our mouse insertion point anywhere within the keyword and pressing F1. This will force the Excel help to default to the specific topic and saves a lot of hunting around. **I urge all users to use this feature to its fullest.**

1.3.4 Comment Text

Comment text is what we can use in a Procedure to help explain what our code is doing, or rather why it is doing it. To identify comment text to Excel we must precede it with a single ' (apostrophe). Excel will ignore any text preceded with a '. I strongly suggest using lots of comment text in procedures as it helps greatly when you come back to it later to edit the code. See example below:

```
Sub MyMacro()

' Place the value 150 in cell A1 of the active sheet
Range.Value=150

End Sub
```

You will notice that comment text is a different color to procedure code. This is so it is easily identified as such.

The default color coding that Excel uses for Comment text, Keyword text, Procedure text etc can be changed by going to Tools>Options and clicking the Editor Format tab. don't get too carried away though :o)

1.3.5 Line Continuation Character

The Line Continuation Character is used to tell Excel that more than one line of code is really a single line. It is represented by _ (space then an Underscore). So to continue a line of code onto the next line in a module you type a space followed by an Underscore. The reason you would use this is so that you would not need to scroll sideways to read a long line of code. See example below:

```
Sub Find100()
'
' Find100 Macro
' Finds the value of 100 on the active sheet.

Cells.Find(What:="100", After:=ActiveCell, LookIn:=xlValues, _
LookAt:=xlPart, SearchOrder:=xlByRows, SearchDirection:=xlNext, _
MatchCase:=False).Activate

End Sub
```

The above example is really one continuos line of code. As you can imagine this would not fit on most PC monitors if we did not use a Line Continuation. Well, not unless you are lucky enough to have an 80 inch screen!

1.4 Important Definitions:

1.4.1 Objects

The word **Object** in the context that we use it is used to describe just about everything in Excel. You will find as you get deeper and deeper into VBA that there are many **Objects**. Don't confuse the word with its usual use of referring to something tangible. In Excel, an **Object** can be a range, a worksheet, a chart or even the Visual Basic Editor (VBE) itself. You will find as you learn more about Excel VBA that there are a **lot** of Objects!

The very least that you need to know about **Objects** is that there is what is known as the **Object Hierarchy**. At the top of this order we have the **Application Object**, Excel itself in this case. Directly underneath the Application we have the **Workbook Object** eg; Book1.xls. Directly underneath the Workbook Object comes the **Worksheet Object**. At the **Worksheet Object** the Object Hierarchy branches off to incorporate all Objects of the Worksheet. The first one you will most likely encounter will be the **Range** Object. Branching from the **Range** Object there are again many other objects such as the **Areas** Object, the **Borders** Object, **Font** Object, **Characters**, etc. To see a complete description, of the hierarchy of Objects go to the VBE for Excel **Help** and type **Microsoft Excel Objects**. Don't let this confuse you, as all you need to know at this stage is the following order:

- Application
- Workbook
- Worksheet
- Range

When you have a group of Objects that are related, this is then known as a "**Collection**". So when we use the term **Workbooks**, we are referring to **all** open Workbooks, and when we use the term Workbook, we are only referring to an individual Workbook (the active Workbook).

You will notice when reading the above paragraph and in the VBE Help, that the **Workbook** Object is part of the **Workbooks** Object. In layman's terms, the Workbook is a member of the Workbooks collection. It is also the same for the **Worksheet** Object, this is a member of the **Worksheet**s collection. Don't worry too much if you don't grasp this concept immediately, it will become apparent as we progress.

1.4.2 Properties

Properties are an attribute of an **Object**. They are used to define an Objects characteristics, so to use an example; the **Worksheet** Object has many Properties, one of which would be its **name**. So by changing it's name, you are directly changing one of its Properties. Also by making the Worksheet hidden, you are again changing one of its Properties, in this case, the **Visible** Property. To be able to change the **Property** of any **Object** we must first identify the **Object** of whose **Property** we wish to change. In the example above we would do this by telling Excel which **Worksheet** we wish to rename. You will see how this can be done later.

1.4.3 Methods

Like Properties, **Objects** also have **Methods**. A Method is simply a procedure that acts on a Object. By this I mean it causes the Object do do something, this might be opening a Workbook (which is an Object) or deleting a Worksheet (another Object) or any one of thousands of other things.

1.4.4 Events

An Event in Excel VBA is as the name implies. For example a mouse click is an event as is the closing or opening of a Workbook, in fact there are hundreds of them. The best part about Events is that we can we can programme Objects to respond to Events. This is covered in detail later in lesson 6.

1.4.5 The Macro Recorder

The Macro recorder in Excel is a great tool to use as a VBA learning aid. It will record any steps taken while in Excel and write the code so as they can be performed again. By studying the code written by the Macro recorder we can better understand the VBA language as applied to Excel. Having said this though it is **important to realize** that it does **not** write very **efficient code**. This is because it is only recording the steps you have taken to perform a particular task. Most of these steps are not necessary when writing efficient VBA code. For example you might record a macro going to all Worksheets within your Workbook and changing the **Format Properties** of the cells. To do this you would need to activate each Worksheet and then scroll to the required cell(s), select them and then format them as required. You would find that the code generated by the Macro recorder would be quite lengthy and most of it would be superfluous. This is because the Macro recorder cannot really write VBA code as it should be written, it can only write the code for **ALL** the steps you have taken. As a general rule with recorded code you can remove all the words like: **Select**, **Activate**, **Scroll** etc. This is because **we very rarely** need to actually **Select**, **Activate** or **Scroll** to a Worksheet or Range to perform any action on it. You will see what I mean by this soon!

While it is true that the Macro recorder is a good method for learning VBA there is also another reason why we should use it. This is because it can eliminate typos and save time. Although I can usually write the code for a particular task in Excel, I often record a macro doing it and then go into the VBE and edit it so that it becomes much more efficient.

To activate the Macro recorder go back to Excel (Alt+F11), go to **Tools>Macro>Record New Macro**. This would display the **Record macro dialog box**. It is here we can give our macro a one word meaningful name. There is also a drop down arrow asking where we want to store the macro, for all purposes unless stated otherwise, we will use "**ThisWorkbook**."

1.5 Exercise

1. Start recording a macro (call it **ChangeRangeProperty**)
2. Select Cell **A1**
3. Go to **Format > Cells > Number > Currency**
4. Select any one of the Currency Formats
5. Click **OK**
6. Stop recording the macro

Open the VBE (**Alt + F11**). In the **Project Explorer**, you should now see under **Microsoft Excel Objects** another folder (Object). In this case the **Modules Object**, and within that you will see **Module1**, which again is another Object. Modules is a collection of Objects which in this case are the **ModuleObject**. Double click on Module1 to open the Module. You should see some code as shown below.

```
Sub ChangeRangeProperty()

Selection.NumberFormat = <your format chosen>
Range("A1").Select

End Sub
```

Let's go through each bit of this and define what each word means.

Sub

The word **Sub** is simply letting Excel know that there is a Procedure contained within it. A Procedure is a series of statements giving Excel instructions on what you wish to do. Basically a Procedure is Marco.

()

After each Procedure name is a pair of empty parenthesis. This is reserved in case wish to add arguments to the Procedure. These can be similar to the arguments that are often used in

Worksheet formulas. Do not worry about the arguments at this stage as it will only add confusion and we can easily get by without them.

Range

A range as we have discussed above is member of the WorksheetObject. You will notice that the recorded macro **does not** include the **WorksheetObject** (the sheet we changed number format of cell A1). This is because the default for the **RangeObject** on it's own is **always** the active sheet. So unless the range we are referring to is on a different sheet, there is no need to use the **WorksheetObject**.

"A1"

This is simply the cell reference address that we selected. It must be enclosed in quotations as Excel sees cell addresses as text or strings (chain of characters that represent the characters themselves rather than their numeric values)

.Select

Select (in this case) becomes a method of the **RangeObject**. *A method is a Procedure that acts on an Object*. If we had selected a Worksheet it would be a **WorksheetObject**, a Chart a ChartObject etc.

Selection

Selection is as it states, simply returning to Excel the Object that has been selected. In this case, the RangeObject. The key word Selection is very generic as it can refer to nearly all Objects within Excel, but as the line Range ("A1").Select is the line of code immediately before it, Excel knows that in this case Selection is referring to a RangeObject only.

.NumberFormat

NumberFormat is a Property of the RangeObject.

= <your format chosen>

Is simply the format type that you have chosen.

End Sub

End Sub tells Excel that the Procedure has finished.

While we can certainly find out a lot from recording a macro and studying it's code, you will find as time goes on that the recorded code will include many lines of code, Properties, Methods etc., that are not needed.

As an example we could easily shorten the above-recorded macro to read:

```
Sub

Range ("A1").NumberFormat = <your format chosen>

End Sub
```

The reason we would do this is

1. It means less typing (good!!)
2. It means faster execution of code (although barely noticeable in this small example)
3. It means there is no need for the active cell to change. So this means that if you ran the Macro immediately above, while any cell except A1 is your active cell, the code will run without the user knowing.

What this means in a nutshell is that in MOST instances there is no need to select or activate an Object to change any one of it's Properties.

1.6 Exercise

Record three macro's as follows:

1. Selecting any cell, typing a number in and pushing **ENTER**.
2. Selecting a different Worksheet and highlighting the range **A1:D10** change the background colour to **yellow**.
3. Selecting any cell, typing a number in and pushing **ENTER**. Copy the number you just typed and paste it to another cell

   When you have recorded these three macros, I would like you to try and modify the code on all of them so that the words "Select" or "Selection" does not appear within the Sub. Make use of the VBE Help. When you have modified as much as you can so that they still work the same, e:mail them to me. If you get stuck, please do not hesitate to contact me. I fully realise that a lot of the Object, Property, terms etc. used in VBA can seem daunting, but please don't be deterred at all if you do not understand or grasp the concept. Take my word for it, there are many programmers out there who quite successfully use VBA without fully understanding the concept of what we have described here.
   Any questions at all relating to this lesson, let me know. Please don't feel that any question is a stupid question. In fact, stupid is to not ask the question if you are unsure.
   I look forward to hearing from you soon. When the next lesson commences is entirely up to you. Just keep in mind that it is important to understand what we have discussed here before moving on.

2. COMMON OBJECTS

Please cross reference the ExcelObjectsExamples.xls Workbook for this lesson.

In this lesson we will look at the 4 most common (and arguably useful) Objects, these are:

- Application
- Workbook
- Worksheet
- Range

We will look at each of these in turn and I will show you what I believe to be their most useful Properties and Methods. One of the best features of VBA for Excel is that by typing an Object name and then typing a period (full stop) Excel will list all of the Properties and Methods associated with it. Use this feature to it's fullest and not only will you save typing but you should eliminate all typos.

2.1 Application

You will recall the Application is at the top of the Object hierarchy and contains many Properties and Methods as well as Collections of other Objects. You would use the Application Object to gain access to Object Collections such as:

2.2 CommandBars

This will return ALL built in CommandBars and ALL custom CommandBars that are within the open Workbook. There are 85 built in CommandBars in Excel. and 845 associated level 1 Controls. How do I know this, I counted them! Not really :o) Included with this lesson is some code I have written that will list all CommandBars and their associated **level 1 Controls**. As you learn more about VBA for Excel and you become comfortable with it, you will no doubt find yourself wanting to do more than just automate common tasks within Excel. You will possibly end up wanting to create your own custom CommandBar and/or modifying Excels built in CommandBars. To do this you will need to gain access to them and their Controls and to do this you will need to know their names or captions. There are two ways to access CommandBars and their Controls, one is to use the Index number and the other is to use the name or caption. I strongly recommend using the latter (name or caption) method as you will be able to tell at a glance which CommandBar or Control you are dealing with.

As CommandBars represent a Collection of the CommandBar Object and CommandBars are a member of the Application we must go through the correct order to gain access. So if we want to hide a particular CommandBar we would use the code:

```
Sub HideACommandBar()
    Application.CommandBars("Formatting").Visible = True
End Sub
```

As you can see this will simply show the Formatting CommandBar. We have told Excel we want a Object member of the CommandBars Collection and that member is called "Formatting". We have then set it's Visible Property to True. the Visible property returns a Boolean ie, either True or False. So having gained access to the "Formatting" CommandBar we can now also gain access to any one of it's Controls. To do this we would use:

```
Sub DisableACommandBarsControl()
   Application.CommandBars("Formatting").Controls("&Fill Color").Enabled =False
End Sub
```

This would grey out (disable) the "Fill Color" icon on the "Formatting" Commandbar. *The use of the "&" before the Control names simply represents the underline that is used on Excel Toolbars, eg **File** would be expressed as ("&File").*

Now if we are dealing with the "Worksheet Menu Bar" Excels main CommandBar we not only need to gain access to the first level Control, but also it's second level and third level. To do this we would use:

```
Sub AccessMenuBarLevel2()
   Application.CommandBars("Worksheet Menu Bar").Controls _
                ("&Insert").Controls("&Name").Enabled = False
End Sub
```

From here we can also go to the last level, to do this we would use:

```
Sub AccessMenuBarLevel3()
   Application.CommandBars("Worksheet Menu Bar").Controls _
                ("&Insert").Controls("&Name").  _
                Controls("&Define...").Enabled = False
End Sub
```

So as you can see, once we know the names of the CommandBar and It's Controls caption we are able to manipulate it in many ways.

2.3 WorkBook

As with the CommandBar Object, to gain access to a Workbook Object we go through the Workbooks Collection. some of the most common uses of the Workbook Object are *Opening*, *Saving*, *Activating* and *Closing***.** Lets look at each of these.

2.3.1 Opening

To open a Workbook we must tell Excel it's name and if we are dealing with more than one directory, it's file path. So to open a Workbook that is on the **same** drive in the **same** folder we would use:

```
Sub OpenAWorkbook()
   Workbooks.Open ("Book1.xls")
End Sub
```

The "**Open**" Method also takes some arguments all of which are optional. (*An argument is a constant, variable, or expression passed to a procedure*). The most common one is the "UpdateLinks"

UpdateLinks tells Excel how to deal with a file we are opening that contains links. If the argument is omitted, the user is asked (via a message box) how links should be updated.

The table below is from the VBE help in Excel

Value	Meaning
0	*Doesn't update any references*
1	*Updates external references but not remote references*
2	*Updates remote references but not external references*
3	*Updates both remote and external references*

So to open a file and NOT update links we would use:

```
Sub OpenAWorkbookWithLinksWithoutUpdating()
   Workbooks.Open "Book1.xls", UpDatelinks:=0
End Sub
```

2.3.2 Saving

With this Method we can Save a Workbook as it's existing name or as another name and path. We can trick Excel into thinking a Workbook is already had it's changes Saved. This may also be a good time to introduce the two methods available to refer to the active Workbook. The first one is:

```
Sub SaveActiveWorkbook()
    ActiveWorkbook.Save
End Sub
```

The second is:

```
Sub SaveThisWorkbook()
    ThisWorkbook.Save
End Sub
```

Both of these Methods will Save the a Workbook as it's existing name. There is one small (which could be huge) difference with these two methods and that is:

1. "**ActiveWorkbook**" **always** refers to the Workbook that happens to be active at the time of running the code.
2. While "**ThisWorkbook**" *always* refers to the Workbook that **houses** the code, regardless of which workbook happens to be the Active Workbook.

Each Method is good in that they are generic (we don't need to know the name of the Workbook). But use them in their wrong context and you could end up accessing the wrong Workbook. I generally use "**ThisWorkbook**" with the exception of the Workbook being an **Add-in**. *An Add-in is a Workbook that has been saved as such (*.xla). Once it has been saved it will always open as a Hidden Workbook.*

As I motioned above we can trick Excel into thinking any changes to a Workbook have been saved, to do this we would use:

```
Sub TrickExcelToThinkWorkbookIsSaved()
    ThisWorkbook.Saved = True
End Sub
```

Run this code immediately prior to closing a Workbook that has had changes and Excel will **think** the Workbook has already been saved and close without saving.

2.3.3 Activating

While we can (in theory) have an infinite number of Workbooks **open** at any one time we can only ever have one of them **Active** at any one time. At times we may need to Activate another Workbook so we can do something with it via VBA. If we try to access another Workbooks Objects etc while it is NOT Open we will encounter a Run time error! So it is important that the Workbook is Open. It is rare that we would need to Activate it, but if we do We can do this easily if we know it's name like this:

```
Sub ActivateAnotherWorkbookViaName()
    Workbooks("Book2").Activate
End Sub
```

Problem is we may not always know the name of any other open Workbook so we need to either find out it's name or use it's Index number, like this:

```
Sub ActivateAnotherWorkbookViaIndex()
    Workbooks(3).Activate
End Sub
```

An important note here is, the index number is the same as the order in which the workbooks were opened. Workbooks(1) is the first workbook opened, Workbooks(2) is the second Workbook opened and so on... Activating a Workbook won't change its index number. All open workbooks are included in the index count, even if they are hidden.

2.3.4 Closing

As with Open Method of a Workbook the Close Method also takes arguments, which again are all optional. To close the open Workbook you would use:

```
Sub CloseThisWorkbook()
    ThisWorkbook.Close
End Sub
```

This would Close the Workbook and Prompt the user to save any changes.

```
Sub CloseThisWorkbookAndSave()
    ThisWorkbook.Close SaveChanges:=True
End Sub
```

This would Save the Workbook without prompting and then Close. Of course using "**False**" in place of "**True**" would Close the Workbook and not save any changes.

2.4 Worksheet

The Worksheet Object will most likely be an Object that you will encounter often. You will need to refer to it to gain access to **it's Objects**, with the **exception of the ActiveSheet**. *If the Worksheet is not identified in your code Excel will by default assume the Active sheet*. As with the Workbooks collection we can refer to a Worksheet via its **Name** (tab name) or its **Index number**. One thing you should be aware of is **we cannot access a Chart Sheet** via the Worksheets Collection Object as a Chart sheet is **not a Worksheet** and as such not a member of the Worksheets Collection. To access a Chart sheet we would use the **Sheets** Collection Object. The Sheets Collection Object represents **ALL sheets** in the ActiveWorkbook or the specified Workbook (including Chart Sheets).

To use its Name you would use:

```
Sub UsingTabName()
    Worksheets("Sheet2").Activate
End Sub
```

Or

```
Sub UsingTabName2()
    Sheets("Sheet2").Activate
End Sub
```

To use its Index number you would use:

```
Sub UsingIndexNumber()
    Worksheets(2).Activate
End Sub
```

Or

```
Sub UsingIndexNumber2()
    Sheets(2).Activate
End Sub
```

This would activate the **second** Worksheet in a Workbook. The Index numbers run from **left to right**.

Personally I use the Sheets Object most of the time not only is it easier to type but it also allows us to use **much better** method of accessing a Worksheet and that is via its **CodeName**. I wont go into too much detail on the Sheets Codename at this point as we will cover it in more detail in your Debugging code lesson later. What I will say is it can be found in the Properties Window of the Sheet object and also in the Project Explorer. It is the

name **NOT** in the parenthesis. So for the two examples above you could use:

```
Sub UsingCodeName()
    Sheet2.Activate
End Sub
```

I will stress now that using the CodeName Method is **one of those habits you should form early**. When we do your Debugging lesson this will become apparent.

2.5 Range

Without doubt the Range Objects is by far the most used and important aspects of not only VBA for Excel but for Excel itself. As Excel is primarily a spreadsheet application its whole concept depends on the Range. To put it in layman's terms, Excel is a load of little boxes (116777216 **per Worksheet**). We can access any one of these Cells (little boxes) by nominating its unique address. We do this by using a grid pattern and this is very similar to finding a Street on a road map it is that simple! When referring to a Cell or a Range of Cells within Excel we use either the **R1C1 method**, where R=Row and Column=Column or the **A1 method**. The A1 is the **preferred Method** so that is what we will use. Now as I have said, Excel has 116777216 Cells on each Worksheet these are represented by 256 Columns and 65536 Rows remember these 2 numbers as they come in very handy **especially** the Row number. The amount of Columns and Rows on a worksheet are fixed (and as a consequence so are the cells), we **cannot** add more and we **cannot** remove any. As we have far more Rows than columns we should always attempt to work with this in mind. Excel is geared up for tables to be set up with Columns as headings and Rows for data storage.

When we are working with the Range Object we should try to avoid ever having to Select it you will find we can achieve this aim 99% of the time. The two most common ways to refer to a Cell are:

Cells(1, 1).Select

Or

Range("A1").Select

Yes , I know I used the word Select :o)

I much prefer to use the latter as I find it much simpler to use a letter for a Column and number for a row. I guess most people do, which is why Excel introduced the A1 style reference. There also another way to refer to a cell and is the way we should use whenever possible, this way is to use its name (if it has one). If we had a Cell or Range of Cells named "**MyRange**" we would use:

Range("MyRange").Select

I will stress here again that form the **habit early of naming Ranges in Excel**. This is very important in VBA as users are inserting Rows, Cells and Columns or cutting and pasting may have you referring to the wrong cell. Once a Cell or Range of Cells are named you wont encounter this potential problem as the Name will move with the Range. Don't misunderstand me here, as I'm not suggesting for a minute you name all Cells within a Workbook but DO name Columns and Rows that you need to remain constant then you can (with reasonable confidence) refer to a particular Cell within the Named Column or Row and access the intended one. Imagine we have Named Row two "**MyRange**" and this Row will contain all our Headings we can refer to the second Cell in this Range like this:

```
Sub ReferToANamedRangesCellMethod1()
    Range("MyRange").Range("B1").Select
End Sub
```

Or

```
Sub ReferToANamedRangesCellMethod2()
    Range("MyRange").Cells(1, 2).Select
End Sub
```

So if a not so helpful user comes along and inserts a Row above our named Row we wont be effected.

You will notice that we are using "Range("B1")" and "Cells(1,2)" to refer to a Range in Row 2 or maybe even Row 500. It doesn't matter where our Named Range "**MyRange**" ends up we would always use the same method to access the second cell within the Range. This is because using the Range (or Cells) Method after the Range Object makes it **Relative**. Perhaps the easiest way to see this is to Record a Macro using the Relative button doing the following:

Selecting the cell immediately to the right of the ActiveCell.

No matter which Cell is the ActiveCell you will always end up with the code:

ActiveCell.Offset(0, 1).Range("A1").Select

Now as with most Recorded code we can edit it, in this case we can use:

ActiveCell.Offset(0, 1).Select

Or

ActiveCell.Range("B1").Select

Or

ActiveCell.Cells(1,2).Select

So when working with the Range Object we have many options open to us.

A common problem faced in VBA for Excel when working with Ranges, is how to find the first and last cell in a range. This is where the number of Rows in an Excel Worksheet comes in very handy. Let's assume you want to find the very last used cell in Column A. In this case we would use:

```
Sub LastUsedCellInColumnA()
    Range("A65536").End(xlUp).Select
End Sub
```

To find the first blank cell in Column A:

```
Sub FirstBlankCellInColumnA()
    Range("A1").End(xlDown).Select
End Sub
```

To find the first blank cell in Row 1:

```
Sub FirstBlankCellInRow1()
    Range("A1").End(xlToRight).Select
End Sub
```

To find the LastUsedCellInRow1:

```
Sub LastUsedCellInRow1()
    Range("IV1").End(xlToLeft).Select
End Sub
```

Once we can find these four points of a range we can apply some VBA to achieve our ultimate aim.

All of the Objects we have discussed above contain many more Object, Properties and Methods than I have shown here. But once you know the Objects hierarchy and how to refer to them it is possible to access all of them. As you become more familiar with VBA you will find that there is virtually nothing that cannot be achieved, all it takes is an open mind as some lateral thinking.

I have included a Workbook example with this lesson that includes all the code mentioned here. It also includes two exercises for you to have a go at. As I have said before, if you solve the problem look at this as the icing on the cake, the cake itself is in trying to solve it.

3. VARIABLES

A Variable is used to store temporary information that is used for execution within the Procedure, Module or Workbook. Before we go into some detail of Variables, there are a few important rules that you must know about.

1. A Variable name must start with a letter and not a number. Numbers can be included within the name, but not as the first character.
2. A Variable name can be no longer than 250 characters.
3. A Variable name cannot be the same as any one of Excel's key words. By this, I mean you cannot name a Variable with such names as Sheet, Worksheet etc.
4. All Variables must consist of one continuous string of characters only. You can separate words by either capitalising the first letter of each word, or by using the underscore characters if you prefer.

You can name variables with any **valid** name you wish. For Example you could name a variable "**David**" and then declare it as any one of the data types shown below. However, it is good practice to formalize some sort of naming convention. This way when reading back your code you can tell at a glance what data type the variable is. An example of this could be the system I use! If you were to declare a variable as a Boolean (shown in table below) I may use: `bIsOpen` I might then use this Boolean variable to check if a Workbook is open or not. The "`b`" stands for Boolean and the "`IsOpen`" will remind me that I am checking if something is open.

You may see code that uses letters only as variables, this is **bad programming** and should be avoided. Trying to read code that has loads of single letters only can (and usually does) cause grief. The only exception I have to this rule is I do use the letter `i` as an Integer variable type. This is because is is very widely recognized as such.

3.1 Variables can be declared as any one of the following data types:

Byte data type

A data type used to hold positive integer numbers ranging from 0 to 255. Byte variables are stored as single, unsigned 8-bit (1-byte) numbers.

Boolean data type

A data type with only two possible values, True (-1) or False (0). Boolean variables are stored as 16-bit (2-byte) numbers.

Integer data type

A data type that holds integer variables stored as 2-byte whole numbers in the range -32,768 to 32,767. The Integer data type is also used to represent enumerated values. The percent sign (%) type-declaration character represents an Integer in Visual Basic.

Long data type

A 4-byte integer ranging in value from -2,147,483,648 to 2,147,483,647. The ampersand (&) type-declaration character represents a Long in Visual Basic.

Currency data type

A data type with a range of -922,337,203,685,477.5808 to 922,337,203,685,477.5807. Use this data type for calculations involving money and for fixed-point calculations where accuracy is particularly important. The at sign (@) type-declaration character represents Currency in Visual Basic.

Single data type

A data type that stores single-precision floating-point variables as 32-bit (2-byte) floating-point numbers, ranging in value from -3.402823E38 to -1.401298E-45 for negative values, and 1.401298E-45 to 3.402823E38 for positive values. The exclamation point (!) type-declaration character represents a Single in Visual Basic.

Double data type

A data type that holds double-precision floating-point numbers as 64-bit numbers in the range -1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values. The number sign (#) type-declaration character represents the Double in Visual Basic.

Date data type

A data type used to store dates and times as a real number. Date variables are stored as 64-bit (8-byte) numbers. The value to the left of the decimal represents a date, and the value to the right of the decimal represents a time.

String data type

A data type consisting of a sequence of contiguous characters that represent the characters themselves rather than their numeric values. A String can include letters, numbers, spaces, and punctuation. The String data type can store fixed-length strings ranging in length from 0 to approximately 63K characters and dynamic strings ranging in length from 0 to approximately 2 billion characters. The dollar sign ($) type-declaration character represents a String in Visual Basic.

Object data type

A data type that represents any Object reference. Object variables are stored as 32-bit (4-byte) addresses that refer to objects. Variant data type A special data type that can contain numeric, string, or date data as well as the special values Empty and Null. The Variant data type has a numeric storage size of 16 bytes and can contain data up to the range of a Decimal, or a character storage size of 22 bytes (plus string length), and can store any character text. The VarType function defines how the data in a Variant is treated. All variables become Variant data types if not explicitly declared as some other data type.

3.2 Why we use variables

Excel will still allow us to run our code without using variables, it is not a **must**! But having said this it is **very bad programming** to not use variables. You could quite easily just assign a value, string or whatever each time you need it, but it would mean:

1. Your code would become hard to follow (even for yourself)
2. Excel would constantly need to look for the value elsewhere.
3. Editing your formula would become awkward.

Let's use an example to highlight the above

```
Sub NoVariable()
Range("A1").Value = Range("B2").Value
Range("A2").Value = Range("B2").Value * 2
Range("A3").Value = Range("B2").Value * 4
Range("B2").Value = Range("B2").Value * 5
End Sub
```

- In the above code, Excel would need to retrieve the value from cell `B2` five times. It would also mean if we had many other procedures using the same value *ie* `B2`, it would need to retrieve it's value even more times.
- There is a lot of editing to be done if we were to change from wanting `B2` value to say, `B5` value.
- It is messy code.

Let's now use a variable to store the value of cell `B2`!

```
Sub WithVariable()
Dim iMyValue as Integer
iMyValue = Range("B2").Value
Range("A1").Value = iMyValue
Range("A2").Value = iMyValue * 2
```

```
Range("A3").Value = iMyValue * 4

Range("B2").Value = iMyValue * 5

End Sub
```

- In the above code Excel only needs to retrieve the value of cell `B2` once.
- To edit our code we only need to change it in one place.
- It is easier to read.

You might be thinking that there is no big difference in the above 2 examples, and to a point you would be correct. But what you must realize is, most VBA projects will consist of hundreds (if not thousands) of lines of code. They would also contain a lot more than one procedure. If you had 2 average size VBA projects, one using variables and one without, the one using variables would **run far more efficiently**!

3.3 Declaring Variables

To declare a variable we use the word "**Dim**" (short for Dimension) followed by our chosen variable name then the word "**As**" followed by the variable type. So a variable dimmed as a String could look like:

```
Dim sMyWord As String
```

You will notice that as soon as we type the word **`As`***, Excel will display a drop-down list of all variables.*

- The default value for any Numeric type Variable is zero.
- The default value for any String type variable is "" (empty text).
- The default value for an Object type Variable is Nothing. *While the default value for an Object type Variable is Nothing, Excel will still reserve space in memory for it.*

To assign a value to a Numeric or String type Variable, you simply use your Variable name, followed by the equals sign (=) and then the String or Numeric type. eg:

```
Sub ParseValue()

Dim sMyWord as String

Dim iMyNumber as Integer

sMyWord = Range("A1").Text

 iMyNumber = Range("A1").Value

End Sub
```

To assign an Object to an Object type variable you must use the key word "**Set**". eg:

```
Sub SetObJect()

Dim rMyCell as Range

Set rMyCell = Range("A1")

End Sub
```

In the example immediately above, we have set the Object variable to the range A1. So when we have finished using the Object Variable "rMyCell" it is a good idea to Set it back to it's default value of Nothing. eg:

```
Sub SetObjectBack()

Dim rMyCell as Range

   Set rMyCell = Range("A1")

      <Some Code>

   Set rMyCell = Nothing

End Sub
```

This will mean Excel will not be reserving unnecessary memory.

In the first example above (`Sub ParseValue()`) we used 2 lines to declare our 2 variables ie

```
Dim sMyWord as String

Dim iMyNumber as Integer
```

We can, if we wish just use:

```
Dim sMyWord as String, iMyNumber as Integer
```

There is no big advantage to this, but you may find it easier.

3.4 Not Declaring Variables

There is a difference between using variables and correctly declaring them. You can if you wish not declare a variable and still use it to store a Value or Object. Unfortunately this comes at a price though! If you are using variables which have not been dimensioned Excel (by default) will store them as the Variant data type. This means that Excel will need to decide each time it (the variable) is assigned a value what data type is should be. The price for this is slower running of code! My advise is do it right and form the good habit early!

There is also another advantage to correctly declaring variables and that is Excel will constantly check to ensure you have spelt the variable name correctly. It does this by capitalizing the all lower case letters that are capitalized at the point it was dimensioned. Let's assume you you use:

Dim iMyNumber As Integer

At the top of your procedure. You then intend to use this variable in other parts of the procedure. Each time you type **imynumber** and then push the **Space bar** or **Enter** Excel will capitalize it for you ie **imynumber** will become **iMyNumber.**This is a very simple and easy way to ensure you have used the correct spelling.

While we are on this subject it is very good practice to type all code in lower case, because not only will Excel do this for variables but also for all Keywords!

There **may** be times when you will actually need to use a Variant data type as you cannot be certain what will be parsed to it, say from cell. It might be text, it maybe a very low or high number etc. In these circumstances you can use:

Dim vUnKnown As Variant

Or, simply:

Dim vUnKnown

Both are quite valid! The reason we do not have to explicitly declare a Variant is because the default for a variable is a Variant.

Scope and Lifetime of Variables

In most instances, when you declare a Variable, you would proceed the Variable name with the keyword "Dim". *The abbreviation "Dim" is short for Dimension.* Depending on where the Variable is declared, will set the scope of where the Variable can be used. By this a Variable "Dimmed" inside a Procedure can only be used within that Procedure. eg;

```
Sub InsideProcedure ()
Dim sMyWord as String
   sMyWord = Range("A1").Text
   <any code>
End Sub
```

In the above example, the Variable "sMyWord" will store within itself whatever text is within range A1. As it has been declared inside the Procedure it will only be available to this Procedure. This is known as declaring a **Variable at Procedure level**. If you try to access the Variable "sMyWord" from within another Procedure, you would encounter a "run time error". This is because as soon as Excel has reached the end of the procedure the variable has been declared in, it destroys its value and reverts back to it's default.

If we now declared the Variable "sMyWord" outside of the Procedure and at the very top of the Module (the Declaration section) it would be available to all Procedures within the same Module. eg:

```
Dim sMyWord as String

Sub OutsideProcedure ()
   sMyWord = Range("A1").Text
   <any code>
End Sub

Sub AnotherProcedure ()
   sMyWord = Range("A2").Text
   <any code>
End Sub
```

In the above example, we could write more Procedures within the same Module and use our Variable "sMyWord". This is known as declaring the **Variable at Module level**.

Another thing to be aware of here is once any Variable has been declared at Module level, and has had a value parsed to it, it will retain that value until such time as it is changed via some code or the Workbook is closed. An example of this may be as below:

```
Dim sMyWord As String

---

Sub InsideProcedure()

sMyWord = Range("A1").Text

'<any code>

End Sub


Sub AnotherProcedure()

MsgBox sMyWord

End Sub

---
```

If you ran the first Procedure "InsideProcedure" and then ran the second Procedure "AnotherProcedure", the Variable sMyWord would still be holding the value of Range A1. Until such time as we change it or close the Workbook.

The final level of declaration for a Variable is known as the **Project level**. What this means is the Variable that is declared at this level will be available throughout all Procedures and Modules within the Project (Workbook). To do this, we must declare the Variable at the Procedure level as above, but instead of using the key word "Dim", we use the word "Public". An example of this is as below:

```
Public sMyWord As String

---

Sub InsideProcedure()

sMyWord = Range("A1").Text

'<any code>

End Sub


Sub AnotherProcedure()

MsgBox sMyWord

End Sub

---
```

We could now write a Procedure in **any Module** within the Project (Workbook) and either access the value of the Variable "sMyWord" or parse a new value to it. *A point to note here is that the variable must be placed at the very top of a **Standard Module**. You could not place it in a **Private Module** and be able to access it in a Standard Module. You can however do the opposite, that is*

place it at the top of a Standard Module and access it from within a Private Module. We will look at Private Modules when we cover Events in a later lesson.

So as you can see, depending on where the Variable is declared, dictates where else we can use the variable without re-dimensioning it ("Dim"). Should you require further information on this, there is a quite a detailed description within the VBE Help under "**Understanding the Lifetime of Variables**".

While it is all too easy to not declare any **Variables** and let Excel decide for you remember, this comes at a cost (Excel by default assumes they are all of the **Variant** type). This mean all your **Variables** are stored at: **22 bytes** (plus string length).

To force yourself into this habit early ensure you have the words "**Option Explicit**" at the top of each **Module**. You can have Excel do this for you by going to **Tools**>**Options** and check the "**Require Variable Declarations**" If this is not on already I urge you to do so! This will force you to declare your variables correctly.

While it is certainly a good habit to declare all Variables correctly, don't fall into the trap of trying to assign the smallest data type to a Variable as you may need to parse a data type which is too big for your Variable to handle. A common one that I have seen happen a lot is declaring a Variable as an Integer and then parsing a Row number to the Variable. In this instance it is very important to realise that an Integer can only range from **-32,768 to 32,767**. If you remember from your last lesson, there are **65,536** Rows in Excel so whenever using a Variable to hold a Row number Value declare it as a Long data type.

3.5 CONSTANTS

Constants are generally used to give a meaningful name to a Value.

An example of this may be a Project that you are working on constantly needs to refer to the Number 35 (for whatever reason). Let's assume in this example the Number 35 refers to a person's age. So rather than type the number 35 each time you require this person's age (for whatever reasons) you could declare a meaningful name such as "BillsAge" to the Value 35. eg:

Const BillsAge as Integer = 35

In the above example, we could use the Constant "BillsAge" throughout the Procedure, Module or Project. Again, as with Variables this is dependent on where the Constant "BillsAge" is declared. So, as with Variables we could make BillsAge available throughout our entire Project by declaring it at the top of a Module, and precede it with the word "Public". eg:

Public Const BillsAge as Integer = 35

Constants are very similar to Variables with one important difference. That is that after a Constant has been declared, **it cannot be modified or assigned a new value**. By this I mean we could not use:

BillsAge as Integer = 40

Anywhere is our Project after we have already set BillsAge to 35 as a Constant.

3.6 EXERCISE

Go to **Tools**>**Options** and uncheck the "**Require Variable Declarations"**

```
Sub Exercise1a ()

iMyNumber = 100

   Range ("A1").Value = iMyNumber

End Sub


Sub Exercise1b ()

   Range ("A1").Value = iMyNumber

End Sub
```

---

1. Open up the VBE and insert a normal Module and paste these two Procedures above in:
2. Declare the Variable to it's correct data type
3. Declare it so it is only available within the Procedure Exercise1a. That is at Procedure Level.
4. When you now run the Sub Exercise1a, the Range A1 of the active sheet should have a value of 100. When you then run the Sub Exercise1b from the same sheet, the value of A1 will be nothing.
5. Now declare the Variable at Module level and run codes Exercise1a and Exercise1b again.
6. This time range A1 should hold the value 100 after both Procedures have run.
7. Now cut and paste Exercise1b into another Module, then go back to Exercise1a and declare the Variable at the Project level. Again, run both codes.

Any problems or questions, just ask

MESSAGE BOX FUNCTION

Excel has available some useful functions that allow us to either inform the user and/or collect information from the user. The most common of these is the Message box function. While this function is very informative it is also very easy to use. For instance, to display a simple message to a user you would only need to use this:

```
Sub MessageBox
```

```
MsgBox "Hello, my name is David."

End sub
```

This is using the message box in it's simplest form. Notice that to tell Excel we want a message box we use the abbreviation **MsgBox**. If we had other code **after** our MsgBox function, our procedure would pause until the user has acknowledged the message.

The syntax for the MsgBox function is:

MsgBox(prompt[, buttons] [, title] [, helpfile, context]**)**

The only part of the MsgBox function that is required is the **Prompt**, all other arguments are optional.

There is a lot of good detailed information on the MsgBox function within the Excel VBA help. Just type MsgBox in any module, select the word then push **F1**. I strongly suggest reading up on this function as it is very useful.

I will use this lesson to show you how you can determine which button they have clicked on a MsgBox should it have **more than one button**. Generally when you wish to 'capture' the return value of a function we need to enclose it within parenthesis. The MsgBox is no exception.

Let's imagine we wish to ask the user if they would like to save their file after a procedure has run. To do this we could use:

```
Sub WhichButton()
Dim iReply As Integer
'<any code>

   iReply = MsgBox("Would you like to save now?", _
          vbYesNo, "ConnectCode")
            If iReply = vbYes Then ThisWorkbook.Save
End Sub
```

This is how you could return to VBA, the button clicked by the user (**Yes** or **No**). Notice how we have used a variable dimmed as a Integer. This is simply because the MsgBox function will return a whole numeric value (Integer) for the button clicked. Each of these Integers that are returned also have a Constant. In our example above, if the user clicks "**Yes**", the Integer returned is **6** and it's Constant equivalent is "**VbYes**". Should the user select "**No**" the value returned would be **7** and the Constant "**VbNo**". It really is that simple!

The values and Constant returned are show in the table below taken from the Excel help:

Constant	Value	Description
vbOK	1	OK

---

vbCancel	2	Cancel
vbAbort	3	Abort
vbRetry	4	Retry
vbIgnore	5	Ignore
vbYes	6	Yes
vbNo	7	No

You may have noticed when you ran the example code, in the **`WhichButton`** procedure, that the default button when the MsgBox was shown was "**Yes**" (first button). If we want to change this we need to assign our "***button***" argument to a variable. We can also use this variable to store our message box type and make our message of the Critical type, or one of many other types.

```
Sub WhichButtonDefault()
Dim iReply As Byte, iType As Integer
'<any code>

    ' Define buttons argument.
      iType = vbYesNo + vbCritical + vbDefaultButton2

        iReply = MsgBox("Would you like to save now?", _
                        iType, "ConnectCode")
           If iReply = vbYes Then ThisWorkbook.Save
End Sub
```

In the example above we have told Excel that we wish to make the "**No**" button our default. This was done with the use of: **`vbDefaultButton2.`** We also told Excel to make our message box Critical and this was done with: **`vbCritical.`** There are many optional arguments for the optional "**buttons**" argument and some are shown in the table below taken from the Excel help.

Constant	Value	Description
vbOKOnly	0	Display **OK** button only.
vbOKCancel	1	Display **OK** and **Cancel** buttons.
vbAbortRetryIgnore	2	Display **Abort**, **Retry**, and **Ignore** buttons.
vbYesNoCancel	3	Display **Yes**, **No**, and **Cancel** buttons.
vbYesNo	4	Display **Yes** and **No** buttons.
vbRetryCancel	5	Display **Retry** and **Cancel** buttons.
vbCritical	16	Display **Critical Message** icon.
vbQuestion	32	Display **Warning Query** icon.
vbExclamation	48	Display **Warning Message** icon.
vbInformation	64	Display **Information Message** icon.

3.7 INPUT BOX

Another very handy function is the **InputBox** function. This will allow us to collect a reply from a user in the form of text. Its syntax is:

InputBox(prompt[, title] [, default] [, xpos] [, ypos] [, helpfile, context]**)**

The only required argument is the **Prompt** and this would be the message you display to the user.

Try this example below:

```
Sub WhatsYourName()
Dim stReply As String
'<any code>

    stReply = InputBox(Prompt:="What is your name?", _
              Title:="ConnectCode", Default:="Mine is David!")
      If stReply <> "" _
       And stReply <> "Mine is David!" Then
          MsgBox "Hello " & stReply
      End If

End Sub
```

In the above example we are asking the user to tell us their name. We have put a default message in the InputBox telling them our name. As with the message box function, we are using a variable to determine their reply. The difference is though, the InputBox returns a **String** and **not an Integer!** If the user decides to **Cancel** the InputBox it would return empty text (""), if they simply clicked **Ok** without changing our **Default** message it would return: "**`Mine is David`!". Both of these possibilities are dealt with in our **`If And`** combination.

There is also another type of InputBox, which is a member of the Application Object. This InputBox will allow us to specify what type of value to pass back to our InputBox. The syntax for this InputBox is:

expression*.InputBox(*Prompt, Title, Default, Left, Top, HelpFile, HelpContextId, Type**)**

Again the only required argument is the **Prompt**. The important part to note with this type of InputBox is that you must precede it with an Application Object, which in Excel's case in the actual word **Application**

The return value type is specified by the Type argument and is shown in the table below taken from the Excel help.:

Value	Meaning
0	A formula
1	A number
2	Text (a string)
4	A logical value (True or False)
8	A cell reference, as a Range object
16	An error value, such as #N/A
64	An array of values

You can use the sum of the allowable values for ***Type***. For example, for an input box that can accept both text and numbers, set ***Type*** to 1 + 2.

So to use the InputBox to return a range Object you could use:

```
Sub Demo()

Dim rMyRange As Range

On Error Resume Next
   Set rMyRange = Application.InputBox _
   (Prompt:="Select any range", Title:="ConnectCode", Type:=8)
      rMyRange.Select
On Error GoTo 0
End Sub
```

Notice the use of `On Error Resume Next` this is used to prevent any possible run time errors that **would occur** if the user clicks **Ok** (without select a range) or **Cancel**. If they did, our Set Statement would not be valid. I have then used the `On Error GoTo 0` to reset all run time errors. The use of these two Statements are discussed in a later lesson.

As you can see, with the use of Excels built in function such as MsgBox and InputBox, we are able to communicate with the user. More importantly, we can collect their replies and answers.

4. LOOPS

Please cross reference the Loops.xls Workbook for this lesson.

This lesson we will focus on Loops. There are many varieties of these, but they are all basically the same in that they will repeat a line, or multiple lines, of code a set number times, or until a condition becomes **True** or **False**. Excel VBA has 5 loops from which we can choose from. Depending on the situation they are to be used for would dictate the type of loop to choose. Some loops are best suited for looping while incrementing a number, while others are ideal for looping through anyone of Excels Objects. Arguably 2 of the most useful loops would be the **For** loop and the **For Each** loop.

Shown below are the 5 loops available to us in Excel VBA.

Do
<Code to repeat>
Loop While <Whatever>

Do While <Whatever>
<Code to repeat>
Loop

Do Until <Whatever>
<Code to repeat>
Loop

For <Variable>= <Any Number> **To** <Any Other Number>
<Code to repeat>
Next <Variable>

For Each <Object Variable> **in** <Object Collection>
<Code to repeat>
Next <Object Variable>

I realise that loops can seem very confusing the first time you encounter them, but they really are not that complicated. Just remember, all they are doing is what you have instructed them to do (via code) either a set number of times or until a condition is met (**True** or **False**).

Let's look at each one in turn and use them in a simple way.

4.1 Do Loop

```
Sub OurDo()
   Do
   <Code to repeat>
   Loop While <Whatever>
End Sub
```

The very first word here is "Do" so we now have to tell our Loop exactly what to Do. For this example we will add the number one to a Integer Variable we called "iMyNumber", so we can put:

```
Sub OurDo1()
Dim iMyNumber as Integer
   Do
   iMyNumber=1+iMyNumber
   Loop While <Whatever>
End Sub
```

Ok, so we have now told our "**Do**" what it is we want to do. Now we have to tell it when to stop. To achieve this we must set a condition for our "**Loop While**". In this case we will: Do iMyNumber=1+iMyNumber until iMyNumber is > 100. So all we need to do is:

```
Sub OurDo2()
Dim iMyNumber as Integer

   Do
   iMyNumber=1+iMyNumber
   Loop While iMyNumber < 100

End Sub
```

So the bottom line here is the **Do Loop** will Loop 101 times.

4.2 Do While

This is very similar to our Do Loop we just used above. The only difference is, instead of the condition we set (iMyNumber < 100) being checked **AFTER** the Do has run at least once, the **Do While** will check the condition **BEFORE** it runs. Let's say our Variable iMyNumber has already had

the Value 101 parsed to it. In the **Do** Loop we used above, it will NOT know the Value of iMyNumber until it has **run ONCE**. This would mean our Variable would come out of the **Do** Loop with a Value of 102. But in the **Do While** below, it would never even enter the Loop:

```
Sub OurDoWhile()
Dim iMyNumber as Integer

   Do While iMyNumber < 100
   iMyNumber=1+iMyNumber
   Loop
End Sub
```

4.3 Do Until

Again this is very similar to the **Do While** Loop we just used above in that it will check the condition **BEFORE** it enters the Loop.

If the Value of iMyNumber is 0 (zero) when it reaches the Loop, the difference is the **Do While** would keep adding the number one to iMyNumber until it reached 100. In the **Do Until** it would never even enter the Loop because the condition MyNumber < 100 has been met already ie; iMyNumber is 0 (zero).

```
Sub OurDoUntil()
Dim iMyNumber as Integer

   Do until iMyNumber < 100
   iMyNumber=1+iMyNumber
   Loop
End Sub
```

4.4 For

The For Loop is **perhaps** the most useful of all the Loops. It runs a line or lines of code a set amount of times in any increment we set. The default increment is one. As you can see from below you **must** use a Variable of the Numeric type to set the amount of Loops it will perform.

```
Sub OurFor()
Dim iMyNumber as Integer
```

```
    For iMyNumber= 1 To 100
    iMyNumber=1+iMyNumber
    Next iMyNumber

End Sub
```

The above **For** Loop will simply Loop through the code:

iMyNumber=1+iMyNumber

101 times. As we have set the condition we want met ie; increment iMyNumber by one, 100 times, we do NOT need to keep adding one to our Variable iMyNumber. This will happen automatically as the default increment is one. Now I know you are thinking why does it Loop 101 times and not 100? Simply because the default value for a Variable of the Numeric Data type is 0 (zero). This means that it must run once before "iMyNumber" will have the value of one.

We also do NOT need to increment our Variable by one each Loop with:

iMyNumber=1+iMyNumber

This is because we have already told our Loop this by using the **For** Loop type. So we could use:

```
Sub OurFor()
Dim iMyNumber as Integer

    For  iMyNumber= 1 To 100
    iMyNumber
    Next iMyNumber
End Sub
```

...and iMyNumber will still end up with the Value of 101. In fact we could even use:

```
Sub OurFor()
Dim iMyNumber as Integer
Dim iMyNumber2 ad Integer

    For iMyNumber= 1 To 100
    iMyNumber2 =1+iMyNumber2
    Next iMyNumber

End Sub
```

....and we would get the same result. You can see this for yourself by the use of a Message Box.

```
Sub OurFor()
Dim iMyNumber as Integer
Dim iMyNumber2 ad Integer

   For iMyNumber= 1 To 100
   iMyNumber2 =1+iMyNumber2
   Next iMyNumber

MsgBox iMyNumber

End Sub
```

The other great part about the **For** Loop is we can increment by any Value we like. We do this by using the Step Key word and telling it the Step (or increment) to use. so we could use:

```
Sub OurFor()
Dim iMyNumber as Integer
Dim iMyNumber2 as Integer

   For iMyNumber= 1 To 100 Step 2
   iMyNumber2 =1+iMyNumber2
   Next iMyNumber

MsgBox iMyNumber

End Sub
```

By doing this we will Loop through our code 51 times instead of 101 times, but the Variable iMyNumber will end up with a Value of 101.

We could also use the Step Key word to work backwards like below:

```
Sub OurFor()
Dim iMyNumber as Integer
Dim iMyNumber2 as Integer

   For iMyNumber= 100 To 1 Step -1
   iMyNumber2 =1+iMyNumber2
   Next iMyNumber
```

```
MsgBox iMyNumber

End Sub
```

This would mean that our Variable iMyNumber would end up with a Value of 0 (Zero).

4.5 For Each

This Loop is slightly different from the others, but only in the fact that it requires an Object as the Variable. What it does is simply Loop though each single Object in a Collection of Objects.

```
For Each <Object Variable> in <Object Collection>
<Code to repeat>
Next  <Object Variable>
```

To put this into something meaningful, we could use:

```
Sub OurForEach()
Dim rMyCell As Range
Dim iMyNumber2 As Integer

   For Each rMyCell In Range("A1:A100")
   iMyNumber2 = 1 + iMyNumber2
   Next rMyCell

MsgBox iMyNumber2

End Sub
```

What this is saying in Layman's terms is:

For Each Cell in the Range A1 to Range A100 add 1 to the Variable iMyNumber2 Where "Cell" is represented by the Range Variable "rMyCell" So it will do this 100 times as there are 100 Range Objects in the Object Collection Range("A1:A100")

We do not need to tell the **For Each** Loop how many times to Loop as it already knows how many Objects (Cells in this case) there are in the Object Collection (**Range(A1:A100)**).

Our Object Collection does not have to be a Range Collection, it could be a Charts, Worksheets,

Workbooks etc Collection. In fact it can be any Collection of Objects. So if we wanted to Loop through all Worksheets in a Workbook we could use:

```
Sub OurForEach()
Dim wWsht As Worksheet

   For Each wWsht In ThisWorkbook.Worksheets
   wWsht.Range("A1") = wWsht.Name
   Next wWsht

End Sub
```

This would Loop through each Worksheet in the Workbook and place the name of the Worksheet in cell A1 of each.

So as you can see one of the big advantages with a **For Each** loop is that we do not need to know how many Objects are within the Object collection that we wish to loop through.

4.6 Inner and Outer Loops

Any type of Loop can have more than one level. This is very similar to Nesting Worksheet formulas on a Worksheet. There is no limit (except memory) of the level to which you can Nest loops. To keep things simple though we will only look at a two level Loop. Let's say we want to Loop through all cells in the Range A1:A10 on each Worksheet and place the address of the cell in each cell. To do this we would use:

```
Sub OurForEach()
Dim wWsht As Worksheet
Dim rMyCell As Range

For Each wWsht In ThisWorkbook.Worksheets
 For Each rMyCell In wWsht.Range("A1:A10")
  rMyCell.Value = rMyCell.Address
 Next rMyCell
Next wWsht

End Sub
```

When you have two Loops Nested like this, you would refer to them as the **Outer Loop** and **Inner**

Loop. With "**For Each wWsht In ThisWorkbook.Worksheets**" being the Outer Loop and "**For Each rMyCell In wWsht.Range("A1:A10")**" being the Inner Loop. What will happen is the code would first encounter the **outer loop** and know that is has to loop through whatever code is within it the same amount of times as there are Worksheets in the Workbook. As soon as it **enters the outer loop** it encounters the **inner loop**, it then knows it must loop through any code within it 10 times (there are 10 range Objects in range A1:A10). Once it has done the 10 loops it **leaves the inner loop**, it then continues on with the **outer loop** which in turn immediately makes it move onto the next Worksheet Object (wWsht) and the cycle starts again.

4.7 Exiting a Loop

In all the above Loop examples we have allowed the loop to continue on until the loop condition is met. But there are times when we may wish to force our loop to leave a loop early. This is done by using the Exit Statement. Let's assume we wish to loop through a range of cells and select a cell if it's value is 100.

```
Sub ExitALoop()
Dim rMyCell As Range

   For Each rMyCell In Range("A1:A10")
      If rMyCell.Value = 100 Then
         rMyCell.Select
         Exit For
      End If
   Next rMyCell
End Sub
```

This loop will only loop 10 times (A1:A10) if no cell within range A1:A10 is equal to 100. If there is a cell within range A1:A10 that is equal to 100 it will select it, then **Exit** the **For Each** loop.

We can also use the **Exit** Statement on any of the Do loops by simply using:

```
Exit Do
```

So that is basically all there is to Loops. Used in the context as shown above would not really be of much use, but it is **far more important** to understand the concept of them than to use them without knowing how they work. The only other part of Loops that you will most likely encounter is what is known as the **endless Loop**. This occurs when you start a loop that will never meet the condition you have set and so it just keeps going around endlessly. When this happens you need to push **Ctrl**+**Break** or **Esc**.

4.8 My Personal Experience

While Loops are without doubt a very handy feature of VBA they are the most misused piece of code in VBA. In all the above examples the Loop will complete in less than a second. But in most real world situations you will need to Loop a lot lot more than 100 times this may be as high as

1000000 times or more! When you try to Loop any more than 1000 times and are writing to cells or some other operation you will notice a time lag. All too often I see code that Loops through thousands of Cells just to look for ten or more cells that meet the condition they are after. This is either due to ignorance and/or laziness. There are many alternatives to Loops in these cases that will do the same job, but about 100 times (or more!) quicker and **more efficiently**. But people tend to stick with what they know or the way they have always done things.

4.9 Don't Fall Into The Loop Yourself

I often use this expression to try and drive home the point that Loops can (and do!) draw people in. A lot of people, when they first get the hang of Loops, find they can use them for a whole range of solutions that they previously THOUGHT was not possible. This then draws them into the Loop where they themselves become trapped. I too used to do the same, but when I started Looping through thousands of cells just to find the few I wanted and it took anything from 10 secs to more than 5 mins, I thought that this just cannot be right. You see when you use Excels Find function (Edit>Find) it searches through ALL 16777215 cells and finds what you want in the blink of the eye, no matter which cell it is in. You have a list of data entries and you only want to see all the entries that start with the letter "**A**", you use AutoFilter and again in the blink of an eye you have them. So initially I started to try and fathom out exactly how these built in Function it did it. Then the penny dropped! Instead of trying to mimic the functions like Find, AutoFilter and the Worksheet Functions to name but a few, I decided I would work them into my code and leave the Loops out altogether. At first this this meant I had to 'Step outside the box' as they say and actually look at a problem from a different angle altogether. But now I will go to no ends to try and find an alternative to some Loop code that needs to Loop anymore than 1000 times.

4.10 Short Term Pain For Long Term Gain

There is no mistaking that trying to find an alternative to a Loop will mean sitting back and thinking a lot harder. It will also mean trying a lot of alternate methods to a Loop that wont work at all and/or give Run Time Errors etc. But believe me, persistence WILL pay off. Once you have done this quite a few times you will realise that Loops only need play a very small part in your programming. Then, like me, you will no doubt look for any other method except a Loop. To date I have been able to use an alternative to a Loop about 80% of the time. I often run into a problem that requires dealing with thousands of Cells, Rows etc. Years ago I would have used a Loop, but now a Loop is something I only use as a last resort. Don't misunderstand me, if I'm only dealing with hundreds rather than thousands and I cannot think of an alternative, I wont ponder on it for too long before using the Loop. But the heading of this paragraph is very true.

4.11 Sometimes You Just Have To Loop

As I have stated above, 80% of the time I can avoid a Loop, but there is that 20% when I have to bite the bullet and use one. If this is the case I will immediately try to think of a way I could narrow down the field to Loop through. Let's say I need to search through a range of 10000 cells (A1:D2050) that contain a mix of numbers, text and formulas and Clear all the cells that contain formulas and have a value of greater than 100. Firstly I'm only interested in Numeric cells and of those I'm only interested in Formula cells. So ideally I would like to narrow the field down to Numbers and Formulas. We can easily do this by setting a Range Object to all cells in the range A1:D2050 that are Formulas and Numeric. We do this by using Excels SpecialCells Method:

```
Sub LoopThroughNumericFormulas()
Dim rNumFormulas as Range

   Set rNumFormulas = _
   Range("A1:D2050").SpecialCells(xlCellTypeFormulas, xlNumbers)
End Sub
```

Now that we have narrowed the field down we can Loop through the cells in our Range Variable "**rNumFormulas**" and Clear all cells that are greater than 100. We would do this like below:

```
Sub LoopThroughNumericFormulas()
Dim rNumFormulas As Range
Dim r100OrGreater As Range

   Set rNumFormulas = _
   Range("A1:D2050").SpecialCells(xlCellTypeFormulas, xlNumbers)

   For Each r100OrGreater In rNumFormulas
      If r100OrGreater.Value > 100 Then r100OrGreater.Clear
   Next r100OrGreater

Set rNumFormulas = Nothing
End Sub
```

This could potentially make our Loop run in less than 1 second as apposed to 5 to 10 seconds or even longer. There many other methods that I use which are similar to this. Below is a list of Excel built in features that can quite often be used either in place of, or to narrow down a Loop.

- **AutoFilter**
- **AdvancedFormulas**
- **Find**
- **SpecialCells**
- **Sort**
- **SubTotals**
- **AutoFill**
- **WorksheetFunction**
- **TextToColumns**
- **Group**

I have included with this lesson some examples of Loops and how they might be used. I will also include a Loop that CAN be narrowed down and done a far quicker way. Don't worry too much if

you cannot achieve it, I would be very surprised if you can! The loop will take a while to run, but I want to show you a comparison.

5. EFFECTIVE DECISION MAKING

5.1 IF Else And Or Not If

The "If" Function in VBA for Excel is very similar to the "IF" function used in a Worksheet formula. It will return either True or False and it does no more or less than this. As with the "IF" used in the Worksheet formula the "If" in VBA can take up to two arguments, one for True and one for False. So the syntax for the "If" is simply:

```
If <Condition to check> Is True Then
'Do one thing
Else
'Do another thing
End If
```

So this same Function used in a realistic way could be

```
Sub TheIfFunction()
   If Range("A1").Value > 100 Then
         Range("B1").Value = 50 + Range("A1").Value
   Else
         Range("B1").Value = 100
      End If
End Sub
```

This is telling Excel that If the Value of A1 is greater than 100 (True) then change the Value of B1 to the Value of A1 plus another 50, Else (False) changes the Value of B1 to 100. This would be the "If" Function used in it's simplest form. Once Excel encounters the "If" Function it will check the Value of A1, if the value is greater than 100 it will enter into the True argument:

```
Range("B1").Value = 50 + Range("A1").Value
```

From there it will Exit the "If" or in other words it will skip the False argument:

```
Range("B1").Value = 100
```

But lets assume we wanted Excel to check if Range A1 is equal to 500 first and only go on if it's not (False), to achieve this we would need to extend the "If" so it will possibly check two conditions before exiting the remainder of the "If". This is how we could do this:

```
Sub TheIfFunction()
   If Range("A1").Value = 500 Then
      Range("C1").Value = 100 - Range("A1").Value
```

```
    ElseIf Range("A1").Value > 100 Then
        Range("C1").Value = 50 + Range("A1").Value
    Else
        Range("B1").Value = 100
    End If
End Sub
```

This Function is saying that, If Range A1 is equal to 500 (True) then:

`Range("C1").Value = 100 - Range("A1").Value`

But If Range A1 is NOT equal to 500 then check another condition, which is:

`ElseIf Range("A1").Value > 100 Then`

If this is True then:

`Range("C1").Value = 50 + Range("A1").Value`

Finally if neither of these conditions are True then:

`Range("B1").Value = 100`

We could in theory keep adding an unlimited amount of "ElseIf" Functions to check for multiple conditions. The problem with this is that our "If" Function would become almost impossible to read and more importantly, decipher. I will show you a much better method soon for checking multiple conditions, but for now we will stick with the "If" Function.

There are two other common Keywords used in conjunction with the "If" Function, they are the "And" and the "Or" Operators. We will look first at the "And" operator.

The "And" operator is use to perform a conjunction of two conditions. Whenever we use the "And" operator with the "If" Function it will only ever return True if BOTH conditions are met (True and True). So if we used the "If" combined with the "And" like below:

```
Sub TheIfAndFunction()
    If Range("A1").Value > 100 And Range("A1").Value < 500 Then
        Range("B1").Value = Range("A1").Value
    End If
End Sub
```

This would tell Excel that If Range A1 is between 100 and 500 (True) then make:

`Range("B1").Value = Range("A1").Value`

If Range A1 is Not between 100 and 500 do nothing. We could again add an unlimited amount of "And" operators all checking different conditions, but again this would become very hard to decipher and is also not very efficient.

The other common Operator used with the "If" Function is the "Or" Operator. This will check if one of two conditions are True and return True if only one of them is met (True and False) or (False and True). Below is an example of this:

```
Sub TheIfOrFunction()
    If Range("A1").Value = 100 Or Range("A1").Value = 500 Then
        Range("B1").Value = Range("A1").Value
    End If
End Sub
```

This "If" Statement will return True If Range A1 is equal to 100 OR If Range A1 is equal to 500 any other condition would return False and do nothing.

These two Operators are by far the most commonly used Operators used with the "If" Function.

So the "If" Function can be used to determine whether a Function is either True or False and act accordingly. Combining it with the Operators "And" and "Or", can extend it's functionality. In all the above examples the "If" Functions use the "End If" Keywords. These simply let Excel know that the "If" Function has finished. If we restrict our "If" Function to one line of code only we can omit the "End If" completely, like below:

```
Sub NoEndIf()
If Range("A1").Value = 100 Then Range("B1").Value = 20
End Sub
```

This can at times make your code slightly easier to read. There is no performance gain by doing the "If" Function this way, so don't get caught in the trap of always trying to fit your "If" Function onto one line. If by doing so you cannot read the entire line without scrolling to the right use two or more lines with the "End If".

There is one other way of evaluating a condition with the "If" and that is called the "Iif". I will only show you this because it exists, but I do not recommend using it for two reasons.

It's slightly slower

It has no advantage over the "If Else"

The syntax for IIf is:

```
IIf(Condition, True part, False part)
```

To use this in a similar way as the "If", we could use:

```
Sub TheIIf()

IIf Range("A1").Value = 100, Range("B1").Value = 20, Range("B1") = 50

End Sub
```

But as I have said I would avoid using this as it holds no advantage.

The other operator we can use with the "If" Statement is the "Not" Statement. This is used to reverse the "If" Statement

```
Sub IfNot()

   If Not Range("A1") = 100 Then

      MsgBox "Not 100", vbInformation, "ConnectCode"

   End If

End Sub
```

In the example above we have used the "Not" statement to reverse the logic of the "If" statement. By this I mean we have told our "If" statement to return True if Range A1 is not equal to 100

Select Case

The other method of checking for single or multiple conditions is the Select Case Function. This is another method used for decision making based on a condition or criteria. It, in my opinion, is much better than If etc. This has the syntax

```
Select Case <Expression to test>
   Case <Test1>
      Do something
   Case <Test2>
      Do something
   Case Else
      Do something else
End Select
```

As you can see the "Select Case" Function is very similar to the "If" Function in that it will only perform some action if a condition is met. While this may seem no better than the "If" Function I feel that it is a MUCH better choice than the "If" Function If more than one condition or expression needs to be tested. Not only is it more efficient but it has a much better structure than the "If" Function. This means it is far easier to read or decipher and believe me you WILL need to go back through your written code frequently to find out a problem (De-bug). While these two reasons alone are enough for me, there is another and that is it has FAR more flexibility. We will first look at the "Select Case" Function in it's simplest form

```
Sub TheSelectCase()

   Select Case Range("A1").Value

      Case 100

         Range("B1") = 50
```

```
    End Select
End Sub
```

As you may have noticed, this does not include a "Case Else" Statement. This is because, like the "Else" Statement in the "If" Function it is optional. The "Select Case" in it's simplest form as shown above, holds no advantage over the "If" Function, in fact it could be argued that it is an incorrect use of the "Select Case" Function. Let us say you need to perform any one of 5 actions depending on the Value of Range A1. If so we could use:

```
Sub TheSelectCase()
    Select Case Range("A1").Value
        Case 100
        Range("B1").Value = 50
        Case 150
        Range("B1").Value = 40
        Case 200
        Range("B1").Value = 30
        Case 350
        Range("B1").Value = 20
        Case 400
        Range("B1").Value = 10
    End Select
End Sub
```

This, in my opinion, is a far better structure and easier to read than an "If" Function with multiple "ElseIf" Statements. If none of the above Conditions were met nothing would occur, unless we use the optional "Case Else" Statement, like:

```
Sub TheSelectCase()
    Select Case Range("A1").Value
        Case 100
            Range("B1").Value = 50
        Case 150
            Range("B1").Value = 40
```

```
    Case 200
        Range("B1").Value = 30
    Case 350
        Range("B1").Value = 20
    Case 400
        Range("B1").Value = 10
    Case Else
        Range("B1").Value = 0
    End Select
End Sub
```

So If the Value of Range A1 is NOT 100,150,200,350 or 400 then place a Value of 0 (zero) in Range B1. Now while this demonstrates how we can check multiple conditions with the "Select Case" Function, what if we want to perform some action If the Range A1 is equal to any one of the Values 100,150,200,350 or 400. If this is the case (no pun indented) we could use:

```
Sub TheSelectCase()
  Select Case Range("A1").Value
    Case 100, 150, 200, 350, 400
      Range("B1").Value = Range("A1").Value
    Case Else
    Range("B1").Value = 0
  End Select
End Sub
```

I don't believe anybody could argue against this being a far better structure than an "If" Function with multiple "Or" Operators.

We used the "If" Function combined with the "And" operator above to demonstrate how to let Excel know if the Value of Range A1 is between two numbers. We can do this also with the "Select Case" Function with even greater ease:

```
Sub TheSelectCase()
  Select Case Range("A1").Value
    Case 100 To 500
      Range("B1").Value = Range("A1").Value
    Case Else
```

```
        Range("B1").Value = 0
    End Select
End Sub
```

As the above example demonstrates, we use the Keyword "To" to test whether Range A1 is between 100 and 500 and if so place the value of Range A1 in Range B1. We can take this a step further if needed to test for multiple "To" conditions, like:

```
Sub TheSelectCase()
  Select Case Range("A1").Value
    Case 100 To 500, 600 To 1100, 1200 To 2000
      Range("B1").Value = Range("A1").Value
    Case Else
      Range("B1").Value = 0
  End Select
End Sub
```

So now with this one "Select Case" Function we can check to see if Range A1 is between any one of three different Values. We could if we needed add more. Another major advantage of the "Select Case" Function over the If Function is that we can use it to determine if certain Text is between two other Text Strings ie; alphabetically. Let's assume you are only interested in the content of Range A1 if the Text within it is alphabetically between "Aardvark" and "Elephant", to do this we could use:

```
Sub TheSelectCase()
  Select Case Range("A1").Text
    Case "Aardvark" To "Elephant"
      Range("B1").Value = "it's between"
    Case Else
      Range("B1").Value = "it's not between"
  End Select
End Sub
```

So if the text in Range A1 is "Budgie" range B1 will read "It's between". If the Text in A1 is "Zebra" then Range B1 would read "It's not between".

So as you can see there are many ways within VBA for Excel we can use to evaluate and determine a Value or Text. You will find yourself using the "If" and "Select Case" Functions quite frequently and as I have already indicated, the Select Case is often a much better option.

5.2 Excel and Dates

Dates are frequently used in Excel and in VBA for Excel, also for this reason I believe it is an important aspect to at least know the fundamentals of. The text below is from the Excel help file and explains how Excel sees or interprets Dates.

How Microsoft Excel performs date and time calculations

Microsoft Excel stores dates as sequential numbers known as serial values and stores times as decimal fractions because time is considered a portion of a day. Dates and times are values and therefore can be added, subtracted, and included in other calculations. For example, to determine the difference between two dates, you can subtract one date from the other. You can view a date or time as a serial number or a decimal fraction by changing the format of the cell that contains the date or time to General format.

End of Excel Help

Using Dates in VBA for Excel offers a lot more flexibility, but also has more pitfalls that can catch the uninformed out. This is due mainly to the fact that Excel is used globally and there is more than one Date system. We will look at the two most common and that is the American (Month-Day-Year) and the European (Day-Month-Year). This issue will be nearly non-existent if you know for a fact that the code written in VBA will only be used on one Date system. But should you write a Procedure that will be used by more than one Date system, problems can arise. In today's market it is not unusual for say an English company to have to deal with a Spreadsheet that uses the American Date System or an vice versa. You can imagine the problems that could arise if the Dates you insert into a Spreadsheet via VBA are assumed to be of the American Date System when in reality they are of the European Date System. Fortunately the makers of VBA for Excel have realised this and provided a universal Function to eliminate possible disasters. I would urge you to again form a good habit early and incorporate it whenever dealing with Dates. The Function is called the "DateSerial" Function. It has the Syntax:

```
DateSerial(year, month, day)
```

As you can see the Function uses neither the American or European Date System. Let's say you want to insert the Date 5-6-2001 (European System) into cell A1, you could use:

```
Sub UniversalDate()
Dim dTheDate As Date
    dTheDate = DateSerial(2001, 6, 5)
    Range("A1").Value = dTheDate
End Sub
```

Using this Function will eliminate any possible confusion of the Date System used. Whenever you use or Parse a Date to a Variable or Range you must enclose it within the # (Hash signs) eg;

```
dTheDate = #22/5/01#
```

If you try and type this into Excel exactly as is you will see that Excel will automatically change it to the American Date System whether you want it to or not. In other words it will end up like:

dTheDate = #5/22/01#

You may or may not notice the Month and Day being switched if you are happily typing away. But worse than this is if you use:

```
Sub UniversalDate()
Dim dTheDate As Date
    dTheDate = #10/12/01#
    Range("A1").Value = dTheDate
End Sub
```

….and you are used to the European Date System you would assume the Date going into Range A1 is the 10th Day of December, 2001. Guess what, WRONG! You will actually end up with the 12th Day of October, 2001 instead. Of course if we aware of this and we have formed the good habit of using the "DateSerial" Function for all Dates no such problem will arise. Eg;

```
Sub UniversalDate()
Dim dTheDate As Date
    dTheDate = DateSerial(2001, 12, 10)
    Range("A1").Value = dTheDate
End Sub
```

I hope this stresses the importance of using the DateSerial Function whenever you are working with Dates in Excel

Date Functions

Now that we realise the possible pitfalls of working with Dates I will show you some handy Functions you can use in Excel when working with Dates.

The first one is the "Date" Function, this will return the current systems Date. As it is returning the current system Date, it will use the Date System of the PC it is run on.

Next we have the "DateValue" Function. This is the same as the Worksheet Formula DATEVALUE in that it will return a real Date from a String Date, eg;

```
Sub UniversalDate()
Dim dTheDate As Date
    dTheDate = DateValue("12/May/01")
    Range("A1").Value = dTheDate
```

```
End Sub
```

This will Parse the Date Value for 12th day of May, 2001 to our Date Variable "dTheDate", which in turn will place it in cell A1. You may also have noticed that we could also use the "DateValue" function to prevent mishaps when working with Dates. There is no reason why you couldn't and it won't matter so long as you use one or the other.

To add a specified period of time to a Date we could use the "DateAdd" Function, this has the Syntax:

```
DateAdd(interval, number, date)
```

You would use it like below:

```
Sub DateAddFunction()
Dim dTheDate As Date
    dTheDate = DateAdd("d", 45, "18/may/2001")
    Range("A1").Value = dTheDate
End Sub
```

In this particular case the Date Variable dTheDate would return the Date 45 days from the 18th day of May, 2001, which is the 2nd of July, 2001. Although the Function is called "DateAdd" in can be used to subtract a specified time period from a Date. To do this you would simply use a negative number. The other allowed arguments for the Interval are:

yyyy	Year
q	Quarter
m	Month
y	Day of year
d	Day
w	Weekday
ww	Week
h	Hour
n	Minute
s	Second

There are many other types of Functions that can be used with Dates but as this is level 1 VBA I won't confuse the issue by going into them all. The two most important points to remember when working with Dates are How Microsoft Excel performs date and time calculations and being aware of the International issues associated with Dates.

As always, if you have any question's on this lesson or past lessons, please feel free to contact me.

6. WORKBOOK & WORKSHEET EVENTS

Please cross reference the EventExamples.xls Workbook for this lesson.

As are probably aware you can have Excel run a macro by using a CommandButton, AutoShape, Shortcut key etc. Most of the time this is fine as you only want to run the macro when you instruct it to run. But there are times when you may like or want the macro to run whenever a particular Event happens. Let's say each time you open your Workbook you would like the current date inserted into a cell, or each time you activate a particular Worksheet you want all data on it to be cleared. With Excel we can do this and much more! Excel has what is known as **Events**. The two most common being either **Workbook Events** or **Worksheet Events**. The word Event in this context means something that occurs whenever a particular action takes place which effects the Workbook or Worksheet Object. This might be a Workbook opening or a Worksheet cell changing or a Workbook closing etc. In fact Excel 2000 has **20 Workbook** Events and **8 Worksheet** Events. These are:

Workbook Events

- Workbook_Activate
- Workbook_AddinInstall
- Workbook_AddinUninstall
- Workbook_BeforeClose
- Workbook_BeforePrint
- Workbook_BeforeSave
- Workbook_Deactivate
- Workbook_NewSheet
- Workbook_Open
- Workbook_SheetActivate
- Workbook_SheetBeforeDoubleClick
- Workbook_SheetBeforeRightClick
- Workbook_SheetCalculate
- Workbook_SheetChange
- Workbook_SheetDeactivate
- Workbook_SheetFollowHyperlink
- Workbook_SheetSelectionChange
- Workbook_WindowActivate
- Workbook_WindowDeactivate
- Workbook_WindowResize

Worksheet Events

- Worksheet_Activate
- Worksheet_BeforeDoubleClick
- Worksheet_BeforeRightClick
- Worksheet_Calculate
- Worksheet_Change
- Worksheet_Deactivate

- Worksheet_FollowHyperlink
- Worksheet_SelectionChange

As you can see Excel offers a very rich environment for full automation if we wish. As there are so many of these Events we look in detail at the most popular and arguably the most useful of them. We will start with the Workbook.

6.1 Workbook Events

The most popular Events for the Excel Workbook are:

- Private Sub Workbook_Open() Occurs when the workbook is opened
- Private Sub Workbook_BeforeClose(Cancel As Boolean) Occurs immediately before the workbook closes. If the workbook has been changed, this event occurs before the user is asked to save changes.
- Private Sub Workbook_BeforeSave(ByVal SaveAsUi As Boolean, Cancel As Boolean) Occurs immediately before the Workbook is saved.
- Private Sub Workbook_NewSheet(ByVal Sh As Object) Occurs when a new sheet is created in the workbook.

You may have noticed that ALL these Events start with the words "**Private Sub**". This means they are only available to the **Workbook Object Module**. This is always called "**ThisWorkbook**" in the "Project Explorer". You can double click "ThisWorkbook" to gain access to the Workbooks Module. If you are not in the VBE you can right click on the sheet picture, top left next to "**File**" and select "View Code". In Excel 97 the default will be:

```
Private Sub Workbook_Open()

End Sub
```

If you are using Excel 2000 + the default will be blank. You can still save yourself the problem of having to type out the Workbook Event you want by clicking the drop arrow on the "**Object**" box (top left of the Module) and selecting "Workbook". This will then default to:

```
Private Sub Workbook_Open()

End Sub
```

In both versions 97 and 2000 + all the Events for the Workbook are listed in the "**Procedure**" box (top right of the Module). Simply select the Event you want and it will be written for you.

Lets look at each one of these is turn and put them to use.

6.1.1 Workbook_Open()

This is fairly self-explanatory. The Event will fire whenever the Workbook that houses the code opens. It can be used to make any needed checks upon opening, or to inform the user of some information, or to make needed changes upon opening. In fact as with most Events, the only limit is ones own imagination. The **Workbook_Open()** is ALWAYS the **very first Event to fire** in both the Workbook Events and the Worksheet Events. The next one to fire will be the **Workbook_Activate()**. Some users will quite often get confused with these two Events. The **Workbook_Activate()** can fire any number of times while the Workbook is open, while the **Workbook_Open()** will only fire once. As an example let's say you have two Workbooks open at one time, **Book1.xls** and **Book2.xls** with **Book1.xls** housing some code in the **Workbook_Activate()**. You need to go to **Book2.xls** so you activate it by going to **Window>Book2.xls** or selecting it from the Task bar. You make the changes then Activate **Book1.xls** again, this time the code that is within the **Workbook_Activate()** will run (or Fire). I frequently use the **Workbook_Activate()** and **Workbook_Deactivate()** to show and hide a custom CommandBar that I have created for a particular Workbook. I would use the **Workbook_Open()** to create the CommandBar only. It is important to note that if the user decides NOT to enable macros **NO Events will fire**. You can also prevent Events from firing by opening the Workbook from within Excel while holding down the **Shift** key.

Ok, so lets now use the **Workbook_Open()** Event in a useful way. Imagine you have a Workbook that needs to remind the user when it is the first day of the Month so that he/she must rollover the needed data. You could do this by displaying a ”Message box” upon opening:

```
Private Sub Workbook_Open()
   If Day(Date) = 1 Then
      MsgBox "It is the " & Format(Date, "dd/mmmm/yyyy") _
      & " today, so please rollover the data.", vbInformation
   End If
End Sub
```

This is a nice and simple way to inform the user of needed changes.

6.1.2 Workbook_BeforeClose(Cancel As Boolean)

This Event is nearly the complete opposite of the **Workbook_Open()** Event. I say nearly because it has the word "**Before**" in it. This means the Event fires just before the Workbook is closed not once it is closed. You will also notice that this Event has the words ”**Cancel As Boolean**" in brackets. This is because this Event (as do many) takes an argument, in this case "Cancel" and returns a Boolean (**True** or **False**). Let's say you click the **X** to close a Workbook or go to **File**>**Close**, if you have made any changes Excel will ask you if you wish to save any changes you have made. On the dialog box it displays three choices "Yes", "No" or "Cancel" it is this "**Cancel**" which returns either True or False to the "BeforeClose" Event. The reason this is there is so that if the user DOES click "Cancel" you can opt whether to run your "BeforeClose" code or not.

Let's again use this in a practical way to see what I mean. Sticking with the "**Workbook_Open**" code example, let's say you can tell whether the data has been rolled over by seeing if Column A contains no data at all. If has not been rolled over you can decide the course of action. But if they "**Cancel**" at the save changes stage you don't want to take any action.

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)
Dim iReply As Integer

'If they have "Cancelled" at the Save _
Changes dialog then do no more.
If Cancel = True Then Exit Sub

    'See if it's the first day of the Month and _
     count All entries in column A.
    If Day(Date) = 1 And WorksheetFunction.CountA _
    (Sheets("sheet1").Columns(1)) = 0 Then

    'Inform them of the rollover and let them decide _
     whether to continue closing or not.
     iReply = MsgBox("You have not rolled over the data." _
     & "Do you still wish to close", vbYesNo)

    'If they choose "No" then stop the Workbook _
     from closing.
     If iReply = vbNo Then Cancel = True
End If
End Sub
```

So the very first thing that will happen when they close is that they will be asked if they want to save changes. If they choose "**Cancel**" then this will be passed back to our code as "**True**". Our code will then simply **Exit Sub** and go no further. ***Exit Sub:*** *Immediately exits the* ***Sub*** *procedure in which it appears.*

Next our code checks that it is the first day of the Month and also if Column A contains any entries at all. If it does then the message box is displayed asking if they still want to close, if they select **No** (vbNo) we use the "**Cancel**" argument to stop the Workbook from closing by setting it to True and then Exit the Sub. Of course if they select "**Yes**" (vbYes) the Workbook would close as normal.

- *The* ***Date*** *function will return the current system date.*
- *The* ***Day*** *function will return a whole number (****Integer****) between 1 and 31, inclusive, representing the day of the month*

6.1.3 Workbook_BeforeSave(ByVal SaveAsUi As Boolean, Cancel As Boolean)

As with the "BeforeClose" event this Event is fired immediately before the Event, which in this case is **Save**. It also takes two arguments *"***SaveAsUi***" and "***Cancel***"* both of which take a Boolean as their Value. The "**Cancel**" argument is nearly the same as the "Cancel" in the BeforeClose Event with one difference, and that is we can only set the argument to True or False we cannot have the argument passed to our code. This is simply because there is no built in function to stop a Save once it has started.

The "**SaveAsUi**" can also be set to True of False and will decide whether the "**SaveAs**" dialog will be displayed. So in a nutshell if we set "Cancel" to True the Workbook will NOT save. If we set "SaveAsUi" to False the "SaveAs" dialog will NOT be displayed.

For this example let's assume we want to stop users from saving a copy of our Workbook as a different name. We know that they can only do this from within the Excel interface by going to **File**>**Save As**. So to stop this from happening and at the same time telling them so, we could use this code:

```
Private Sub Workbook_BeforeSave _
(ByVal SaveAsUI As Boolean, Cancel As Boolean)
   If SaveAsUI <> False Then
      MsgBox "Please do not save another copy of this file", vbCritical
      Cancel = True
   End If
End Sub
```

This code will fire immediately before the user Saves, so if they just click Save (**not Save As**) the code will NOT enter the **If** statement because the "SaveAsUi" will return False (it's default). If however, they decide to go to **File**>**Save As**, a Boolean Value of **True** will be passed back to "SaveAsUi" and so our code WILL enter the If statement and show our message box. As soon as they "**OK**" the message box the "Cancel" argument will be set to True and stop the Workbook from saving.

6.1.4 Private Sub Workbook_NewSheet(ByVal Sh As Object)

This Event is fired whenever a new sheet is added to the Workbook. Some of its uses can be used to stop a user from adding any more sheets, pre-formatting a sheet for them, or automatically naming it. Again you will notice that it has "**ByVal Sh As Object**" this is really declaring the new sheet which is added as an **Object**. You may be wondering why it uses an Object and not a Worksheet for its declaration, this is because the Event also fires whenever a Chart sheet is added. ***A Worsheet Object cannot refer to a Chart Object***. To put this Event to use let's make it easy on our user by automatically naming the sheet the same name as the current Month. As you are no doubt aware Excel will not allow you to have two sheets of the same name so we will need to place in some code to ensure they do not already have a sheet for the current Month.

```
Private Sub Workbook_NewSheet(ByVal Sh As Object)
'Stop run time error if sheet exists
 On Error Resume Next

'Name the new sheet the current month
 Sh.Name = Format(Date, "mmmm")

   'If a sheet with the same name does exist then _
```

```
    the sheet will be called sheetx. _
    This means we should delete the sheet.
    If Sh.Name <> Format(Date, "mmmm") Then
       'Prevent the Excels standard sheet deletion warning.
        Application.DisplayAlerts = False
        Sh.Delete 'Delete the sheet
       'Turn warnings back on
        Application.DisplayAlerts = True
       'Tell them what has happened.
        MsgBox "You already have a sheet for " & Format(Date, "mmmm")
    End If
End Sub
```

This code will work exactly as I stated above. You will notice that I have used "**On Error Resume Next**", "**Application.DisplayAlerts=False**" and "**Application.DisplayAlerts=True**" The "**On Error Resume Next**" will prevent the Run time error which **would occur** if there is already a sheet of the same name, in fact it will stop any Run time errors for the remainder of the code. Then "**Application.DisplayAlerts=False**" is used to stop Excel default warning messages such as the one you get when you delete a Sheet. Setting it back to True is just putting it back to its default.

The four above Workbook Events can be used in many different ways for many different reasons. They are ideal for automating procedures that you want to run when a specific Event occurs.

6.2 Worksheet Events

Let's now look at some of the **Worksheet** Events. These are not a lot different from the Workbook Events except that while the Workbook Events can apply to all Worksheets the Worksheets Events **only apply the the Worksheet which houses them**.

The most popular Events for the Excel Worksheet are:

- Private Sub Worksheet_Change(ByVal Target As Excel.Range) Occurs when cells on the worksheet are changed by the user or by an external link
- Private Sub Worksheet_Calculate() Occurs after the worksheet is recalculated
- Private Sub Worksheet_BeforeRightClick(ByVal Target As Excel.Range, Cancel As Boolean) Occurs when an embedded chart or worksheet is right-clicked, before the default right-click action
- Private Sub Worksheet_Deactivate() Occurs when a Worksheet in deactivated.

As with the Workbook Events all these start with the words "Private Sub". Again this means they are **only** available to the Worksheet Object in which they are housed. These Worksheet Objects can be seen in the "Project Explorer" and have the same name as the Worksheet tab name. You can double click the Worksheet Object to gain access to the Worksheet Module. If you are not in the VBE you can right click on the sheet name tab and select "***View Code***". In Excel 97 the default will be:

Private Sub Worksheet_SelectionChange(ByVal Target As Excel.Range)

End Sub

If you are using Excel 2000 + the default will be blank. You can still save yourself the problem of having to type out the Worksheet Event you want by clicking the drop arrow on the "**Object**" box (top left of the Module) and selecting "**Worksheet**". This will then default to:

Private Sub Worksheet_SelectionChange(ByVal Target As Excel.Range)

End Sub

In the both versions 97 and 2000 + all the Events for the Worksheet are listed in the "**Procedure**" box (top right of the Module). Simply select the Event you want and it will be written for you.

Lets look at each one of these is turn and put them to use.

As the **Workbook_Open** is arguably the most popular Workbook Event the **Worksheet_Change** is the most popular Worksheet Event.

Private Sub Worksheet_Change(ByVal Target As Excel.Range)

This event is fired whenever a cell on a Worksheet changes. The exception to this is deleting a cell, the Event does not fire then. You can see that this Event takes an argument that is called "**Target**" and is parsed to the Event as a range. The Target **is always the cell that triggered the Event**. Let's say that every time a user types a number in cells **A1:A10** we need to multiply it by itself.

```
Private Sub Worksheet_Change(ByVal Target As Excel.Range)
Dim rWatchRange As Range
On Error GoTo ResetEvents

'Set range variable to A1:A10
 Set rWatchRange = Range("A1:A10")

    'The cell they have changed (Target) _
     is within A1:A10
    If Not Application.Intersect _
            (Target, rWatchRange) Is Nothing Then
            'They entered a number
             If IsNumeric(Target.Value) Then
                'Prevent the Event from firing again _
                 when we multiply it(the Target)by itself.
                 Application.EnableEvents = False
```

```
            'Multiply it by itself
             Target.Value = Target.Value * Target.Value
        End If
    End If

ResetEvents:
Application.EnableEvents = True
End Sub
```

Let's now work our way through the code to see what it does.

On Error GoTo ResetEvents

This is put in to force Excel to go to the line of code directly below "ResetEvents" should any unexpected errors occur.

Set rWatchRange = Range("A1:A10")

This sets our range variable to the Range we are interested in.

If Not Application.Intersect(Target, rWatchRange) Is Nothing Then

The Intersect returns a Range object that represents the rectangular intersection of two or more ranges. We use this to find out whether the cell which triggered the Event (**Target**) is within **A1:A10** (rWatchRange). If the Target is NOT within the range A1:A10 the Intersect Method would return **Nothing**. *The default for a empty Range Object*. The "**Not**" is used to reverse the If statement.

If IsNumeric(Target.Value) Then

This will return True if the Target contains a number.

Application.EnableEvents = False

This is very important when using Worksheet Events. If we did not include it we could end up with an endless Loop. This is because we are changing the Target cell which would in turn re-trigger the change event, which would then run our code again and so on…..

Target.Value = Target.Value * Target.Value

Once we finally reach here we can multiply the value of the Target by itself.

ResetEvents:
Application.EnableEvents = True

The "**Application.EnableEvents=True**" turns the Events back on so it will run the next time. The "**ResetEvents:**" is a sign post for our "**On Error GoTo**". If an unexpected error does occur after Events have been disabled it will at least enable them again.

Private Sub Worksheet_Calculate()

This Event is fairly straightforward, it fires immediately after the Worksheet calculates. If we had a Worksheet full of formulas that were recalculating frequently we could use this to check whether the recalculated value is a certain cell has reached a certain limit. If it has, we could display a message box telling them to perform some action.

```
Private Sub Worksheet_Calculate()
   If IsNumeric(Range("A1")) Then
     If Range("A1").Value >= 100 Then
       MsgBox "Range A1 has reached is limit of 100", vbInformation
     End If
   End If

End Sub
```

As you can see the Worksheet_Calculate Event takes no arguments.

Private Sub Worksheet_BeforeRightClick (ByVal Target As Excel.Range, Cancel As Boolean)

This Event fires immediately before the default action of the Worksheet right mouse click. In the case of the Worksheet that default action is the shortcut popup menu that appears. This Event takes two arguments "**Target**" and "**Cancel**". The Target is the same as the Target for the Worksheet ie; it refers the active Range Object. The Cancel refers to the shortcut popup menu, setting it to **False** will prevent the popup menu from showing. I have used this Event in the past to simply prevent the popup menu from showing when I have hidden all Excels standard Command Bars and replaced them with a customized one. I have also used it to display my own custom popup menu. As this is probably the best use for this Event, let's use it as an example.

```
Private Sub Worksheet_BeforeRightClick _
(ByVal Target As Excel.Range, Cancel As Boolean)
Dim cBar As CommandBar
Dim cCon1 As CommandBarControl
Dim cCon2 As CommandBarControl

'Prevent the standard popup showing
 Cancel = True

'We must delete the old one first
```

```
On Error Resume Next
Application.CommandBars("ConnectCode").Delete

    'Add a CommandBar and set the CommandBar _
     variable to a new popup
     Set cBar = Application.CommandBars.Add _
     (Name:="ConnectCode", Position:=msoBarPopup)

    'Add a control and set our 1st CommandBarControl _
     variable to a new control
     Set cCon1 = cBar.Controls.Add
    'Add some text and assign the Control
    With cCon1
        .Caption = "I'm a custom control"
        .OnAction = "MyMacro"
    End With

    'Add a control and set our 2nd CommandBarControl _
     variable to a new control
     Set cCon2 = cBar.Controls.Add
    'Add some text and assign the Control
    With cCon2
        .Caption = "So am I"
        .Caption = "AnotherMacro"
    End With

cBar.ShowPopup
End Sub
```

Don't worry too much if you cannot fully understand how the adding Popup works, I only use this as an example because I believe it is the most relevant example. Let's step through it.

Cancel = True

As I said above this setting the Cancel argument to True will stop the default popup.

On Error Resume Next
Application.CommandBars("ConnectCode").Delete

This part is very important! While the code would run ok the very first time we ran it, it would not run the next time or anytime after. This is because a CommandBar called "**ConnectCode**" would already exist and so cause a run time error. The "**On Error Resume Next**" will prevent a run time error the first time it tries to delete the Command Bar "ConnectCode", which wouldn't yet exist.

Set cBar = Application.CommandBars.Add _
(Name:="ConnectCode", Position:=msoBarPopup)

Here we are adding a Command Bar calling it "ConnectCode" and setting it's Position to "**msoBarPopup**". You must do this with a Popup. If it was a normal Command Bar we could Position it at the top of the screen with all other Command Bars. We then set our variable to the new Command Bar.

Set cCon1 = cBar.Controls.Add

We now add a Control to our new Command Bar and set it to a variable.

```
With cCon1
    .Caption = "I'm a custom control"
    .OnAction = "MyMacro"
End With
```

Now we use the Control Object (cCon) and add a Caption, then assign it to a macro called "Mymacro", this would reside in a normal Macro.

cBar.ShowPopup

This simply tells Excel to show our new Popup.

Private Sub Worksheet_Deactivate()

This Event occurs whenever the Worksheet is Deactivated. You could use this on worksheet to ensure that the sheet always remained hidden. The user would have to go to **Window**>**Unhide** to view the sheet.

```
Private Sub Worksheet_Deactivate()
   Me.Visible = xlSheetHidden
End Sub
```

Note the use of the keyword "**Me**" here. As the code is in the Private Module of the Worksheet Object we can refer to the Worksheet with this. The same could be used in all Worksheet Events. If we were dealing with a Workbook Object Event "**Me**" would always refer to the Workbook itself.

6.3 Summary

So as you can see we can have any macro or code run whenever a particular Event occurs. This can be a big advantage to the programmer as we do not have to rely on the user to activate it. It can also make life easy for the user as it can do things for them automatically, even without them knowing.

7. DEBUGGING

Please cross reference the Debugging and Error Handling.xls Workbook for this lesson.

This lesson we will concentrate on **De-bugging** code and **Error Handling**. You will no doubt find that debugging is something you will be doing a lot in the early stages of learning. A good programmer will never think that their finished project doesn't contain any errors, if you do, you won't bother placing in any code to handle errors and that is a BIG mistake. There are many features in Excel that can make **De-bugging** code a reasonably easy task. To avoid confusion and overload we will discuss in detail what I believe to be the best method and by far the first one is the best:

7.1 Prevention is better than cure

Now we have all heard the saying that "**prevention is better than cure**" and this really holds true with **VBA**. I believe that a lot of code errors can be prevented just by forming good habits early. One very simple habit is, when typing code in the **VBE** always use all lower case. This is so any **Keywords** that Excel recognizes will automatically have their first letter capitalised. This also holds true for typing **Variable** names or **Control** names, in fact it holds true for a lot of cases, so develop this habit early. Not only are bad habits hard to break, but so are good ones!

Another saying that holds true in **VBA** for Excel is "**Never assume anything**". So whenever writing your code try to think what could cause this to fail. Some of the most common causes are **Worksheet** names changing, **Worksheets** position changing, **Worksheet** protection turned on, encountering **Text** when you expect a **Number** or vice versa and NOT **naming ranges**. Accounting for these potential problems can mean more typing of code, but believe me the short term pain will far out-weighed by the long term gain. Let's look at each of these in turn and see how we can eliminate them.

7.2 Worksheet Names

By far the best way to overcome this is to use the sheet objects **"CodeName**". This can be seen in the **Properties Window** of the sheet or in the **Project Explorer** (it's the name NOT in the brackets). The **CodeName** does not change if the sheet **Tab** name changes or if the sheet is moved to a different position in the **Workbook**. As an added bonus it is also very helpful when coding. If you type: **Sheet1** and then place the period (dot) after it, Excel will list all it's **Properties** and any **Objects** on that sheet.

It is not possible to change a Sheets **CodeName** via VBA, it must be done in the **Properties Window** for the Sheet Object. To do this ensure the **Project Explorer** is visible (**Ctrl**+**R**) then display the **Properties Window** (**F4**). Select the Sheet Object in the **Project Explorer** then you will see it's CodeName in the **Properties Window** next to "**(Name)**". However, there should not really be any need to change this!

7.3 Worksheet Protection

It is vital when you develop a project that you **Protect** as much as possible, in particular the **Worksheet Cells**. But of course this means that if you try to perform a change to the sheet without **UnProtecting,** your code will fail. So you can start your procedure with the line:

Sheet1.Unprotect Password:="secret"

and end with..

Sheet1.Protect Password:="secret"

This will work just fine. Or you can use a more generic approach and create **2 Procedures** within a normal **Module** like:

```
Option Explicit
Dim wWsht As Worksheet
Sub UnProtectAllSheets()
    For Each wWsht In ThisWorkbook.Worksheets
        wWsht.Unprotect Password:="secret"
    Next wWsht
End Sub

Sub ProtectAllSheets()
    For Each wWsht In ThisWorkbook.Worksheets
        wWsht.Protect Password:="secret"
    Next wWsht
End Sub
```

Then you can start your **Procedures** with the line:

Application.Run "UnProtectAllSheets"

and end with

Application.Run "ProtectAllSheets"

Just be aware with both methods, that if the code fails for or any reason and does not reach theProtect statement, your Worksheets will be Unprotected. I will show you how we can overcome this later on.

There is another method that can be used to protect your Worksheets from being changed by a user, but still allow any changes via VBA. This is done by setting the Protects last argument (UserInterfaceOnly) to True. The syntax for Sheet protection is:

<SheetObject>.Protect(Password, DrawingObjects, Contents, Scenarios, UserInterfaceOnly)

However, if you do opt for this method it maybe at your own peril! I have used this method in the past and then used the FillAcrossSheets Method which left the Worksheets fully unprotected! My advise is to play it safe and give it a miss.

7.4 Encountering Text when you expect a Number or vice versa

One of the downsides with declaring all **Variables** is they will be expecting what you have dimensioned them for. So the code:

```
iResult=Sheet1.Range("A1").Value
```

will cause a "**Type mismatch**" error If **A1** contains **Text** and not an **Integer**. So unless we can be certain range **A1** contains a **Number** we should first check like this:

```
If IsNumeric(Sheet1.Range("A1").Value) Then
  IResult=Sheet1.Range("A1").Value
End If
```

The other side of the coin would be encountering a **Number** when we expect **Text**. Fortunately this won't cause an error, as Excel will simply make the **Number** a **Text number**, i.e; `SResult=Sheet1.Range("A1").Value` when range A1 contains the **Number** 100 will return "100" as text. But try and do any calculations with it and we will receive an error. So if you will need to use it in calculations use one of the **Conversion Functions**.

```
CInt(SResult)
```

There are many of these in Excel and are well documented. Push **F1** and type: **Conversion Functions**

7.5 Not Naming Ranges

This is very important in **VBA** coding as the code will NOT automatically follow the address of a cell should it be moved. If you place a **formula** in cell **A1** that refers to cell **B1** and then **Cut** and **Paste** cell **B1** to cell **D1**, the **formula** will automatically change it's reference to cell **D1**. Not so in **VBA**! So for this reason it is very important to **name ranges** and then our code will references the correct range.

You may be thinking "What has any of the above got to do with **De-bugging** ?" The answer is simply, it will prevent a lot of unnecessary **De-bugging**. Now it's almost inevitable that you will have to do some **De-bugging** while learning, but follow the above rules and it wont be very hard.

7.6 Compile Regularly

Perhaps one of the easiest and most overlooked method is the use of "**Compile VBA project**" option found under "**Debug**" on the main menu within the **VBE**. The beauty of this feature is you can pick up a lot of (but not all) errors without even running your code. If you can imagine you have written some code that goes though and makes hundreds of changes to a spreadsheet. You decide to **Run** your code to see if all is OK and on the second last line you code causes a **Run-time error**.

- ***Run-time error***: *An error that occurs when code is running. A run-time error results when a statement attempts an invalid operation.*

You now have to **MANUALLY undo** (**undo changes** will not work) all your changes so you can test your code again. You may be tempted to just close the **Workbook** without saving, but be very careful you are not going to loose changes you **DO want saved.**

Each time you finish (or what you think is finished) writing **VBA** code within a **Procedure** or **Sub** click the "**Compile VBA project**". Notice it mentions the word "**project**" not **Sub** or **Procedure**. Doing this will immediately pick up any errors in our code and take you straight there and display a message box telling you the problem. Now while this eliminates any errors directly related to your code, it wont pick up any errors directly related to **Cells**, **Worksheets** etc. What I mean by this is you could have the line:

```
Sheets("ThisYear").Cells(1,1).Value=2001
```

and even if there was no sheet in you **Workbook** called "**ThisYear**" Compiling will not pick it up. Try making the same mistake using the **CodeName** and you will know without even having to **Compile**.

7.7 The Dreaded Run time error

The very last one is the handling and fixing of **Run time errors**. As I said above, it's almost inevitable that you will have to do some **De-bugging**, but the job should not be a problem if you have followed all the above rules. One other thing which I have not mentioned above is the use of **Comment** text within code. Be liberal when commenting code as you will be thankful you did when it comes time to **De-bug**. One important thing to keep in mind though while writing your comments is, DON'T write too much on **what** the code does but rather **why** it does it. You should be able to see the "**what**" from reading your code.

7.8 The Local Window

This is one of my preferred methods of **De-bugging** and it is very rare that I need to go any further into **De-bugging** should I even get this far. The **Locals Window** can be found under "**View**" on the menu bar. The reason I prefer this is that, it will automatically displays all **Declared Variables** in the current **Procedure** along with their values. But you must be in **Break mode**:

- ***Break mode***: *Temporary suspension of program execution in the development environment. In break mode, you can examine, debug, reset, step through, or continue program execution. You enter break mode when you:*

1. *Encounter a breakpoint during program execution.*
2. *Press CTRL+BREAK during program execution.*
3. *Encounter a Stop statement or untapped run-time error during program execution.*
4. *Add a Break When True watch expression. Execution stops when the value of the watch changes and evaluates to True.*
5. *Add a Break When Changed watch expression. Execution stops when the value of the watch changes.*

The easiest way to do this is to insert a **Breakpoint** at the start of your code. You can do this by a single left mouse click on the **Margin indicator bar** (left of the code window) or by pushing **F9** . Excel will place a large dot on it. Now when you run your code it will stop at the **Breakpoint** and all variables in the procedure will be displayed for you in the **Local Window**. You can now **Step** through your code 1 line at the time by pushing **F8**. You can also toggle between the **VBE** and the Excel interface while your code is in **Breakmode** by pushing **Alt+F11** or selecting your workbook from the taskbar. Now between these 2 steps you should be able to track down exactly where the problem lies. If you do find a problem you can change the code while in **Breakmode,** you can change it there and then if you wish and see immediately what effect the change has made. You can also see the value of all your **Variables** while in **Breakmode** by waving your mouse pointer over any highlighted code.

I have followed the above methods now for many years and I can honestly say that I rarely need to get as far as using any one of Excels **De-bug** windows.

7.9 Error Handling

Now we have covered the basics of **De-bugging** code let's move on to using some sort of **Error handling** within our code. An "**Error handler**" is pretty much as the name suggests, it handles any errors the may arise. Now **Error handling** in Excel can become very complicated and is a very large area, but in the interest of keeping things simple we will look at what I believe to be the least you need to know. When you have written your code you will, as described above, try to ensure that no problems will arise. But unfortunately it is virtually impossible to be 100% sure nothing will go wrong. So for this reason we must take a pessimistic point of view and place in some code to allow for errors. The last thing the end user wants to see is a **Run time error,** it will probably mean nothing to the user and only make him/her lose faith in your ability and the programs. When you have written a Procedure and are satisfied it will do as it is supposed to, it's time to get very negative and look through it with a pessimistic point of view. This means we must expect the worst!

Before we move into how we can account for these potential errors I will go through what I believe to be the most effective preventers of errors.

On Error GoTo <line>
Where "**`<line>`**" is any line label in the same procedure as the
"**On Error GoTo**" Statement.

a line label is described as:

- ***line label***:*Used to identify a single line of code. A line label can be any combination of characters that starts with a letter and ends with a colon (**:**). Line labels are not case sensitive and must begin in the first column.*

Do not fall into the trap of using this method a lot as it can make code very hard to follow and **De-bug. You end up with what is know as "**spaghetti code**".**

On Error Resume Next This is one of the most popular methods of handling any potential errors. It simply tells Excel to keep going whenever a **Run time error** is encountered and go to the next line of code. Be very careful when using this one as it can hide any **Run time errors** that you should know about. I generally only use this to allow some code to check another line of code. As shown in the next example

On Error GoTo 0 This will turn off the "**On Error Resume Next**" and allow any **Run time errors**.

To put to use the "**On Error Resume Next**" and the "**On Error GoTo 0**" you could use it in this context:

```
Sub IsWorkbookOpen()
Dim wBook As Workbook

On Error Resume Next
   Set wBook = Workbooks("Book2")
      If wBook Is Nothing Then
         MsgBox "Book2 is not open!", vbCritical
      Else
         MsgBox "Book2 is open!", vbInformation
      End If
On Error GoTo 0
End Sub
```

This will allow our code to continue to our If Statement where we can now check to find out whether a Workbook called "**Book2**" is open on the users PC. Remembering that the default for an Object variable is **Nothing**. Without the "**On Error Resume Next**" we would encounter a "**Subscript out of range**" error and our code would be stopped dead. Try to only use this in this context and you shouldn't have any problems.

Now, I mentioned earlier that we could use some code like:

```
Application.Run "ProtectAllSheets"
```

Which would re-protect all our Worksheets within the Workbook once our code had run. But I as I mentioned, if for any reason our code encounters a **Run time error** that we have not thought of, our entire Workbook would have all it's Worksheets unprotected. This is not something we want! So this is where we could use the "**On Error GoTo** <line>". So we would have the two procedures below in a normal Module:

```
Dim wWsht As Worksheet
Sub UnProtectAllSheets()
   For Each wWsht In ThisWorkbook.Worksheets
      wWsht.Unprotect Password:="secret"
   Next wWsht
End Sub
```

```
Sub ProtectAllSheets()
   For Each wWsht In ThisWorkbook.Worksheets
      wWsht.Protect Password:="secret"
   Next wWsht
End Sub
```

So our first line of code in the **Procedure that called the two above procedures**, would be:

Application.Run "UnProtectAllSheets"

and

Application.Run "ProtectAllSheets" as one of our last line

So we could now use some code like this:

```
Sub DoToAllSheets()
Dim wWsht As Worksheet

On Error GoTo ReProtect
   Application.Run "UnProtectAllSheets"
      For Each wWsht In ActiveWorkbook.Worksheets
         Range("A1") = 10
      Next wWsht
ReProtect:
Application.Run "ProtectAllSheets"
End Sub
```

So if for any reason our code encounters a **Run time error**, all Worksheets would still be protected.

7.10 Exit Sub

There is one other means of trapping errors and that is to have your code do what is know as **`Exit`** your procedure cleanly and throw up a message box. I use this method only as my last "port of call". The **`Exit Sub`** used in combination with the "**`On Error GoTo <line>`**" can allow us to place within Procedures we choose a generic type of Error Handler. I have included in this lesson a working example of this along with some other related examples.

8. WORKSHEET FUNCTIONS

This lesson we will focus this lesson on **Worksheet Functions** in VBA for Excel. Excel has a number of functions that can be used in the Visual basic environment only, but it also has at it's disposal a long list of the standard **Worksheet Functions**. When these are combined with VBA for Excel it adds even more power and flexibility to the Visual Basic environment. Whenever you write a formula in Excel we **must** precede it with an = (equal sign). When we use the same formula or function in Excel VBA we **must** precede it with the words "**WorkheetFunction**". The **WorkheetFunction** is a Object member of the **Application** Object and as such we do not need to use the full: **Application.WorksheetFunction**. You can see the full list of **Worksheet Functions** available to the us within VBE by looking under "**List of Worksheet Functions Available to Visual Basic**". It is a great idea to use **Worksheet Functions** within your code as it can often means doing away with unnecessary filling up of ranges with formulas. This in turn will mean a faster Workbook.

We will also look at making use of Excels **built-in features** and how to use them within VBA. I am of the very strong opinion that this is a **MUST** when using VBA for Excel, as combining the **built-in features** with VBA will result in being able to overcome almost any problems you are faced with. It will also keep us away from those **Loops** and use them only when **really** needed. As with **Worksheet Functions** we can use almost all of Excels **built-in features** and the only limit to their use is your own imagination. In particular, we will look at the **Find**, **Autofilter**, **AdvancedFilter** and **SpecialCells** (Go to Special in the Excel interface). There is no doubt at all in my mind that these built-in features will do for us in less than a second what could take 100 times as long by writing our own VBA procedure to do the same thing. Although this is patently clear to me I am always amazed at the amount of experienced VBA coders that insist on re-inventing the wheel.

We will look first at using **Worksheet Functions** in VBA.

8.1 Making life a bit easier

Before we look at some specific examples I should point out the guidance Excel will provide when you use any of the **Worksheet Functions**. As soon as you type: **WorksheetFunction** and then place the "**.**" (period) after it, Excel will list all the Worksheet functions that are available in alphabetical order. Once you have picked the function you need you will type: **(**(an open parenthesis) and Excel will then show the expected syntax for that particular function. Sometimes this is enough to jolt the memory if the function is one that you have used before. At other times it won't be of much help at all. There are two ways I use to overcome this. For both examples I will use the **VLOOKUP** function

8.1.1 Method 1

1. Switch back to the Excel interface (Alt+F11) then in any cell type: **=vlookup**
2. Then push Ctrl+Shift+A. This will give you the syntax for **VLOOKUP**.
3. Copy this from the Formula bar and switch back to the VBE (Alt+F11).
4. Paste this at the top of the Procedure as a comment.

8.1.2 Method 2

1. Switch back the Excel interface (Alt+F11) then in any cell type the formula as you would normally.
2. Copy this from the Formula bar and switch back to the VBE (Alt+F11).
3. Paste this at the top of the Procedure as a comment.

Then all you need to do is look at the comment and use this as a guide.

8.2 Specific examples

Let's look at some specific examples. Assume you have a range of cells on a Worksheet and you want to find out the sum total of one particular column, to do this we would use:

```
Sub SumColumn()

Dim dResult As Double

   dResult =WorksheetFunction.Sum (Columns(1))
  MsgBox dResult
End Sub
```

This makes use of the **SUM** function that is available to the Excel interface. As you are no doubt aware, the **SUM** function used in the Excel interface normally uses a range address as it's argument. When using **Worksheet Functions** with the Visual Basic environment we replace this with a Range Object. "**Columns(1)**" is a Range Object.

You will also notice that the **SUM** function can take up to **30** arguments, so we could supply up to another **29** Range Objects if needed. Which means we could also use:

```
Sub SumMoreThanOneColumn()
Dim dResult As Double

   dResult = WorksheetFunction.Sum(Columns(1), Columns(3), Columns(5))
   MsgBox dResult
End Sub
```

Notice how in both examples I have parsed the result of the **SUM** function to a variable that has been dimensioned as a **Double**. This is done so we do not end up with a whole number only. For example, if the answer to the **SUM** function was **1001.265** and we parsed this to an **Integer** or **Long** we would end up with an answer of **1001**. Of course if this is close enough for our result we could use a **Integer** or **Long**.

Both of the above example will only **SUM** the values in the Range Object(s) stated of the ActiveSheet. We could do the same for three different sheets if needed.

```
Sub SumMoreThanOneColumnSeperateSheets()
Dim dResult As Double

   dResult = WorksheetFunction.Sum _
      (Sheet1.Columns(1), Sheet2.Columns(3), Sheet3.Columns(5))
   MsgBox dResult
End Sub
```

Ok, so we know that we need to use a Range Object in any **Worksheet Function** that would normally use a range address. But a lot of **Worksheet Functions** will also accept values as their arguments. So if we had a number of numeric variables that we needed to know the sum total of, we could use:

```
Sub SumVariables()
Dim dResult As Double
Dim dNum As Double
Dim iNum As Integer
Dim lNum As Long

   iNum = 1000
   lNum = 70000
   dNum = 25.65
      dResult = WorksheetFunction.Sum(iNum, lNum, dNum)
      MsgBox dResult
End Sub
```

In addition to using values or a Range Object we could also use the **Selection** Method. This of course would be a **Range Object** in this context providing the **Selection** was a range of cells and not some other type of Object. So the Procedure:

```
Sub SumSelection()
Dim dResult As Double

   dResult = WorksheetFunction.Sum(Selection)
   MsgBox dResult
End Sub
```

Is a perfectly valid Procedure.

There will of course be times when you will need to use one of the other **Worksheet Functions** so let's go through a couple more of them.

8.3 COUNTIF

```
Sub UseCountIf()
Dim iResult As Integer

   iResult = WorksheetFunction.CountIf(Range("A1:A10"), ">20")
   MsgBox iResult
End Sub
```

8.4 VLOOKUP

```
Sub UseVlookUp()
Dim sResult As String

   sResult = WorksheetFunction.VLookup("Dog", Sheet2.Range("A1:H500"), 3, False)
   MsgBox sResult
End Sub
```

Be aware that if the text "Dog" does not exist in the first Column of Sheet2.Range("A1:H500") a Run-time error will be generated. This can be handled in a number of ways, but possibly the best would be to use a small **If** Statement.

```
Sub UseVlookUp()
Dim sResult As String

   If WorksheetFunction.CountIf _
     (Sheet2.Range("A1:H500"), "Dog") <> 0 Then
        sResult = WorksheetFunction.VLookup _
                  ("Dog", Sheet2.Range("A1:H500"), 3, False)
        MsgBox sResult
   Else
      MsgBox "No dog can be found"
End If
End Sub
```

There is really not much else that can be said about the use of **Worksheet Formulas** within the Visual Basic environment. I do though **highly** recommend using them in many situations as the speed at which they will supply an answer will always be far quicker than any VBA code written to do the same.

8.5 Excels Built-in Features

This would no doubt be one of Excel most under-utilised areas. It seems that when Excel users learn VBA they tend to forget the **built-in features** that are available to us in the Excel interface are still available in the Visual Basic environment. The examples I will show you can be applied to a lot of different situations and it will nearly always be worth sitting back and trying to think where they can be used. The easiest way to get the **foundations** for your code that makes use of one of

Excels **built-in features** is without doubt by using the Macro Recorder, but it should only be a **starting point** not the code itself.

8.5.1 Find

When using the **Find** in VBA it is a good idea to parse the result to a properly declared variable. The reason for this is so that we can check the result to see if the value we are looking for has been found. To do this we **must** also use the: On Error Resume Next Statement so that our Procedure does not produce a Run-time error. The examples below are just some of the ways I have used this feature in my VBA code of past. For all examples I will use the ActiveSheet, but the examples can (and often are) used on another sheet.

```
Sub FindObject()
Dim rFound As Range

On Error Resume Next
   Set rFound = Cells.Find(What:="100", After:=Range("A1"), _
                    LookIn:=xlValues, LookAt:=xlWhole, _
                    SearchOrder:=xlByRows, SearchDirection:=xlNext, _
                    MatchCase:=False)

         If rFound Is Nothing Then
           MsgBox "Cannot find 100"
        Else
          MsgBox rFound.Address
       End If
On Error GoTo 0
End Sub
```

```
Sub FindRow()
Dim lFound As Long .

On Error Resume Next
   lFound = Cells.Find(What:="100", After:=Range("A1"), _
                     LookIn:=xlValues,LookAt:=xlWhole,  _
                     SearchOrder:=xlByRows, SearchDirection:=xlNext, _
                    MatchCase:=False).Row

      If lFound = 0 Then
          MsgBox "Cannot find 100"
      Else
         MsgBox "100 is on row " & lFound
      End If
On Error GoTo 0
End Sub
```

```
Sub LookUpLeft()
Dim sFound As String

On Error Resume Next
      sFound = Columns(3).Find(What:="dog", After:=Range("C1"), _
                    LookIn:=xlValues,LookAt:=xlWhole, _
```

```
                    SearchOrder:=xlByRows, SearchDirection:=xlNext, _
                    MatchCase:=False).Offset(0, -1)

        If sFound <> "" Then
           MsgBox "One column to the left of dog, on the same row is " & sFound
        Else
           MsgBox "No value found"
        End If
On Error GoTo 0
End Sub
```

```
Sub BoldAllDogs()

Dim iCount As Integer
Dim rFound As Range

On Error Resume Next
Set rFound = Range("A1")


   For iCount = 1 To WorksheetFunction.CountIf(Cells, "dog")
      Set rFound = Cells.Find(What:="dog", After:=rFound,  _
                   LookIn:=xlValues,LookAt:=xlWhole, _
                   SearchOrder:=xlByRows, SearchDirection:=xlNext, _
                   MatchCase:=False)
                  rFound.Font.Bold = True
   Next iCount
On Error GoTo 0
End Sub
```

Do the procedure directly above (**BoldAllDogs**) using a standard Loop and you will have time to make and drink a coffee! Yet I see people doing similar time and time again.

I will also suggest strongly that you read the help file on the **Find** Method as it describes each argument in good detail. If you do not read it you will at least need to know this:

Remarks

The settings for **LookIn**, **LookAt**, **SearchOrder**, and **MatchByte** are saved each time you use this method. If you don't specify values for these arguments the next time you call the method, the saved values are used. Setting these arguments changes the settings in the **Find** dialog box, and changing the settings in the **Find** dialog box changes the saved values that are used if you omit the arguments. **To avoid problems, set these arguments explicitly each time you use this method.**

8.5.2 AutoFilter and SpecialCells

I will use the **Autofilter** with the **SpecialCells** Method in some of the examples below. This is because when using the **AutoFilter** you will most likely only be interested with the Visible cells. Again it may be needed that we use the **On Error Resume Next** Statement so we don't generate a Run-time error.

We can toggle between having Excels **Autofilter** on/off by using some code like:
Range("A1:H1").AutoFilter

This would turn **on** the AutoFilter if they were **off**.
It would turn them **off** if they were **on**.

8.5.3 AutoFilterMode

We can check to see if the AutoFilters are on by using the **AutoFilterMode** Property. It will return **True** if they are **on** and **False** if they are **off**.

```
Sub AreFiltersOn()
    If ActiveSheet.AutoFilterMode = False Then
        MsgBox "Filters are off"
    Else
        MsgBox "Filters are on"
    End If
End Sub
```

We can also use the **AutoFilterMode** Property to turn **off** the **AutoFilters** by setting it to **False**. But we cannot turn them on by setting it to **True**.

```
Sub TurnFiltersOff()

    If ActiveSheet.AutoFilterMode = True Then
        ActiveSheet.AutoFilterMode = False
    End If
End Sub
```

8.5.4 FilterMode

We can also check to see whether a Worksheet is in **FilterMode**, this will return **True** if the Worksheet currently has hidden rows as result of the **AutoFilter**. This is not the same as the **AutoFilterMode** Property, as **FilterMode** would return **False** if the sheet had **AutoFilters** turned on but not in use (not filtered down), while the **AutoFilterMode** would return **True**. This Property is **Read Only** so we cannot change it to **False** if **True** or **True** if **False**.

```
Sub IsSheetFiltered()
If Sheet1.FilterMode = True Then
    MsgBox "Yes it is"
     Else
      MsgBox "No it's not"
End If
End Sub
```

The other Method that we can use is the **ShowAllData**. This will set the AutoFilters back to "**(All)**" if they are currently in use (**FilterMode = True**). Be careful though as a Run-time error is generated if we use this method on a sheet that is not currently filtered (**FilterMode = False**). To avoid this we use the **FilterMode** Property to check first.

```
Sub IsSheetFiltered()
 If Sheet1.FilterMode = True Then
    Sheet1.ShowAllData
 End If
End Sub
```

Ok, so that is all the means and ways to find out the current status of a sheet with regards to **AutoFilters**. What we can do now is move on to actually using the **AutoFilter** Method in some VBA code.

Let's say we wanted to copy all the rows of a sheet that have the word "**dog**" in Column **C** and place them on another Sheet. This is where the **AutoFilter** Property can help us.

```
Sub ConditionalCopy()
With ActiveSheet
   If WorksheetFunction.CountIf(.Columns(3), "dog") <> 0 Then
      .AutoFilterMode = False
      .Range("A1:H1").AutoFilter
      .Range("A1:H1").AutoFilter Field:=3, Criteria1:="Dog"
      .UsedRange.SpecialCells(xlCellTypeVisible).Copy _
       Destination:=Sheet2.Range("A1")
      .AutoFilterMode = False
      Application.CutCopyMode = False
   End If
End With
End Sub
```

Let's step through this and see what we did:

1. `If WorksheetFunction.CountIf(.Columns(3), "dog") <> 0 Then` So the first thing we do is check to see whether the word "dog" is in Column C.
2. `.AutoFilterMode = False` We then set the .**AutoFilterMode** Property to **False**. We do not need to check if the **AutoFilters** are on because if they aren't nothing will happen. If they are on and/or in use they will be turned off.
3. `.Range("A1:H1").AutoFilter` We then apply the **AutoFilters** to the range we are interested in.
4. `.Range("A1:H1").AutoFilter Field:=3, Criteria1:="Dog"` We then set the **Criteria** Property to "**Dog**".
5. `Field3` is Column **C** and is relavent to the range we filtered. `.UsedRange.SpecialCells(xlCellTypeVisible).Copy Destination:=Sheet2.Range("A1")` Here we have used the **UsedRange** Property and the **SpecialCells** Method to copy **only**

the cells that are **visible** after applying the **AutoFilter**. We then paste them to **Sheet2.Range("A1)**.

6. .AutoFilterMode = False
   Application.CutCopyMode = False We then turn the filters off and clear the Clipboard.

Using this method we can easily make use of Excels **Autofilter** to get what we are after. What we need to do now is look at the **SpecialCells** Method. This method returns a Range object that represents all the cells that match the specified type and value.

expression.**SpecialCells(*Type*, *Value*)**

As I have already mentioned, this Method is the same as using **Edit>Go to-Special**. Recording a Macro doing just this is the best way to get the code you are wanting. Once you have done this a few times you will be able to skip the Recording bit.

8.6 AdvancedFilter

The last feature we will look at is the **AdvancedFilter**. This is feature is great for creating a list of unique items from a list. It can of course do a lot more than just this, but in the interest of keeping things simple I will only show how it can be used to create a unique list. Should you wish to go into this any deeper then I suggest Recording a Macro using this feature in the Interface and studying the code. If you have any problems or questions at all let me know and I will endeavour to help you. As with the **SpecialCells** Method it would pay to read the Excel help on **AdvancedFilter**

8.7 AdvancedFilter Method

Filters or copies data from a list based on a criteria range. If the initial selection is a single cell, that cell's current region is used.

Syntax

expression.**AdvancedFilter(*Action*, *CriteriaRange*, *CopyToRange*, *Unique*)**

expression Required. An expression that returns a **Range** object.

Action Required **Long**. The filter operation. Can be one of the following **XlFilterAction** constants: **xlFilterInPlace** or **xlFilterCopy**.

CriteriaRange Optional **Variant**. The criteria range. If this argument is omitted, there are no criteria.

CopyToRange Optional **Variant**. The destination range for the copied rows if ***Action*** is **xlFilterCopy**. Otherwise, this argument is ignored.

Unique Optional **Variant**. **True** to filter unique records only. **False** to filter all records that meet the criteria. The default value is **False**.

END OF EXCEL HELP

Let's look at an example of how we would copy a list that contained **non unique** items and paste it as list of **unique** items. Assume our list is in Column **A**.

```
Sub UniqueCopy()
 Sheet2.Columns(1).Clear
 On Error Resume Next
   With ActiveSheet
       .Range("A1", .Range("A65536").End(xlUp)).AdvancedFilter _
          Action:=xlFilterCopy, CopyToRange:=Sheet2.Range("A1"), Unique:=True
   End With
On Error GoTo 0
End Sub
```

This code would copy the list from Column **A** of the **ActiveSheet** to Column **A** of **Sheet2**.

1. Sheet2.Columns(1).Clear This is needed as the **AdvancedFilter** can generate a Run-time error if the **CopyToRange** contains data.
2. On Error Resume Next Will prevent any Run-time error should our list not contain enough data, ie less than 2 items.
3. .Range("A1", .Range("A65536").End(xlUp)) is used to define the range in Column **A**, starting from cell A1 down to the very last cell in Column **A** that contains an entry.

This simple bit of code would give us a unique list of items on a seperate Worksheet. You may not be aware that it is not possible to copy a unique list to another sheet via the **AdvancedFilter** in the Excel interface. I tell you this so when/if you Record a Macro, don't try and copy to another sheet. Just copy to any old range and change the code afterwards.

8.8 Summary

So by using the above examples as a starting point it is more often than not possible to do a task in VBA that will run very quick and clean. The only limit to the **built-in features** of Excel is usually your own imagination. I have yet to see any code that can operate as quickly or efficiently as one of Excels **built-in features**. They also give you the advantage of being able to Record the code needed to use the features. But remember, you should modify the code to become efficient! You will find that once you have been able to use one of Excels **built-in features** to do what you want, you will start thinking of other areas it can be adopted. I quite often will take an hour or more to rack my brains trying to come up with a efficient piece of code that fully utilises one of Excels **built-in features**. Once I have the idea it usually only takes a few minutes to put in place.

9. USER DEFINED FUNCTIONS

In this lesson we will look at **User defined functions** (UDF's) or **Custom functions** as they are also called. While Excel already has over **300** Functions available to us, at times it doesn't quite have the one we need, or if it does, it requires nesting several of these to create the formula we want. It is in these situation where UDF's come in very handy.

9.1 User Defined Functions Negatives

Creating a **UDF** requires the use of VBA, there is no way around it. By this I mean a user cannot Record a UDF, you have to create the UDF yourself. This is not to say though, that you cannot copy and paste bits of a Recorded macro into your UDF.

Before we look at UDF's in detail I should point out that UDF's do **not** have the same flexibility as a standard Procedure. a UDF cannot alter the structure of a Worksheet, such as change the Worksheet name, turn off gridlines, protect the Worksheet etc. They **cannot** change a physical characteristic of a cell, including the one that houses the UDF. So we cannot use a UDF to change the font colour, background colour etc of any cell. They cannot be used to try and change any part of another cell in any way at all. This means a UDF cannot place a value into any other cell except the cell housing the UDF. A UDF cannot use many of Excels built in features such as AutoFilters, AdvancedFilters, Find, Replace to name but a few!

We can use a UDF to Call (Run) another standard Procedure, but if we do the standard Procedure will then be under the same restrictions as the UDF itself. To make matters even worse, when you use a line of code in a UDF that cannot be executed you don't receive any error, other than one of the error values (eg; #VALUE!) in the cell housing the UDF. This can make de-bugging UDF's very difficult and leave one scratching their head! For this reason I recommend not calling a standard Procedure from a UDF, unless the standard Procedure was written solely for the UDF. So basically a UDF is very much as the name suggests a "User Defined Function" with the emphasis on Function. They should only be used to perform a calculation of some sort and not take the place of a Procedure. I find the most helpful thing to keep in mind when writing a UDF is that they are only an extension of the Paste Function (Function Wizard).

While all this negativity may leave you thinking "well what is the use of them then!" They can and do come in very handy so long as we are aware of the restrictions imposed upon them. When used in the correct context and you become comfortable with them you can build your own library of Functions that are not normally available to other Excel users.

9.2 User Defined Functions Positives

I hope the above section has not put you off UDF's as they really are very handy! I only point out the negative side to them because a new user to UDF's often thinks they are no different to a standard Procedure. In other words they have learnt all this really cool stuff that we can do with VBA but cannot apply it to UDF's and have no idea why their UDF is not working and give up in frustration.

In my opinion the biggest plus for UDF's is that we can use any of the WorsheetFunctions that are available to us while in the VBE. This means with a bit of imagination and "out of the box" thinking we can not only extend the use of the standard WorksheetFunctions we but can also write a whole

new Function that will do exactly what we want. Some of Excels standard Functions cannot be used across multiple Worksheets, but with the aid of a UDF we can change that.

But having said this don't be tempted to go overboard with UDF's in your Workbook. A UDF is not as efficient in a lot of cases as a very long nested standard formula. While it may look a whole lot neater in the cell, you can just about be certain that the code behind it is not as efficient as the code behind one of Excels standard Functions.

9.3 User Defined Functions Arguments

Now that we are fully aware of what we can and cannot do with UDF's let's move on to seeing some in action. A UDF can be contained within a standard module (Insert>Module) just as a standard Procedure. The difference is that all UDF's must begin with Function (as apposed to Sub) and end with End Function (as apposed to End Sub). So a UDF might look like:

```
Function MyOwn()

   'Code goes here

End Function
```

When we write a standard Procedure we would not normally use the Parenthesis to include any arguments. With a Function it is the opposite, we usually do use them. As you are aware most of Excels standard Functions do take arguments, with the exception of Function like NOW(), TODAY etc. We could if we wanted write a very simple UDF that takes no arguments:

```
Function SheetName()
   SheetName = ActiveSheet.Name
End Function
```

Or

```
Function MyPath()
   MyPath= ActiveWorkbook.FullName
End Function
```

This would of course returns the name of the sheet that houses the formula in the first example and the full name and path of the Workbook that house the formula in the second. They would be used on the Worksheet like:

```
=SheetName()
```

and

=MyPath()

Both take no arguments. Which means they must be entered exactly as shown above. If we tried to put an argument in either one, they would fail. Simply because we have not included any arguments in the Functions. Let's say we wanted the first function to take an argument and return the name of the sheet it refers to:

```
Function SheetName(rCell)
   SheetName = rCell.Parent.Name
End Function
```

The Parent Property returns the Parent Object of the Object that it is used with.

To use this we would put in any cell:

```
=SheetName(Sheet3!A1)
```

Where **Sheet3!** could be any sheet and **A1 could** be any cell on the sheet.

While this UDF will and does work, it is missing something that we should always do. That is declare any Variables. In the case of Functions the name of the Function is a Variable as are any arguments supplied. The correct way to write this Function would be:

```
Function SheetName(rCell As Range) As String
   SheetName = rCell.Parent.Name
End Function
```

As you can see we have declared our argument rCell as a range. This is because we are not the slightest bit interested in the Value of the chosen cell, all we want is the cell itself so that we can use the Parent Property. The "As String" after the Parenthesis is telling the Function to return a **String** (in this case the name of a Worksheet).

A word of caution with declaring the return type to a UDF. Unless you are certain it cannot be any other type omit declaring it. In the above example we can be certain that we will only ever return a String. But if we were dealing with numbers we must be very cautious as we usually cannot know for certain what size the number will be, or even if it is a whole number or not. For this reason I feel it is worth the trade-off and omitting the declaration of the Function name when dealing with numbers.

You will have noticed in the above examples that we parsed the result to the Function name itself.

SheetName = ActiveSheet.Name
and

SheetName = rCell.Parent.Name

In other words the cell that houses the Function will **always** be the cell that shows the result. This happens by default and cannot be changed. Remember we cannot change the value of any other cell with a UDF!

The number of arguments used in a UDF can vary, we are certainly not restricted to only one. Just remember though a Function used on a Worksheet that has a lot of arguments can be very confusing. So for this reason I would suggest keeping them to as few as possible.

9.4 Inputting Into Cells

Once we have written a UDF we can go ahead and place it in a cell to see the result (or possible problems). There are two ways this can be done. One is to just type the Function directly into a cell, preceded with an equal sign. The other way is to activate the **Paste Function** (Shift+F3) and scroll down to "**User Defined**". Then in the "**Function Name**" box to the right you will see a list of all UDF's available to the Active Workbook. Select the Function and then use the Wizard as you would for any normal Function. This is perhaps the easiest method of the two as it will eliminate any possible typos. If we do type it straight in and we miss-spell the name of the Function you will see the **#NAME?** in the cell. This is Excel telling you that it cannot find any Functions by that name. When you type a standard Function into a Worksheet cell it is always good practice to use all lower case, this way Excel will automatically capitalize the Function name if we have used it in the right context or not misspelt it. This does not happen with UDF's.

We can also nest our UDF with a Standard Function or vice versa. This can give our UDF even more versatility. For example if we had a UDF that only summed the last three entries in a Range we could use:

```
Function Sum3Max(rCell As Range)


Sum3Max = _
   WorksheetFunction.Sum _
      (WorksheetFunction.Large(rCell, 1), _
       WorksheetFunction.Large(rCell, 2), _
       WorksheetFunction.Large(rCell, 3))

End Function
```

This would work just fine, but only if there are three or more cells in the range we chose, for example:

```
=Sum3Max(A1:A10)
```

Would return a **#VALUE!** error if there are less than 3 numeric entries in the range **A1:A10**. We could overcome this in a number of ways by nesting our UDF within a standard Function:

```
=IF(COUNT(A1:A10)<3,"",Sum3Max(A1:A10))
or
=IF(ISERR(Sum3Max(A1:A10)),"",Sum3Max(A1:A10))
```

Both methods would prevent the **#VALUE!** occurring. However I would go with the first method as it will only hide the **#VALUE!** if the number of numeric cells are less than three. The **ISERR** will hide it any many other error types and you won't know why. It is not good practice to hide any and all errors for any reason as the error is telling you something. Remember UDF's are hard enough to de-bug as they are!

You may find that you have written a UDF that would be very useful in a number of different Workbooks. If this is the case you can either copy the Function code into all workbooks (you will get sick of that pretty quick) or simply store it in a standard module within your Personal Macro Workbook. This way it will always be available to all Workbooks. The down side to this is that you will **have** to precede the UDF name with: **PERSONAL.XLS!** So if we stored our **Sum3Max** Function in our Personal Macro Workbook. When used in a cell it would look like:

```
=PERSONAL.XLS!Sum3Max(A1:A10)
```

Not really a problem if we used the **Function Wizard**, but definitely a pain if we type it in the cell directly. There is a way that can help (but not eliminate) this and that is to save a Workbook as a one letter name eg; **X.xls** and place this in the Start folder of Excel, normally **C:\WINDOWS\Application Data\Microsoft\Excel\XLSTART**. Then you could place the Function within a module in the Workbook **X.xls** and use:

```
=X.XLS!Sum3Max(A1:A10)
```

But in most cases, for every upside there is downside! In this case it would mean that Excel would need to open two Workbooks as hidden each time Excel is started. Unless of course you don't have a Personal Macro Workbook.

The other thing we can do to make our UDF a bit more user friendly is to include a brief description of what it is designed to do. It's all very well to be liberal with our comments within the code itself, but that will not be of any use when we call it up from within the **Paste Function**. All of Excels Functions include a brief description in the bottom of the **Paste Function** dialog box whenever they are selected. Most users do not include this as there is a bit of a trick to doing so. Here what you can do.

1. Record a Macro and call it the name you will give your function.
2. In the "**Description**" box type a brief description of what it does.
3. Click **OK** to Stop Recording.

4. Now open the VBE and change the word "**Sub**" to "**Function**" Excel will automatically change the "**End Sub**" to "**End Function**"
5. Write the code as normal. Do not remove any apostrophise though.

So using the Function Sum3Max it would look like:

```
Function Sum3Max(rCell As Range)

' Sums the max values in the range as set by 'rCell

Sum3Max = _
   WorksheetFunction.Sum _
      (WorksheetFunction.Large(rCell, 1), _
       WorksheetFunction.Large(rCell, 2), _
       WorksheetFunction.Large(rCell, 3))

End Function
```

9.5 User Defined Functions Calculations

When we input a UDF into a cell it will only recalculate when any of the cells it is referencing changes value. This is normally just fine as there is normally no need for it to recalculate whenever any old cell changes. But at other times we may want to force a recalculation as often as possible. This can be done by making the Function what is known as **Volatile**. Excel already has a number of **Volatile** functions, with **NOW()** and **TODAY()** being the best known. These Volatile functions recalculate whenever any cell on the Worksheet calculates. To do this we simply place **Application.Volatile(True)** as the very first line within the Function. The **(True)** is optional as the default is True. So **Application.Volatile** will do exactly the same. You will most likely use **Application.Volatile** when dealing with Time periods. You may have a function that you want to retrieve the value of another cell each day, but only after midday.

```
Function AfterMidDay(rCell As Range)

If Time > 0.5 Then AfterMidDay = rCell

End Function
```

This Function would obviously need to be forced to recalculate so that it knows what the Time is. To do this we would simply use:

```
Function AfterMidDay(rCell As Range)

Application.Volatile
If Time > 0.5 Then AfterMidDay = rCell

End Function
```

Let's now look at another Function. This particular Function will sum all the cells is a range that have a Background colour the same as a nominated cell.

```
Function SumColor(rColor As Range, rSumRange As Range)
Dim rCell As Range
Dim iCol As Integer
Dim vResult

Application.Volatile (True)
iCol = rColor.Interior.ColorIndex

   For Each rCell In rSumRange
   If rCell.Interior.ColorIndex = iCol Then
      vResult = WorksheetFunction.Sum(rCell) + vResult
   End If
 Next rCell

SumColor = vResult
End Function
```

This would be used in a cell like:

```
=SumColor(A1,A1:A10)
```

Where cell A1 contains the Background colour of the cells we want to Sum. There is one drawback with this function though and that is it will not recalculate whenever the Background colour of any cell changes. This is because changing the Background colour of a cell does not cause a recalculation even though we have used Application.Volatile (True). I have only included it to show you the possible drawbacks to UDF's.

The use of Application.Volatile does come at a price though. If you have a lot of Volatile functions within the same workbook calculation can be slowed down dramatically.

9.6 Examples

What I will do now is include two User Defined Functions. I have purposely not written any comments in the code because I would like you to put on your thinking cap and see if you can put then to real use. This will of course mean you will need to understand what they are doing. Make full use of the De-bugging tools in the VBE if you need.

I will also help you write some User Defined Functions that you may have in mind for a real situation that you have.

```
Function ExtractNumber(rCell As Range)
Dim iCount As Integer, i As Integer
Dim sText As String
Dim lNum As Long
sText = rCell
```

```
For iCount = Len(sText) To 1 Step -1
 If IsNumeric(Mid(sText, iCount, 1)) Then
  i = i + 1
 lNum = CInt(Mid(sText, iCount, 1)) & lNum
End If

If i = 1 Then lNum = CInt(Mid(lNum, 1, 1))

Next iCount

ExtractNumber = lNum
End Function


Function SumOfAllSheets(SumRange As Range)
Dim Wst As Worksheet
Dim SThisSheet As String

Application.Volatile
SThisSheet = ActiveSheet.Name

For Each Wst In ActiveWorkbook.Worksheets

    If Wst.Name <> SThisSheet Then
        SumOfAllSheets = WorksheetFunction.Sum _
        (Wst.Range(SumRange.Address)) + SumOfAllSheets
    End If

Next Wst

End Function
```

9.7 Summary

User Defined Functions are very handy to know and are especially good if you combine then with some of Excel's existing Worksheet Functions. As long as you remember that they cannot act on any other cell and cannot change the structure of the Workbook in any way at all. Be careful that you are not reinventing the wheel when you do write a User Defined Function if the same could be achieved via the use of a Worksheet Function then you are better off doing so. Unless of course it requires a deeply nested function to be written many time over.

10. CONTROLS

Please cross reference the ActiveX Control Examples.xls Workbook for this lesson.

Prior to Excel 97, the only Controls available to the user on the spreadsheet were Controls from the Forms Toolbar. While these Controls certainly served a good purpose, they have nowhere near the flexibility of the Controls that are now available in later versions of Excel. Basically, the only thing that can be done with the Controls from the Forms Toolbar was assigning a pre-recorded or pre-written macro. Now in Excel 97 and 2000 the user has access to what are known as ActiveX Controls. The Controls on the Forms Toolbar are still provided for backward compatibility with earlier versions of Excel. While these Controls are really only for backward compatibility, I strongly believe that if the only reason a Control is needed on your Worksheet is to run a pre-recorded or pre-written macro, then a Control from the Forms Toolbar is a good choice. The reason being that ActiveX Controls found on the Control Toolbox Toolbar do carry a lot of overheads. This means if your Workbook contains a lot of Controls, it can have an adverse affect on the size of your Workbook if all the Controls are from the Control Toolbox Bar, ie; ActiveX Controls.

To display the Forms Toolbar, go to **View>Toolbars>Forms** and to display the Control Toolbox Toolbar, go to **View>Toolbars>Control Toolbox.**

What we will focus on here is the Controls from the Control Toolbox Toolbar. To start off with, I will run through a brief description and possible purpose of the commonly used Controls.

10.1.1 Checkbox

A Checkbox is generally used to allow the user to indicate a choice. By default, a checkbox has two possible states; TRUE or FALSE, or CHECKED or UNCHECKED. It is possible though, to set the checkbox so that it can actually have three states, TRUE, FALSE and NULL, or CHECKED, UNCHECKED and INTERMEDIATE.

10.1.2 Textbox

The Textbox Control is generally used to allow the user to type text into it. While the purpose of this Control itself is quite simple, it still allows the developer to set it in such a way that would best suit their purpose.

10.1.3 CommandButton

A Command Button is nearly always used to activate pre-written VBA code. The most common uses would be as an "OK" button, or "CANCEL" button. But they are certainly not limited to just these two options.

10.1.4 OptionButton

While an Option Button is very similar in function to the Checkbox Control, there should be a fundamental difference in the purpose that they are used for. An Option Button will allow the user to choose one of many similar mutually exclusive options. For this reason, they are usually displayed in what is known as "Groups". When they are in "Groups", it is only possible to choose one option of the same Group. A simple example of this may be that you may have five Option

Buttons grouped, with each of them representing a colour. If the user selected Option Button 1, the colour they have chosen would be red. If they then chose Option Button 2, the colour they have chosen would then change to blue. In other words, Option Button 1 would become de-selected while Option Button 2 is selected. An Option Button has two states; they are TRUE or FALSE or SELECTED and DESELECTED and cannot be set to have three states like the Checkbox.

10.1.5 ListBox

The ListBox is, as the name suggests, is a box that contains a list of items. The ListBox can be set to allow the user to select only one item at a time, or set to allow the user to select more than one item at a time. The list that it can display can be one column of data or more.

10.1.6 ComboBox

The ComboBox, while similar to the ListBox in that it will display a list for the user to choose from, will only ever allow the user to select one item at a time. It also can only display a one column list.

10.1.7 ToggleButton

The ToggleButton's function is virtually identical to that of the CheckBox Control in that it allows the user to make a choice of two states, TRUE or FALSE. It can also be set to accept three states, TRUE, FALSE or NULL.

10.1.8 SpinButton

The SpinButton Control is used to allow the user to increment a numerical value. A SpinButton would generally be used when we only want the user to cycle through a range of numbers set by us.

10.1.9 ScrollBar

The ScrollBar is almost identical in function to the SpinButton, except that as the name suggests, it also has a bar that scrolls. While a SpinButton can only change a value, by hitting the up or down arrow, the ScrollBar can do this but also allows the user to move the ScrollBar itself.

10.1.10 Label

The main function of the Label Control is to display static text. While this is it's main function and it is rarely used for much else, it can be set in such a way as to be used like a TextBox.

10.1.11 Image

The Image Control is used to house an image or picture; eg; bitmap, metafile, etc., on the Worksheet. By placing an image in the Image Control, it allows us to determine how the image will be displayed eg; clipped, stretched, tiled etc.

When we use Excel we are actually most of the time using an ActiveX Control to change, alter, display, increase, decrease, select, choose etc our choice. To see what I mean activate the Options dialog box (Tools Options). Now click around on each page tab and you will see most of the Controls mentioned above. The Checkbox is probably the most frequent Control on this dialog box. In case you are wondering the square boxes are CheckBoxes, while the round ones are OptionButtons. The rectangular boxes with the drop arrow on the right with a choice displayed are ComboBoxes. If you select the Edit page you will see a SpinButton that sets the Decimal places

within a TextBox. On the Custom List page you will see two ListBoxes. Have a click around with these as they will give you a good idea of the type of circumstance that a particular Control would be used for.

The Controls discussed above are only the most commonly used Controls from the Control ToolBox Toolbar. It does, however, allow us if needed to access many more Controls. To see a list of these, there is a button at the very bottom of the Control ToolBox Toolbar, which is called More Controls. It has a symbol of a hammer and a spanner in the shape of an "X". If you click on this button, you will see a list of all other available Controls. For obvious reasons, we won't be going through any of these at this stage.

10.2 Control ToolBox Toolbar

Let's now go through the Toolbar and see how it would be used.

By default, the Control ToolBox Toolbar is what is known as a "floating toolbar". We can change this quite easily if it has not been done already by either dragging it to the top of the Workbook where the other toolbars are located and releasing, or by clicking the blue title bar, in this case "Control Toolbox". When we do this, the toolbar becomes what is known as "docked". Whether you have the toolbar docked or floating is entirely up to you.

Other than the Controls already discussed, you will notice that there are three buttons represented by symbols at the top of the Control Toolbox Toolbar. As with all toolbars, to get the name of the particular button, simply wave your mouse pointer over it.

The first one is what is called "Design Mode". This is represented by a symbol of a ruler, triangle and pencil. If you click this button you will set the Control Toolbox Toolbar and any Controls on the Toolbar or the active Worksheet to "design mode". When they are in design mode they have no functionality other than allowing us to manipulate them in a way which suits our purpose. You will probably have noticed that when you clicked the "Design Mode" button another small floating toolbar appeared. By clicking this you will exit design mode.

The next button is called "Properties" and is represented by a picture of a sheet of paper and a hand pointing to it. Clicking this button will display what is known as the "Properties" window of the selected Object. An Object in this context would be a Control from the Toolbar. If no Control is selected, the Properties window will be for the active Worksheet itself. To close the Properties window, simply click the X in the top right hand corner of it.

The third and final button is called "View Code" and is represented by a picture of a sheet of paper and a magnifying glass. Selecting this button will take us into the Visual Basic Editor (VBE) of the current Workbook. By default, the module that will be displayed will be the Private Module of the active Worksheet. The reason for this is that all ActiveX Controls, once attached to a Worksheet become an Object of the Worksheet Object itself.

10.3 Properties

As you have no doubt realised, all Controls have their own Properties, some of which are unique to the particular Control itself, while others are common across all Controls. Let's use a Checkbox to look at the Properties of a Control. The first thing we need to do is attach (or embed) a Checkbox

to a Worksheet. This is simply done by a single left click on the Checkbox Control and then a single click on the Worksheet where you wish the Control to appear. You will also notice that by doing this, Excel will automatically put us into "Design Mode". We can now access the Properties of this Control in two ways. The first is by selecting the Properties button from the Control Toolbox Toolbar and the second is by right clicking on the Control and selecting "Properties".

The Properties window has two tabs on it, both tabs contain the same properties, the only difference is that one is "Alphabetic" and the other is "Categorised". I will use the second tab (Categorised) to look at the most likely used Properties. When we change the Property of any Control in this way, we are changing it in what is referred to as "design time". If we change the Property of a control during the execution of some VBA code we are changing it in what is referred to as "run time". To alter the Property of any Control we either type the Property in the box to the right of the Property or select it from a drop down list of choices.

Let's now look at some Properties. It is important to note that the all Properties of a Control can only be changed while in "Design Mode" or at "Run Time" via VBA.

10.4 Appearance

As the name suggest these Properties will change the appearance of the Control. Most of these Properties are common to all Controls. The exception is the Caption Property.

10.4.1 Alignment

This will change the position of the Checkbox from right to left. In other words it will reverse the Control.

10.4.2 BackColour

This simply allows you to choose from any of the colors available to Excel. Changing this will change the color of the Control.

10.4.3 BackStyle

This will give you two choices of having either a Transparent Control or Non-transparent (Opaque)

10.4.4 Caption

This is where you would type a Caption that you want the user to see. The default is always the name of the Control and a number. The numbers will follow in sequence, ie Checkbox1, Checkbox2, Checkbox3 etc

10.4.5 ForeColor

ForeColor refers to the Font color of the Caption.

10.4.6 SpecialEffect

This is where you can alter the appearance of the chosen Control. We can make it either Flat or Sunken(3D effect). Some Controls have up to six SpecialEffects.

10.4.7 Value

As mentioned above this is where you would set it's default value. TRUE would make it appear checked, while FALSE would make it appear unchecked. We could also set it to NULL, but only if we have set it's "TripleState" to TRUE. We will see this soon.

10.4.8 Behaviour

It is under this Category that we can set how the Control will act under certain circumstances. With the exception of "TripleState" these Properties are common to most Controls.

10.4.9 AutoSize

In the case of a Checkbox this would only effect the size of the Checkbox if the Caption was altered. It takes a Boolean value of either TRUE or FALSE. Let's say we set it to TRUE then changed the Caption to a very long word, the CheckBox would automatically change size to accommodate the new text. If we left it set to FALSE it would not.

10.4.10 TextAlign

This has three settings, Left, Right and Center. In the case of the CheckBox it only applies to the Caption. If the Control was a type that did not have a Caption Property, it would apply to the Text or Number it holds.

10.4.11 TripleState

As discussed this is only available to a CheckBox or ToggleButton. It takes a Boolean, TRUE or FALSE. When set to TRUE the Control Value can be either TRUE, FALSE or NULL. When set to FALSE the Value can only be TRUE or FALSE.

10.4.12 WordWrap

This is exactly the same as setting a cells Alignment Property to "Wrap Text" under the Format Cells dialog. As with the TextAlign, if the Control has a Caption Property it applies only to the Caption. If the AutoSize Property is set to TRUE the WordWrap will not behave as expected.

10.4.13 Font

This is the only Property here and as the name suggest it allows us to alter the Font. It is important to note that the Font of any Control cannot be changed at Run time. Again, if the Control has a Caption Property it applies to the Caption only.

10.4.14 Misc

This is always the largest category for a Control and houses all the Properties that do not fit under the above Categories. I will not describe all of these as some are rarely used. For the ones I do not describe you can have Excel display the Help Topic for the selected Property by selecting it and pushing F1

10.4.15 (Name)

Common to ALL controls, this is where you define a name for your Control. Once the Name is set you then use this to identify your Control. No two Controls on the same Worksheet can have the same name. The Name follows the same naming convention as for named ranges. As with the Caption, the default name is always the type of Control followed by a number. It is good practice to name your Controls in a descriptive manner and also include some method of being able to identify the Control type by looking at the Name. For example if you have a CheckBox that is used to change the Back Color of a cell you might use: ChBxCellColor. The name can only be set at Design time.

10.4.16 Enabled

This is where we can disable a Control so that a user cannot access it in any way. It takes a Boolean and if set to TRUE the Control becomes disabled. It should be noted that we can still access the Control when it is Disabled via VBA. In other words it only Disables the Control for the user.

10.4.17 GroupName

This Property only applies to OptionButtons. By default the GroupName will be the Worksheets Tab name at the time the Control was added (embedded) to the Worksheet. If the Sheet Tab name is changed after the Control is added the GroupName will not reflect the change. A set of OptionButtons that all have the same GroupName will only allow the user to set the Value Property to TRUE (selected) of one OptionButton at one time. Why the CheckBox has this Property I honestly do not know!

10.4.18 Left

This determines the position of the Control on the Worksheet. Zero would place the Control in the very top left of the Worksheet, ie cell A1.

10.4.19 LinkedCell

This would be any cell that you nominate to store the current Value Property of the Control, eg TRUE or FALSE.

10.4.20 Locked

Very similar to the Enabled Property and is usually set to work in conjunction.

10.4.21 Visible

This takes a Boolean and when set to FALSE the Control is no longer Visible if we are not in Design Mode.

All the Properties that are mentioned above are used with most Controls with exception of Caption, TripleState and GroupName. What we will do now is look at the most likely used Properties which are often unique to certain Controls. Let's start with the ListBox.

10.5 ListBox Properties

10.5.1 BoundColumn

When you have a ListBox it can display more than one Column of data at a time as well as more than one Row. When you set the BoundColumn for a ListBox it will determine the Column that is returned as the current value for a ListBox. Lets say we had a ListBox with three Columns of data and 10 Rows. If we set the BoundColumn to 2 and then selected the fifth row the current value of the ListBox would be whatever is on Row 5 - Column 3. The reason it would be Column 3 and not Column 2 is because the first Column is always 0 (zero) as is the Row.

10.5.2 ColumnCount

This where we can set the number of Columns to display in our ListBox. Setting this to 0 (zero) means no Columns will be displayed. To display all available Columns you set this to -1 If the ListBox is *unbound to a **datasource the limit for ColumnCount is 10 ie; 0-9

10.5.3 *unbound

Describes a control that is not related to a worksheet cell. In contrast, a bound control is a data source for a worksheet cell that provides access to display and edit the value of a control

10.5.4 **data source

The location of data to which a control is bound, for example, a cell in a worksheet. The current value of the data source can be stored in the **Value** property of a control. However, the control does not store the data; it only displays the information that is stored in the data source.

10.5.5 ColumnHeads

Must be either TRUE or FALSE (Boolean). Setting it to TRUE will display a single row of Column headings in your ListBox. These cannot be selected if the first row of data is used as the ColumnHeads. When using a range of cells to fill a ListBox (ListFillRange described below) the Row immediately above the first Row is used as the ColumnHeads. This means if the ListFillRange was A2:D50 the range A1:D1 would be your ColumnHeads. If the ListFillRange was A1:D50 the ColumnHeads would be Excels Column headings, ie; A:D

10.5.6 ColumnWidths

This determines the width of each Column used in a ListBox. The setting must be a String. and each width separated with the PC's List separator, usually the ; (Semicolon). See the help text below:

Setting **Effect**

90;72;90	The first column is 90 points (1.25 inch); the second column is 72 points (1 inch); the third column is 90 points.
6 cm;0;6 cm	The first column is 6 centimetres; the second column is hidden; the third column is 6 centimetres. Because part of the third column is visible, a horizontal scroll bar appears.
1.5 in;0;2.5 in	The first column is 1.5 inches, the second column is hidden, and the third column is 2.5 inches.
2 in;;2 in	The first column is 2 inches, the second column is 1 inch (default), and the third column is 2 inches. Because only half of the third column is visible, a horizontal scroll bar appears.
(Blank)	All three columns are the same width (1.33 inches).

10.5.7 ListFillRange

This Property takes a range address or name as it's value. For example A1:D50 or MyRange (in the case of a named range) are valid entries.

10.5.8 ListStyle

This Property determines how your list will look. There are only two choices fmListStylePlain and fmListStyleOption The first is the default and has no real visual effect. The second will place small squares to the right of each Item in the ListBox which become checked when the user select the Item.

10.5.9 MatchEntry

This Property is used to assist the user in looking for a particular Item in the list. It takes effect as the user starts to type.

Constant	Value	Description
fmMatchEntryFirstLetter	0	Basic matching. The control searches for the next entry that starts with the character entered. Repeatedly typing the same letter cycles through all entries beginning with that letter.
FmMatchEntryComplete	1	Extended matching. As each character is typed, the control searches for an entry matching all characters entered (default).
FmMatchEntryNone	2	No matching.

If for example your list had the entries Aardvark,Absolute,Acorn,Addict etc and you set the MatchEntry to fmMatchEntryFirstLetter and the user typed A or a they would see Aardvark, if they then typed Ad or ad they would see Addict.

10.5.10 MultiSelect

This determines whether the user can select more than one Item in the ListBox.

Constant	Value	Description
fmMultiSelectSingle	0	Only one item can be selected (default).
fmMultiSelectMulti	1	Pressing the SPACEBAR or clicking selects or deselects an item in the list.
fmMultiSelectExtended	2	Pressing SHIFT and clicking the mouse, or pressing SHIFT and one of the arrow keys, extends the selection from the previously selected item to the current item. Pressing CTRL and clicking the mouse selects or deselects an item.

When the setting is fmMultiSelectMulti the user can select more than one item in the ListBox.

10.5.11 TopIndex

Sets and/or returns the Item that will appear in the top of the List. So if you had a list that contained 10 rows and set TopIndex to 5 the user would only be able to see the last 6 rows. The first row is 0 (zero).

Lets now look at the SpinButton and Scrollbar

10.6 SpinButton and ScrollBar

10.6.1 LargeChange (ScrollBar only)

This property sets the amount the user can move when the user clicks between the Scroll box and Scroll arrow.

10.6.2 Max and Min

This property determines the Maximum or Minimum Value the SpinButton or ScrollBar will increment to. The setting must be a Integer.

10.6.3 Orientation

This will make the Control either vertical or horizontal. The default is for fmOrientationAuto which means Excel will position the Control based on the Controls dimensions.

10.6.4 SmallChange

Determines how movement will occur whenever a user spins or scrolls up or down. The default setting is 1. The Value must be an Integer.

Let's now look at the TextBox and some of it's Properties that have not been mentioned.

10.7 TextBox and ComboBox

10.7.1 AutoWordSelect

This Property takes a Boolean and determines what will be the base unit that is used to extend a selection. When set to TRUE the base unit is a word. When set to FALSE the base unit is a single character. If AutoWordSelect is set to TRUE and the user places the mouse insertion point into the TextBox in the middle of a word and then extends (drags) the selection, the entire word is selected. Doing this when AutoWordSelect is set to FALSE would mean only one character at a time would be selected.

10.7.2 DragBehaviour

This Property can be either enabled (fmDragBehaviorEnabled) or disabled (fmDragBehaviorDisabled). When enabled the user can drag-and-drop (cut or copy and paste). If the Property is disabled and the user drags within the TextBox any text is only highlighted.

10.7.3 EnterFieldBehaviour

This Property sets the method in which the text is selected when entering a TextBox. Its two settings are fmEnterFieldBehaviorSelectAll (default) and fmEnterFieldBehaviorRecallSelection. when left set as fmEnterFieldBehaviorSelectAll the entire content of the TextBox is selected when the user enters the TextBox. If set to

fmEnterFieldBehaviorRecallSelection the selection is the same as the last time the Control was active.

This only applies when the user Tabs to the Control.

10.7.4 HideSelection

Takes a Boolean and determines whether selected Text still appears selected when the Control no longer has Focus. Setting this to TRUE (default) means Text is not highlighted unless the Control has Focus.

10.7.5 MaxLength

This Property sets the maximum number of characters that can be placed into a Control. The default is 0 (zero) which means the Control has no limit set and any number of characters can be entered.

10.7.6 MultiLine (TextBox only)

Specifies whether a **Textbox** can have multiple lines of Text. The setting is a Boolean with the default being **TRUE**. If set to FALSE the Textbox will only ever have one line of Text regardless of the number of characters and size of the **Textbox**.

10.7.7 PasswordChar (TextBox only)

This Property determines whether ***placeholder** characters are displayed when the user types in a **TextBox**. The setting can be any String.

10.7.8 *placeholder

A character that masks or hides another character for security reasons. For example, when a user types a password, an asterisk is displayed on the screen to take the place of each character typed.

So as you can see we are able to set the visual and actual effect that happens when the user does anything to a Control. The Properties can be used in different combinations to produce different effects. Most Properties can be set either at Design-time or Run-time. While some Properties can only be set at Design-time. The Font type is one of these. While we can set the Fonts attributes Bold, Italic, Underlined etc we cannot change the Font type, eg; Ariel to Times New Roman at Run-Time.

10.7.9 Controls Parent

As mentioned earlier whenever a Control is placed on a Worksheet is becomes a Object of the Worsheet. This means it is part of the Worksheets Object collection. This basically means that if we want to access a Control that is on a Worksheet we must first go through the Worksheet Object as the Controls Parent is the Worksheet that has the Control embedded in it. Lets say we have a TextBox on a Worksheet who's CodeName is Sheet1 and we want to Select it. We would use:

```
Sub SelectKnownTextBox()
   Sheet1.TextBox1.Select
End Sub
```

Very easy indeed! But lets assume we have no idea of the name of the TextBox all we know is that a TextBox does exists on Sheet1. In this case we will need to Loop through all Objects on the Worksheet until we find one that is a TextBox. We can determine this by using the **ProgId** Property. But before we go and Loop through ALL Objects we can narrow our search down to the type of Object we are interested in. For ActiveX Controls the type is an OLEObject and the OLEObject is a member of the OLEObjects Collection. So we could use this method:

```
Sub SelectUnknownTextBox()
Dim tBox As OLEObject

   For Each tBox In Sheet1.OLEObjects
      If tBox.ProgId = "Forms.TextBox.1" Then
         tBox.Select
      End If
   Next tBox

End Sub
```

So as you can see we have used a **For Each** (with each referring to OLEObjects) to Loop through only OLEObjects that are on Sheet1. We then use the ProgId Property to check and see if the OLEObject is a **TextBox** or not. The list of identifiers for ActiveX Controls is shown below. The

identifier of a ActiveX Control can also be seen in the Formula bar when the Control is Selected, eg; **=EMBED("Forms.TextBox.1","")**

To create this control	Use this identifier
CheckBox	Forms.CheckBox.1
ComboBox	Forms.ComboBox.1
CommandButton	Forms.CommandButton.1
Frame	Forms.Frame.1
Image	Forms.Image.1
Label	Forms.Label.1
ListBox	Forms.ListBox.1
MultiPage	Forms.MultiPage.1
OptionButton	Forms.OptionButton.1
ScrollBar	Forms.ScrollBar.1
SpinButton	Forms.SpinButton.1
TabStrip	Forms.TabStrip.1
TextBox	Forms.TextBox.1
ToggleButton	Forms.ToggleButton.1

So using the above list we can access any of the above mentioned ActiveX controls on any Worksheet.

10.8 Events

As we have Events for the Workbook Object and Worksheet Object we also have Events for ActiveX Controls. We can see all the Events associated with a particular Control by viewing it's code and looking in the Procedure box. We can gain access to the code for a Control in two ways. Right click on the Control and select "View Code" or Double click the Control. For both methods we must be in Edit Mode.Whichever method we use, Excel will open up the Private Sub of the Worksheet. The Procedure written by default will be the default Procedure for the Control. This is usually the Change Event. For example double clicking a TextBox gives us:

```
Private Sub TextBox1_Change()

End Sub
```

We can then see all the Events this Control has by placing our mouse insertion point anywhere within the Procedure and clicking the drop down arrow in the Procedure box (top right of the Module). Most Controls have around 15 Events that are available to them. As with the Events for a

Workbook or Worksheet any code within an Event Procedure for a Control will run whenever the Event occurs. The way in which we apply these Events is purely up to us.

For the rest of the lesson I have attached a Workbook that has Controls on it and some code placed within the Event Procedures. Browse through these at your leisure as the code is only very simple. We will look at the Events for Controls in the Excel UserForms course.

11. WHAT NEXT?

You can continue your Excel Training by purchasing more lessons online at

http://www.spreadsheetml.com/exceltraining/excel_training.shtml

11.1 Excel Training Level 2

- **Lesson 1 - Naming Constants. Dates and Times**
  In this lesson, we review the basic concepts of Excel and go into details the concept of Naming Constants and how Excel interprets Dates and Times.
- **Lesson 2 - Worksheet Formulas**
  In this lesson we will look at the thing that Excel does best -- Calculations. Excel has over 300 built in functions that can perform simple additions to some very obscure engineering functions. We will also look at how we can easily perform Nesting, Formula Auditing and Troubleshooting.
- **Lesson 3 - Specific Worksheet Formulas**
  We would like to show you some of Excel most useful and popular functions like COUNT, Address, COUNTIF, DATEVALUE, HLOOKUP, MATCH, MAX and VLOOKUP.
- **Lesson 4 - Decision making and extracting data**
  This lesson provides a good overall look at deciding what functions to use and how to use them. There are no hard and fast rules as more often than not you will need to use a nested formula that will incorporate many types of functions. Extracting data from tables and databases is something that is done quite frequently from within Excel. This lesson is meant as a guide to extracting data from a database or table.
- **Lesson 5 - Validation and External References**
  One thing you can be certain of when allowing other users to input into a spreadsheet you have designed is that they undoubtedly will enter incorrect data at some time or another. This lesson shows you how Excel can help you perform validation.
- **Lesson 6 - AutoFilters, Advanced Filters and Range Names**
  This lesson shows you how you can use Excels Filters (Advanced and Auto) to display virtually any criteria we want, so long as we have set up our table or list is a sensible manner.
- **Lesson 7 - Introduction to Charts**
  Charts in Excel can range from the very simple to the very complex and can be used for a multitude of reasons. They can be used to keep track of spending, stock performance, statistics, employee details and much more. Most charts are used to show a comparison of past data in a highly visual style. A well set up chart should be able to inform the user at a glance exactly what the picture is that it is painting. We have all heard the saying "A picture paints a thousand words" and this should always hold true with charts in Excel. If an informed user looks at a chart and cannot tell what it is that the chart is representing, then it would be fair to say that the chart has not been set up very well. Excel is arguably the best of the Office Applications when it comes to creating and/or using charts. It allows us to create

almost any kind of chart needed with minimal effort. As can be said with a lot of Excels features and capabilities, we often find ourselves using only a very small aspect of them, charts are no different in this respect.

- **Lesson 8 - FOUR NOT-SO-COMMON Handy Features**
  For this lesson we thought we would look at a few of Excel's little known but very handy features. These are Custom Formats, Text to Columns, Save Workspace and Custom Lists.
- **Lesson 9 - Protecting and Hiding**
  There are times when using Excel that we do not want other people to make changes to our Workbooks, Formulas, Worksheets or Cells. We can achieve this in many ways, ranging from hiding data to preventing the Workbook from opening at all without a password to protecting the Workbook or Sheet.
- **Lesson 10 - Custom Views and Report Manager**
  Custom Views can be used on it's own or in conjunction with the Report Manager. The purpose of Custom Views is to save the appearance of a chosen Workbook. This means we can have more than one person work on a Workbook and each person can save their own Custom View. It also saves a lot of time with Print settings that can be very time consuming and fiddly.

11.2 Excel Training Level 3

- **Lesson 1 - Advanced Formulas**
  in this lesson, we will show you how to easily write those mega functions that you have seen elsewhere. This includes concepts like Nesting, Array Formulas and Dfunctions. Nesting means to use the result of one formula as the argument in another. Array formulas allow you to perform calculations and return a result. Dfunctions are known as Excel Database functions.
- **Lesson 2 - Advanced Filters**
  In this lesson, we will have a good look at one of Excel's arguably most useful features and that is the Advanced Filter. Advanced Filter will allow us to nominate where we would like our filtered data to be placed. The choices are: Filter the list, in place or Copy to another location. Advanced Filter has a built-in function that will allow us to filter by unique records. Advanced Filter allows us to use a formula as our criteria.
- **Lesson 3 - Pivot Tables (Part 1)**
  In this lesson we will start to look at arguably one of Microsoft Excel most powerful and useful feature, that is Pivot Tables. In a nutshell a Pivot Table takes two dimensional data (your spreadsheet) and creates a three dimensional table (the Pivot Table itself). They are a great way to produce statistical information from a table of data. We would use a Pivot Table to produce meaningful information from a table of information. You will recall in lesson 2 we looked at Excel's Advanced Filter feature and how it could be used to extract information from a table of data based on a set of criteria. A Pivot Table could be used on that same table to create a table that could tell us much statistical information about all the data contained within it.
- **Lesson 4 - Pivot Tables (Part 2)**
  In this lesson, we will look at constructing some more complicated pivot tables and extracting meaningful data from them. This includes Grouping Fields options where

we could, for example, group people (data) by the years (attributes) in which they were born (rules).

- **Lesson 5 - Pivot Tables (Part 3)**
  In the last lesson on Pivot Tables, we looked at setting up a complete pivot table and how to use the grouping and field options of Pivot Tables. We will also discuss some of the more advanced features like Calculated field and, more importantly, be fully aware of the pitfalls associated with them.
- **Lesson 6 - Scenarios and Goal Seek**
  In this lesson we are going to look at two of the tools that are specifically designed for use in "What-If analysis", and as such make up part of the "What-If Analysis Toolpack of Microsoft Excel". These tools are used to determine different outcomes of your data by changing different cells within a Worksheet model. The two tools we will discuss here are called **Scenarios** and **Goal Seek**. Both of these tools are used widely in financial, accounting and engineering businesses today, and once understood, can be a huge aid in determining various outcomes and projections.
- **Lesson 7 - Data Tables and Consolidation**
  In this lesson, we are going to look at two of the very handy, but little known features of Excel -- Data Tables and Consolidation. Data Tables are another one of the tools that are specifically designed for use in "What-If analysis". By now, you will be familiar with how "What-If Analysis" works and what it is used for. Data Tables are just a range of cells that are used for testing and analysing outcomes on a larger scale. Consolidation is a powerful feature of Microsoft Excel that enables you to combine data from separate worksheets into one consolidated worksheet. It also enables you to perform many calculations on this data, including 3-D formulas, which are formulas which refer to cells on multiple Worksheets.
- **Lesson 8 - Excel on the Web**
  One extremely handy feature that Excel offers is its ability to allow you to publish a spreadsheet on the web, so you can communicate your data to anyone you like, irrespective of whether they use or even have Microsoft Excel on their computers. In other words, your Excel spreadsheet can be viewed in a web browser. This could be useful for such things as employers wanting their employees to access things such as sales data from different areas, cost calculations, time sheets and many other uses. Basically, if your Excel Workbook is placed on the web, people anywhere in the world can access it, with or without Excel.
- **Lesson 9 - MACROS (Part 1)**
  In our last two lessons in our Excel Level 3 course, we will be taking a look at recording, editing and running macros using Excels Macro Recorder. A macro is simply an action or a set of actions you can use to automate a particular task or tasks. You use the Macro recorder a bit like using a video camera. You switch it on, record what you want to record, and then switch it off. In effect a macro is like a mini-program that performs the actions that you have recorded. Macros are a fantastic time-saving feature and are ideal for automating repetitive tasks. They are also useful if you are setting up an application for others to use as you can use macros to create buttons and dialog boxes to guide a user through your application as well as automating the processes involved.
- **Lesson 10 - MACROS (Part 2)**
  In this, our last lesson in our Excel Level 3 course, we will be delving further into the macro feature of Excel. You now know how to record and run both absolute and relative macros, and hopefully you now understand a little about the actual

macro (VBA) language. In this lesson we are going to create a mini-application using macros and we will assign these macros to an object.

11.3 Excel VBA (Visual Basic for Applications) Training Module

- **Lesson 1 - Introduction**
  VBA is short for Visual Basics for Applications. This is the standard Macro language used in most Microsoft Office products. The word "Applications" can represent any one of the Office products it is used within e.g.; Excel, Access etc. The VBA language is a derivative of Visual Basic (VB), which in turn is a derivative of the language Basic. The fundamental difference with VBA from VB is that VBA is (as the name implies) used within an Application. By far the most mature of these Applications when it comes to VBA is Excel. You will find as we delve deeper into VBA for Excel that we can modify the Application so it will behave in almost any way possible.
- **Lesson 2 - Common Objects**
  In this lesson we will look at the 4 most common (and arguably useful) Objects in VBA, these are: Application, Workbook, Worksheet and Range. The most useful Properties and Methods of these objects and the relationships between these objects will be explored. Sample codes of these objects will also be provided.
- **Lesson 3 - Variables**
  A Variable is used to store temporary information that is used for execution within the Procedure, Module or Workbook. You can name variables with any valid name you wish. For example, you could name a variable "David" and then declare it as any one of the data types supported by VBA. However, it is good practice to formalize some sort of naming convention. This lesson explores the details of Variables and also the various rules and naming conventions that you must know about.
- **Lesson 4 - Loops**
  This lesson we will focus on Loops. There are many varieties of these, but they are all basically the same in that they will repeat a line, or multiple lines, of code a set number times, or until a condition becomes True or False.
- **Lesson 5 - Effective Decision Making**
  IF Else And Or Not If The "If" Function in VBA for Excel is very similar to the "IF" function used in a Worksheet formula. It will return either True or False and it does no more or less than this. As with the "IF" used in the Worksheet formula the "If" in VBA can take up to two arguments, one for True and one for False. This lesson will guide you to the effective use of the "If" Function in VBA.
- **Lesson 6 - Workbook and Worksheet Events**
  There are times when you may like or want the macro to run whenever a particular **Event** happens. Let's say each time you open your Workbook you would like the current date inserted into a cell, or each time you activate a particular Worksheet you want all data on it to be cleared. With Excel we can do this and much more! Excel has what is known as Events. The two most common being either Workbook Events or Worksheet Events. The word Event in this context means something that occurs whenever a particular action takes place which effects the Workbook or Worksheet Object. This might be a Workbook opening or a Worksheet cell changing or a Workbook closing etc. This lesson will explore Workbook and Worksheet Events.

- **Lesson 7 - De-bugging**
  This lesson we will concentrate on De-bugging code and Error Handling. You will no doubt find that debugging is something you will be doing a lot in the early stages of learning. A good programmer will never think that their finished project doesn't contain any errors, if you do, you won't bother placing in any code to handle errors and that is a BIG mistake. There are many features in Excel that can make De-bugging code a reasonably easy task. To avoid confusion and overload we will discuss in detail what I believe to be the best method.
- **Lesson 8 - Worksheet Functions**
  In this lesson, we will focus Worksheet Functions in VBA for Excel. Excel has a number of functions that can be used in the Visual basic environment only, but it also has, at its disposal, a long list of the standard Worksheet Functions. When these are combined with VBA for Excel it adds even more power and flexibility to the Visual Basic environment. It is a great idea to use Worksheet Functions within your code as it can often means doing away with unnecessary filling up of ranges with formulas. This in turn will mean a faster Workbook.
- **Lesson 9 - User Defined Functions**
  In this lesson we will look at User defined functions (UDF's) or Custom functions as they are also called. While Excel already has over 300 Functions available to us, at times it doesn't quite have the one we need, or if it does, it requires nesting several of these to create the formula we want. It is in these situation where UDF's come in very handy.
- **Lesson 10 - Controls**
  Prior to Excel 97, the only Controls available to the user on the spreadsheet were Controls from the Forms Toolbar. While these Controls certainly served a good purpose, they have nowhere near the flexibility of the Controls that are now available in later versions of Excel. ActiveX Controls are available in later version of Excel. It is important to note that they carry a lot of overheads. This means if your Workbook contains a lot of Controls, it can have an adverse affect on the size and performance of your Workbook. This lesson explores Controls in details.

11.4 Excel VBA User Form Training Module

- **Lesson 1 - UserForms - Introduction**
  This lesson will introduce you to Excel VBA UserForms. UserForms were first introduced into Excel in Excel 97. It works like a Dialog box (sheet) and has superior Event handling that allows us to respond to a users actions in ways that were previously not possible. By this I mean we can have specific code run when the user clicks the control, enters, exits, double clicks, right clicks, types and much more. Basically we are able to capture any action that the user takes. If you want to give your projects a professional and consistent look and feel, then UserForms will certainly help you do this.
- **Lesson 2 - Filling UserForm Controls**
  The vast majority of UserForms that are designed within Excel are used so that users can easily select and input data. This also ensures that any entries that are entered into a spreadsheet are within the requirements needed. Excel has many Controls that can be placed on a UserForm that can make this not only easy for the user, but also for the designer of the UserForm. The two most useful Controls for

this are the "ComboBox" and the "ListBox". In this lesson, we will look at each of these Controls.

- **Lesson 3 - MultiPage Control**
  In this lesson we will look at the MultiPage control, arguably the most useful controls to use if your UserForm will be containing many different controls and/or you wish to have different controls associated with different aspects of your project. The other thing that we can do with a MultiPage is make our UserForm behave in the same manner as any one of Excel's standard Wizards. A Wizard by definition is an aid that steps you through a particular process. An example of this would be the Pivot Table Wizard or Chart Wizard.
- **Lesson 4 - Validating The Users Inputs**
  One of the biggest problems that is (or should be) faced by a developer (other than understanding just what they want) is developing a project that will only accept valid data. By far the quickest and easiest way to achieve this is to use a ListBox and/or a ComboBox that presents the user with only valid data. The approach is one that I will try to employ whenever possible. Unfortunately, it's not always possible to use a ComboBox or ListBox and we may need to allow the user to type in an entry. This means you will need to check whether the user has typed in a valid entry. In this lesson, we will explore how this can be achieved.
- **Lesson 5 - When to Apply Validation**
  What we shall look at in this lesson is when to reject or accept a users entries. By this I mean should we allow the user to completely fill out the UserForm first and then inform him/her that some of their entries are not valid, or would it be better to inform the user at each step.
- **Lesson 6 - Which Controls to Use and When**
  In this lesson, we will look at which controls should be use and for what purpose should we use them? Excel has basically a Control for every possible job type. It is important to know which Controls to use for which job. It is fair to say that there are no hard and fast rules for this as each project usually has something which makes it unique. So, we need to at times use a Control and modify it to suit our specific needs. However, there are certain guidelines that we can follow.
- **Lesson 7 - Passing Control Values back to a Spreadsheet, Passing range values to the UserForm Controls**
  In this lesson we will be looking at how we can pass values from the Worksheet to a Control or any number of Controls on a UserForm and also look at how we can do the opposite which is to pass the value from a Control back to the Worksheet.
- **Lesson 8 - Option Buttons and Checkboxes in Detail**
  In this lesson we will look at Option Buttons and Checkboxes in detail. The rules of which one to use and when are really quite simple and boils down to basically if the user should be allowed to make one selection only from a choice of many, then it should be OptionButtons. If the user should be allowed to make one or more choices from many, then it should be CheckBoxes.
- **Lesson 9 - Creating Template Controls, Using the RefEdit Control**
  Excel allows us to easily create Template Controls which we can format and name appropriately so that each time the Control is needed, rather than repeat the process over and over again, we can simply select the Template Control that we have created and place it on to a UserForm. This is extremely easy to do and can save many hours of repetitious work in the long run. We will look at Template Controls in this lesson. On top of that, we will look at the RefEdit Control. Normally

used on a user form, the RefEdit control will display the address of a range, or single cell, that you've entered (typed in) or selected.

- **Lesson 10 - Finding the ActiveControl, Creating Controls at Runtime**
  At times when creating a project in Excel VBA which incorporates the use of a UserForm, you may wish to create actual Controls via the choice of a user. The advantage of actually creating the Control is the fact that it will require the need for less Controls on the UserForm at any one time, which in turn can lessen overheads. We will look at how this can be done in this lesson.

12. EXCEL ADD-INS, TEMPLATES & TRAINING

12.1 Add-ins for Excel

- ConnectCode Duplicate Remover for Excel - a tool for finding and removing duplicate entries in Excel.
- ConnectCode Deluxe Add-In for Excel - a collection of commonly used add-ins for Excel. Use it to generate report on Excel formulas, generate calendar and random numbers, remove matching rows, name dynamic ranges and perform scheduled jobs.
- SparkCode Professional - a professional tool for creating dashboard reports in Excel using Sparklines.
- ConnectCode Barcode Font Pack - enables barcodes in office applications and includes an add-in for Excel that supports mass generation of barcodes.
- ConnectCode Number Manager - an add-In that improves productivity when working with numbers in Excel.
- Excel Sparklines Software Add-In - open source project to generate Sparklines.
- ConnectCode Text Manager - an add-In that improves productivity when working with text in Excel.

12.2 Excel Templates

- Free Excel Templates - resource for all kinds of Excel Templates.

12.3 Excel Training

- Free Excel Training - 10 Lessons of Excel Training - FREE.
- Excel Training - resource for all kinds of Excel Training.