CONTENTS

1. WHAT IS VBA ?

VBA, which stands for Visual Basic for Applications, is a programming language developed by Microsoft. Excel, along with the other members of Microsoft Office, includes the VBA language (at no extra charge).

VBA is the language that people use to develop programs that control Excel.

You can automate almost anything you do in Excel. To do so, you write instructions that Excel carries out. Automating a task by using VBA offers several advantages:

- Excel always executes the task in exactly the same way.
- Excel performs the task much faster than you can do it manually.
- If you're a good macro programmer, Excel always performs the task without errors.
- If you set things up properly, someone who doesn't know anything about Excel can perform the task.
- You can do things in Excel that are otherwise impossible — which can make you a very popular person around the office.
- For long, time-consuming tasks, you don't have to sit in front of your computer and get bored. Excel does the work on its own.

But there may be some risks too:

- You have to know how to write programs in VBA.
- Other people who need to use your VBA programs must have their own copies of Excel. It would be nice if you could press a button that transforms your Excel/VBA application into a stand-alone program, but that isn't possible.
- Sometimes, things go wrong. In other words, you can't blindly assume that your VBA program will always work correctly under all circumstances.
- Welcome to the world of debugging and, if others are using your macros, technical support.
- VBA is a moving target. As you know, Microsoft is continually upgrading Excel. Even though Microsoft puts great effort into compatibility between versions, you may discover that VBA code you've written doesn't work properly with older versions or with a future version of Excel.

Yet the overall balance is very positive.

2. VBA PROGRAMMING TOOLS

a. Installing and enabling VBA

Macro language viruses such as those written in VBA are relatively easy to write—even for a beginning programmer. As a result, Microsoft has added several levels of security to its Office programs in order to protect against macro viruses. The first level of security Microsoft has implemented is simply to disable macro language support for its Office programs. Disabling macro language support is now the standard for the normal installation of Office or any of its component programs.

You must install VBA and enable macro language support before you can access the VBA IDE and create your own projects.

To install or enable VBA, you must insert the CD that contains the Excel program into your computer and run the Office/Excel setup program by doing the following:

1. Double-click the Add/Remove Programs icon in the Microsoft Windows Control Panel (found on the Start menu).

2. If you installed Excel as part of Microsoft Office, click Microsoft Office (edition and version) in the currently installed programs box, and then click the Change button. If you installed Excel individually, click Excel (edition and version) in the currently installed programs box, and then click the Change button.

3. On the installation options screen in the Setup program, click the plus sign (+) next to Office Shared Features.

4. Select Visual Basic for Applications, click the arrow next to your selection, and then click Run from My Computer.

After the installation is complete, you may also need to change the macro security setting in Excel before you can run any VBA programs. To change the macro security setting in Excel 2007, do the following:

1. Click the Office button, and then click the Excel Options button.

2. Click the Trust Center button from the left-hand navigation menu, and then click the Trust Center Settings button, which opens the Trust Center window.

3. From the Trust Center window, click the Macro Settings icon from the left-hand navigation menu and choose the Disable All Macros with Notification option.

The Disable All Macros with Notification option is the default macro security level. This setting disables all macros present in an Excel file and provides you with a security warning in the Excel message bar.

You can decide whether or not to enable macro content by clicking the Options button in the Security Warning area, which launches a Security Alert window that allows you to enable the Macro and ActiveX content.

b. Displaying the Developer Tab

You need to make a small change so Excel will display a new tab at the top of the screen: Developer. When you will click the Developer tab, the Ribbon is going to display information that is of interest to programmers.

The Developer tab is not visible when you first open Excel; you need to tell Excel to show it. Getting Excel to display the Developer tab is easy (and you only have to do it one time). The procedure varies, though, depending on which version of Excel you use.

Excel 2010 Users, follow these steps:

1. Right-click any part of the Ribbon, and choose "Customize the Ribbon".
2. In the Customize Ribbon tab of the Excel Options dialog box, locate Developer in the second column.
3. Put a check mark next to Developer.
4. Click OK, and you're back to Excel with a brand-new tab: Developer.

Excel 2007 Users, follow these steps:

1. Choose File > Excel Options.
2. In the Excel Options dialog box, select Popular.
3. Place a check mark next to Show Developer tab in the Ribbon.
4. Click OK to see the new Developer tab displayed in the Ribbon.

c. Recording and running a Macro

Follow these instructions to record a macro:

1. Select a cell — any cell will do.
2. Choose Developer > Code >Record Macro, or click the macro recording button on the status bar. The Record Macro dialog box appears.
3. Enter a name for the macro.
4. Click in the Shortcut Key box and enter a shortcut, for instance Shift+P. Specifying a shortcut key is optional. If you do specify one, then you can execute the macro by pressing a key combination — in this case, Ctrl+Shift+P.

5. Make sure the Store Macro In setting is This Workbook.
6. You can enter some text in the Description box if you like. This is optional.
7. Click OK. The dialog box closes, and Excel's macro recorder is turned on. **From this point, Excel monitors everything you do and converts it to VBA code.**
8. Do whatever you want to do.
9. Choose Developer > Code > Stop Recording. **The macro recorder is turned off.**

You just created a new Excel VBA macro. You may want run the macro now:

1. You can use the keyboard shortcut if you specified one.
2. Another way to execute the macro is to choose Developer > Code > Macros (or press Alt+F8) to display the Macros dialog box.

d. Editing a Macro

So far, you've recorded a macro and you've tested it. To edit it, you will need to activate the Visual Basic Editor, which is VBA Integrated Development Environment (IDE).

Follow these steps:

1. Choose Developer > Code >Visual Basic (or press Alt+F11). The Visual Basic Editor program window appears.
2. In the VBE window, locate the window called Project. The Project window (also known as the Project Explorer window) contains a list of all workbooks and add-ins that are currently open. Each project is arranged as a tree and can be expanded (to show more information) or contracted (to show less information).
3. Select the project that corresponds to the workbook in which you recorded the macro. If you haven't saved the workbook, the project is probably called VBAProject (Book1).
4. Click the plus sign (+) to the left of the folder named Modules. The tree expands to show Module1, which is the only module in the project.
5. Double-click Module1. The VBA code in that module is displayed in a Code window.
6. You can now edit whatever you wish to.

e. Saving a workbook that contains Macros

If you store one or more macros in a workbook, the file must be saved with "macros enabled." In other words, the file must be saved with an XLSM extension rather than the normal XLSX extension.

3. DIFFERENT TYPES OF PROCEDURES

a. Sub Procedure

This is the most commonly used procedure that a recorded and edited macro typically uses.

It executes code line by line in order, carrying out a series of actions and/or calculations. The typical look for this type of procedure is:

```
Sub NameOfProcedure([Arguments])
      1st line of executed code 'Comments
      2nd line of executed code 'Comments
      ..........
End Sub
```

The 'Arguments' element is optional which can be **explicit** or **implicit**. This allows values and /or references to be passed into the calling procedure and handled as a variable.

When recording a macro, no arguments are used and the parenthesis for the named procedure remains empty.

If you create a procedure intended as a macro in Excel, users must not specify any arguments.

Sub procedures can be recursive meaning that branching to another procedure is permitted which then returns back to the main calling procedure.

Calling another procedure can include the **Call** statement followed by the name of the procedure with optional arguments. If arguments are used, users must use parenthesis around the argument list.

```
 'Procedure to be called
Sub MyMessage(strText As String)
      MsgBox strText
End Sub
```

Correct

```
 'Test the calling procedure

Sub TestMessage()
      Call MyMessage("It worked!")
End Sub
```

Incorrect - must use the parenthesis

```
 'Test the calling procedure

Sub TestMessage()
      Call MyMessage "Did it work?"
End Sub
```

Correct (alternative) - No Call keyword used & no parenthesis therefore required.

```
 'Test the calling procedure

Sub TestMessage()
      MyMessage "It worked!"
End Sub
```

A procedure can be prematurely terminated, placed before the '***End Sub***' statement by using the '***Exit Sub***' statement.

```
 'This procedure will terminate after part A and never run part B.

Sub TerminateNow()
      Code part A here...
      Exit Sub
      Code part B here....
End Sub
```

b. Sub Function Procedure

The main difference between a **Sub** and **Function** procedure is that a **Function** procedure carries out a procedure and will return an answer whereas a **Sub** procedure carries out the procedure without an answer.

A simple analogy of a **Function** procedure compared to that of a **Sub** procedure could be illustrated using two example features of Excel:

- *File*, *Save* is an action and does not return the answer – *Sub Procedure*.
- The *Sum* function calculates the range(s) and returns the answer – *Function Procedure*.

The signature for this type of procedure is:

```
 Function NameOfProdedure([Arguments]) [As Type]

     Code is executed here
     NameOfProcedure = Answer of the above code executed
 End Function
```

The **Arguments** element is optional which can be **explicit** or **implicit**. This allows values and /or references to be passed into the calling procedure and handled as a variable.

The optional **Type** attribute can be used to make the function explicit. Without a type declared, the function is implicit (*As Variant*).

The last line before the **End Function** signature uses the name of the procedure to return the expression (*or answer*) of the function.

Users cannot define a function inside another function, sub or even property procedures.

This type of procedure can be called in a module by a **Sub** procedure or executed as a user defined function on a worksheet in Excel.

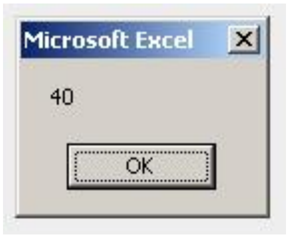
A procedure can be prematurely terminated, placed before the ***End Function*** statement by using the **Exit Function** statement. This acts and responds in the same way as described in the previous section (*Sub Procedures*).

An example of a Function procedure:

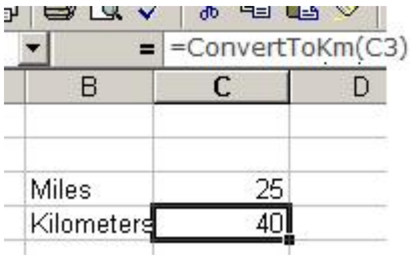
```
'This function calculates the distance of miles into kilometres.

Function ConvertToKm(dblMiles As Double) As Double
      ConvertToKm = dblMiles * 1.6
End Function
```

A **Sub** procedure that uses of the above function:

```
'Using the above function that must use parenthesis.

Sub CarDistance
      MsgBox ConvertToKm(25)
End Sub
```



In Excel, this function can also be used (known as a **User Defined Function - UDF**)


```
=ConvertToKm(C3)
```

	B	C	D
	Miles	25	
	Kilometers	40	

4. THE CONCEPT OF VARIABLES

Variables are like mail boxes in the post office. The contents of the variables change every now and then, just like the mail boxes. In Excel VBA, variables are areas allocated by the computer memory to hold data. Like the mail boxes, each variable must be given a name. To name a variable in Excel VBA, you have to follow a set of rules, as follows:

a. Variable Names

The following are the rules when naming the variables in VBA :

- It must be less than 255 characters
- No spacing is allowed
- It must not begin with a number
- Period is not permitted

Examples of valid and invalid variable names :

Valid Name	Invalid Name
My_Car	My.Car
thisYear	1NewBoy
Long_Name_Can_beUSE	He&HisFather (& is not acceptable)
Group88	Student ID (Space not allowed)

b. Declaring Variables

In VBA, we need to declare the variables before using them by assigning names and data types. There are many VBA data types, which can be grossly divided into two types, namely the numeric data types and the non-numeric data types.

Numeric Data Types

Numeric data types are types of data that consist of numbers, which can be computed mathematically with various standard arithmetic operators such as addition, subtraction, multiplication, division and more. In VBA, the numeric data are divided into 7 types, which are summarized below.

Type Storage	Range of Values
Byte 1 byte	0 to 255
Integer 2 bytes	-32,768 to 32,767
Long 4 bytes	-2,147,483,648 to 2,147,483,648
Single 4 bytes	-3.402823E+38 to -1.401298E-45 for negative values 1.401298E-45 to 3.402823E+38 for positive values.
Double 8 bytes	-1.79769313486232e+308 to -4.94065645841247E-324 for negative values 4.94065645841247E-324 to 1.79769313486232e+308 for positive values.
Currency 8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal 12 bytes	+/- 79,228,162,514,264,337,593,543,950,335 if no decimal is use +/- 7.9228162514264337593543950335 (28 decimal places).

Non-numeric Data Types

Nonnumeric data types are summarized below:

Data Type	Storage	Range
String(fixed length)	Length of string	1 to 65,400 characters
String(variable length)	Length + 10 bytes	0 to 2 billion characters
Date	8 bytes	January 1, 100 to December 31, 9999
Boolean	2 bytes	True or False
Object	4 bytes	Any embedded object
Variant(numeric)	16 bytes	Any value as large as Double
Variant(text)	Length+22 bytes	Same as variable-length string

c. Declaring a Variable

Declaring a variable allows you to state the names of the variables you are going to use and also identify what type of data the variable is going to contain.

For example, if `Result = 10`, then the variable `Result` could be declared as being an **Integer**.

d. Explicit Declaration

Explicit Declaration is when you declare your variables to be a specific data type.

Note that if you do not specify a data type, the **Variant** data type is assigned by default.

The Declaration Statement is written as follows:

```
Dim Result As Integer

Dim MyName As String

Dim Sales As Currency

Dim Data
```

Example:

The following example declares the `MySheet` variable to be a **String**.

```
Sub VariableTest()
    Dim MySheet As String
    MySheet = ActiveSheet.Name
    ActiveCell.Value = MySheet
End Sub
```

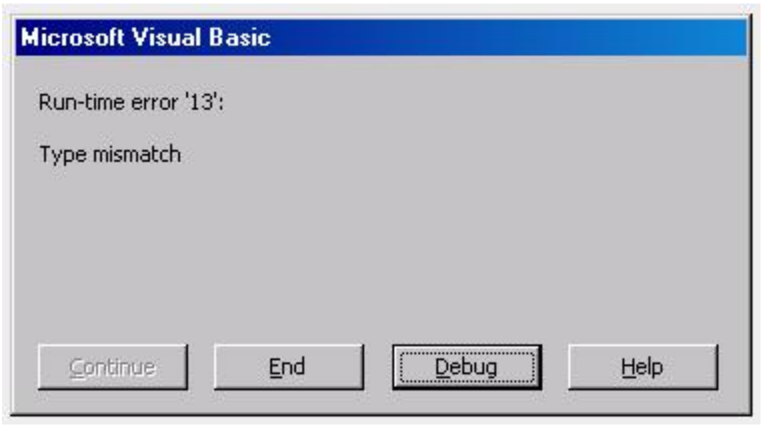
The position of the declaration statement is important as it determines whether the variable is available for use throughout the module or just within the current procedure (see *Understanding Scope & Visibility* later in this section).

Another Example:

The following example declares the `MyData` variable to be an **Integer**.

```
Sub VariableTest()
    Dim MyData As Integer
    MyData = ActiveWorkbook.Name
    ActiveCell.Value = MyData
End Sub
```

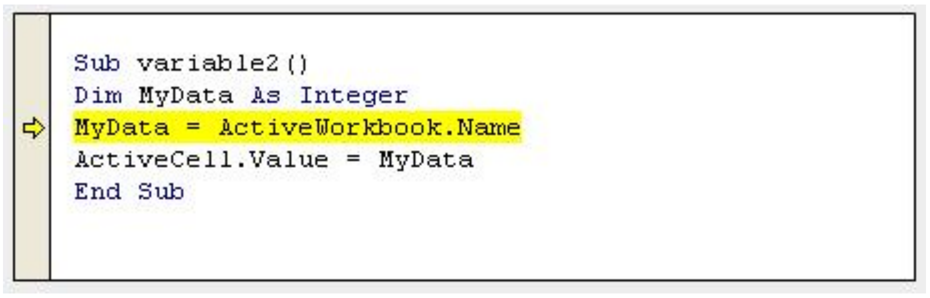
However, when you run this macro, an error will occur because `MyData = ActiveWorkbook.Name` is invalid since `ActiveWorkbook.Name` is not an **Integer** but a **String**.


Microsoft Visual Basic

Run-time error '13':

Type mismatch

When you click on the **Debug** button, it will highlight incorrect code.



```
Sub variable2()
Dim MyData As Integer
MyData = ActiveWorkbook.Name
ActiveCell.Value = MyData
End Sub
```

But if you declare `MyData as` **`String`** (i.e. text) then `MyData = ActiveWorkbook.Name` will become valid.

```
Dim MyData As String
```

e. Benefits of using Explicit Declaration

If you do not specify a data type, the **Variant** data type is assigned by default. Variant variables require more memory resources than most other variables. Therefore, your application will be more efficient if you declare the variables explicitly and with a specific data type.

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

By default, all variables not explicitly defined are **Variant** (the *largest memory allocation reserved*).

f. Constants

Constants are values that do not change. They prevent you from having to repeatedly type in large pieces of text.

The following example declares the constant **`MYFULLNAME`** to equal `"Ben Beitler"`. Therefore, wherever **`MYFULLNAME`** has been used, the value that will be returned will be`"Ben Beitler"`.

```
Sub ConstantTest()
    Const MYFULLNAME As String = "Ben Beitler"
    ActiveCell.Value = MYFULLNAME
    ActiveCell.EntireColumn.AutoFit
End Sub
```

(**Note**: When using a constant, the convention is normally in uppercase).

g. Implicit Declaration

As previously mentioned, if you do not declare your variables and constants, they are assigned the **Variant** data type, which takes up more resources and spelling mistakes are not checked.

A **Variant** Variable/Constant can contain any type of data.

```
Data = 10

Data = "Fred"

Data = #01/01/2010#
```

When you run the following macro, the value in the active cell will be 10.

```
Sub ImplicitTest()
    data = 10
    ActiveCell.Value = data
End Sub
```

When you run the following macro, the value in the active cell will be Fred.

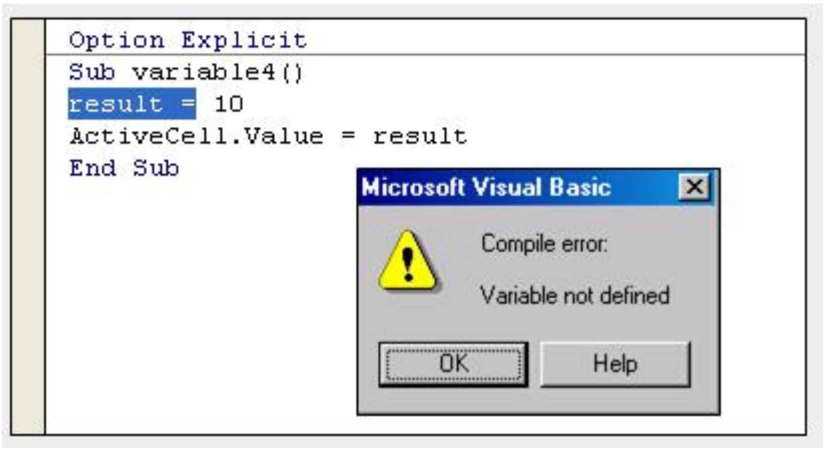
```
Sub ImplicitTest()
    data = "Fred"
    ActiveCell.Value = data
End Sub
```

This can lead to errors and memory abuse though VBA is relaxed in using variables this way - *it's just not good practice!*

h. Option Explicit (Declaration)

If you type `Option Explicit` at the top of the module sheet, you must declare all your variables and constants.

If you don't declare your variables/constants you will get the following message.



```
Option Explicit
Sub variable4()
result = 10
ActiveCell.Value = result
End Sub
```

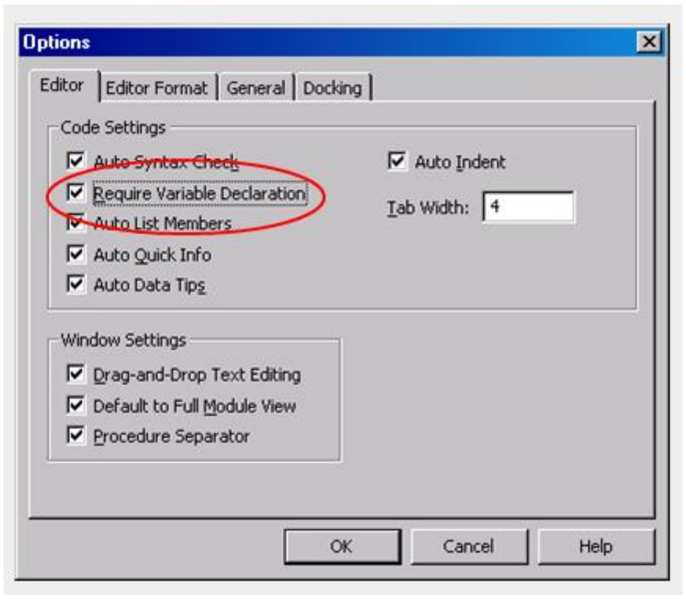
Microsoft Visual Basic

Compile error:

Variable not defined

If you wish to ensure that `Option Explicit` is always entered at the top of the module:

1. Go into the Visual Basic Editor.
2. Click on the **Tools** menu, select **Options..**.
3. Click on the **Editor** tab and select "**Require Variable Declaration**".


Options

Editor | Editor Format | General | Docking

Code Settings
- [✓] Auto Syntax Check
- [✓] Require Variable Declaration
- [✓] Auto List Members
- [✓] Auto Quick Info
- [✓] Auto Data Tips
- [✓] Auto Indent
- Tab Width: 4

Window Settings
- [✓] Drag-and-Drop Text Editing
- [✓] Default to Full Module View
- [✓] Procedure Separator

You now must always use the `Dim` keyword to declare any variable.

i. Understanding Scope & Visibility

Variables and procedures can be visible or invisible depending on the following keywords used:

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

Depending where users use the above keywords, visibility can vary too within a module, class module or user-form.

In a standard module when using the keyword `Dim` to declare a variable. If the variable is outside a procedure and in the main area of the module, this variable is automatically visible to all procedures in that module. The lifetime of this variable is governed by the calling procedure(s) and can be carried forward into the next procedure within the same module.

If a variable declared with the `Dim` keyword is within a procedure, it is visible for that procedure only and the lifetime of the variable expires when the procedure ends.

The `Dim` keyword can be used in either the module or procedure level and are both deemed as private to the module or procedure.

Instead of using the `Dim` keyword, it is better practice to use the `Private` keyword when declaring variables at the module level. Users must continue to use the `Dim` keyword within a procedure.

Using `Public` to declare a variable at the module level is potentially unsafe as it is exposed beyond this module to all other modules and user-forms. It may also provide confusion if the two variables with the same name exist across two modules. When a variable is declared `Public`, users should take caution and try and be explicit in the use of the variable.

For example:

Module A

```
Option Explicit

Public intMonth As Integer

code continues..........
```

Module B

```
Option Explicit

Private intMonth As Integer



Sub ScopeVariables()
    intMonth = 10 'This is ModuleB's variable
    ModuleA.intMonth = 10 'This is ModuleA's variable (explicit)
End Sub
code continues..........
```

Two variables with the same name and data type were declared in both module A and B. A procedure in module B calls the local variable and then explicitly calls the public variable declared in module A. Users must therefore use the notation of the name of the module followed by the period separator (**.**) and its variable.

`Public` and **`Private`** keywords can also be applied to a procedure. By default, in a standard module, all procedures are **`Public`** unless explicitly defined as **`Private`**.

It is good practice to apply either **`Public`** or **`Private`** for each procedure as later releases of Visual Basic may insist on this convention.

If a procedure is **`Private`**, it can only be used within the module it resides. This is particularly designed for internal procedures being called and then discarded as part of a branching routine (*nested procedures*).

If users mark a procedure as **`Private`**, it cannot be seen in the macros dialog box in Excel.

j. Static Variables

Using the **`Static`** keyword allows users to declare variables that retain the value from the previous calling procedure.

Example using `Dim`:

```
'Standard variable (non-static).
Sub TotalValue()
    Dim intValue As Integer
    intValue = 10
    Msgbox intValue
End Sub
```

Example using `Static`:

```
'Standard variable (non-static).
Sub TotalValue()
    Static intValue As Integer
    intValue = 10
    Msgbox intValue
End Sub
```

Running the first example will simply display the value 10 and the variable's value will be lost when the procedure ends.

Running the second example will display the value 10 but it will retain the variable and its value in memory so that when running it once more, the value displayed now equals 20 and so on until the file is either closed or the reset command is executed.

`Static` can only be used in a procedure and is therefore private.

Do not confuse `Static` with `Const` (*constant*).

Use the `Const` keyword to fix a value for lifecycle of the module or procedure. Users will not be able to modify the value at run time as with conventional variables.

Example:

Public

```
'Vat Rate is currently fixed at 17.5%
Public Const VATRATE As Single = 0.175
```

or **Private**

```
'Vat Rate is currently fixed at 17.5%
Const VATRATE As Single = 0.175
```

Using the constant

```
Sub GrossTotal()
    Dim dblNet As Double
    dblNet = 100
    MsgBox Round(dblNet * (1 + VATRATE), 2)
End Sub
```

It is acceptable to use uppercase convention for constants.

`Const` keyword can be public or private (*private by default*) declared at the module and private only at the procedure level.

User forms, which allow users to design custom form interfaces, also have scope issues using `Private` and `Public`

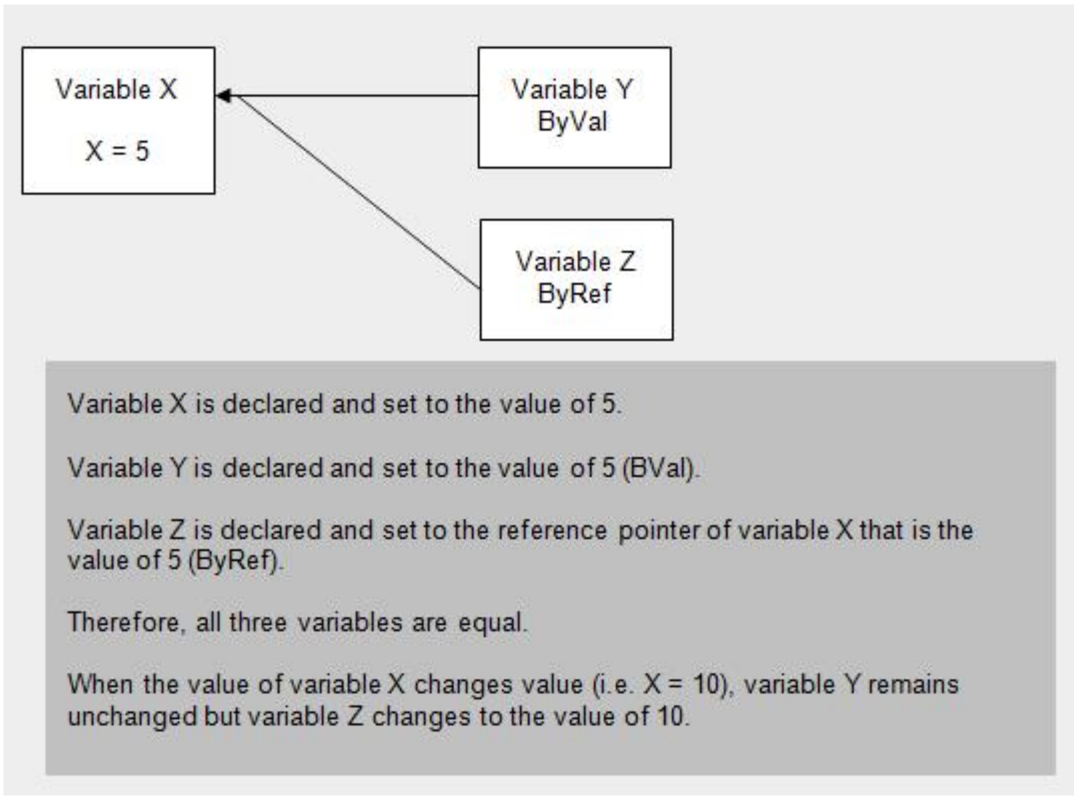
By default, any control's event that is drawn onto a form will be private as the form should be the only object able to call the procedure.

Other event driven procedures, which can be found in a worksheet or a workbook, will also be private by default.

k. ByVal versus ByRef

Passing arguments is a procedure can be either by value (ByVal) or by reference (ByRef). Both keywords precede the argument name and data type and if omitted isByRef by default.

When passing an argument by value, the value is passed into a calling procedure and then lost when the procedure ends or reverts back to the original value as it returns to the original procedure.



Variable X is declared and set to the value of 5.

Variable Y is declared and set to the value of 5 (BVal).

Variable Z is declared and set to the reference pointer of variable X that is the value of 5 (ByRef).

Therefore, all three variables are equal.

When the value of variable X changes value (i.e. X = 10), variable Y remains unchanged but variable Z changes to the value of 10.

ByVal *example:*

```
Sub CustomSub(ByVal AddTo As Integer)
    AddTo = AddTo + 5
    MsgBox AddTo
End Sub

Sub TestVariable()
    Dim x As Integer
    x = 5
    Call CustomSub(x)
    MsgBox x
End Sub
```

The procedure **TestVariable** starts by seting *x = 5*. **CustomSub** procedure is called passing the variable's value of *x* and incremented by 5. The first message box seen is via the **CustomSub** procedure (*shows the value 10*). The second message box is via the **TestVariable** procedure which follows as it returns the focus (shows the value 5). Therefore the **ByVal AddTo** variable stored is lost as it is passed back into the original call procedure (**TestVariable**) resulting in *x* being equal to 5 again.

ByRef *example:*

```
Sub CustomSub(ByRef AddTo As Integer)
    AddTo = AddTo + 5
    MsgBox AddTo
```

```
End Sub

Sub TestVariable()
    Dim x As Integer
    x = 5
    Call CustomSub(x)
    MsgBox x
End Sub
```

The procedure **TestVariable** starts by seting *x = 5*. **CustomSub** procedure is called passing the variable's value of *x* and incremented by 5. The first message box seen is via the **CustomSub** procedure (shows the value 10). The second message box is via the **TestVariable** procedure which follows as it returns the focus (shows the value 10 again). Therefore the **ByRef AddTo** variable stored is not lost as it is passed back into the original call procedure (**TestVariable**) resulting in *x* now being equal to 10.

5. PERFORMING ARITHMETIC OPERATIONS IN VBA

In order to compute input from the user and to generate results in Excel VBA, we can use various mathematical operators. In Excel VBA, except for + and -, the symbols for the operators are different from normal mathematical operators, as shown below.

Operator	**Mathematical function**	**Example**
+	Addition	1+3=4
^	Exponential	2^4=16
*	Multiplication	4*3=12
/	Division	12/4=3
Mod	Modulus (return the remainder from an integer division)	15 Mod 4=3
\	Integer Division (discards the decimal places)	19\4=4
+ or &	String concatenation	"Visual"&"Basic"="Visual Basic"

6. USING MESSAGE BOX AND INPUT BOX

There are many built-in functions available in Excel VBA which we can use to streamline our VBA programs. Among them, *message box* and *input box* are most commonly used. These two functions are useful because they make the Excel VBA macro programs more interactive. The input box allows the user to enter the data while the message box displays output to the user.

a. The MsgBox ( ) Function

The objective of the MsgBox function is to produce a pop-up message box and prompt the user to click on a command button before he or she can continue. The code for the message box is as follows:

```
yourMsg=MsgBox(Prompt, Style Value, Title)
```

- The first argument, Prompt, displays the message in the message box.
- The Style Value determines what type of command button that will appear in the message box. The table below lists the command button that can be displayed.
- The Title argument displays the title of the message box.

Style Value	Named Constant	Button Displayed
0	vbOkOnly	Ok button
1	vbOkCancel	Ok & Cancel buttons
2	vbAbortRetryIgnore	Abort, Retry and Ignore buttons
3	vbYesNoCancel	Yes, No and Cancel buttons
4	vbYesNo	Yes and No buttons
5	vbRetryCancel	Retry and Cancel buttons

We can use the named constant in place of integers for the second argument to make the programs more readable. In fact, VBA will automatically show a list of named constants where you can select one of them. For example, yourMsg=MsgBox("Click OK to Proceed", 1, "Startup Menu") and yourMsg=Msg("Click OK to Proceed". vbOkCancel,"Startup Menu") are the same. yourMsg is a variable that holds values that are returned by the MsgBox ( ) function.

You can also add an icon by using one of the following style.

Style Value	Named Constant	Icon Displayed
16	vbCritical	Critical icon
32	vbQuestion	Question icon
48	vbExclamation	Exclamation icon
64	vbInformation	Information icon

The two name constants can be joined together using the "+" sign.

The values returned by the MsgBox are determined by the type of buttons being clicked by the users. It has to be declared as Integer data type in the procedure or in the general declaration section. The table below shows the values, the corresponding named constants and the buttons.

Style Value	Named Constant	Button Pressed
1	vbOk	Ok button
2	vbCancel	Cancel button
3	vbAbort	Abort button
4	vbRetry	Retry button
5	vbIgnore	Ignore button
6	vbYes	Yes button
7	vbNo	No button

b. The InputBox( ) Function

An InputBox( ) is a function that displays an input box where the user can enter a value or a message in the form of text. The format is

myMessage=InputBox(Prompt, Title, default_text, x-position, y-position)

myMessage is a variant data type but typically it is declared as a string, which accepts the message input by the users. The arguments are explained as follows:

- Prompt - The message displayed in the inputbox.
- Title - The title of the Input Box.
- default-text - The default text that appears in the input field where users can use it as his intended input or he may change it to another message.
- x-position and y-position - the position or the coordinates of the input box.

7. USING IF….THEN….ELSE IN EXCEL VBA

If-Then is VBA's most important control structure.

The If-Then structure has this basic syntax:

```
If condition Then statements [Else elsestatements]
```

Use the If-Then structure when you want to execute one or more statements conditionally. The optional Else clause, if included, lets you execute one or more statements if the condition you're testing is not true.

a. Conditional Operators

To control the VBA program flow, we can use various conditional operators. Basically, they resemble mathematical operators. Conditional operators are very powerful tools which let the VBA program compare data values and then decide what action to take. For example, it can decide whether to execute or terminate a program. These operators are shown in the table below.

Operator	**Meaning**
=	Equal to
>	More than
<	Less Than
>=	More than and equal
<=	Less than and equal
<>	Not Equal to

You can also compare strings with the above operators. However, there are certain rules to follows: Upper case letters are lesser than lowercase letters, "A"<"B"<"C"<"D".......<"Z" and numbers are lesser than letters.

b. Logical Operators

In addition to conditional operators, there are a few logical operators that offer added power to the VBA programs :

- And : All sides must be true
- Or : At least one side must be true
- Xor : One side must be true but not both
- Not : Negates truth

c. If-Then

The following routine demonstrates the If-Then structure without the optional Else clause:

```
Sub GreetMe()
        If Time < 0.5 Then MsgBox "Good Morning"
End Sub
```

The GreetMe procedure uses VBA's Time function to get the system time.

If the current system time is less than .5 (in other words, before noon), the routine displays a friendly greeting. If Time is greater than or equal to .5, the routine ends and nothing happens.

To display a different greeting if Time is greater than or equal to .5, add another If-Then statement after the first one:

```
Sub GreetMe2()
  If Time < 0.5 Then MsgBox "Good Morning"
  If Time >= 0.5 Then MsgBox "Good Afternoon"
End Sub
```

d. If-Then-Else example

Another approach to the preceding problem uses the Else clause. Here's the same routine recoded to use the If-Then-Else structure:

```
Sub GreetMe3()
  If Time < 0.5 Then MsgBox "Good Morning" Else MsgBox "Good Afternoon"
End Sub
```

In the preceding examples, the If-Then-Else statement are actually a single statement. VBA provides a slightly different way of coding If-Then-Else constructs that use an End-If statement. Therefore, the GreetMe procedure can be rewritten as:

```
Sub GreetMe4()
  If Time < 0.5 Then
    MsgBox "Good Morning"
  Else
    MsgBox "Good Afternoon"
  End If
End Sub
```

In fact, you can insert any number of statements under the If part, and any number of statements under the Else part. This syntax because is easier to read and makes the statements shorter.

What if you need to expand the GreetMe routine to handle three conditions: morning, afternoon, and evening? You have two options: Use three If-Then statements or use a nested If-Then-Else structure. Nesting means placing an If-Then-Else structure within another If-Then-Else structure. The first approach, the three statements, is simplest:

```
Sub GreetMe5()
  Dim Msg As String
  If Time < 0.5 Then Msg = "Morning"
  If Time >= 0.5 And Time < 0.75 Then Msg = "Afternoon"
  If Time >= 0.75 Then Msg = "Evening"
  MsgBox "Good " & Msg
End Sub
```

The Msg variable gets a different text value, depending on the time of day. The final MsgBox statement displays the greeting: Good Morning, Good Afternoon, or Good Evening.

The following routine performs the same action but uses an If-Then-End If structure:

```
Sub GreetMe6()
  Dim Msg As String
  If Time < 0.5 Then
    Msg = "Morning"
  End If
  If Time >= 0.5 And Time < 0.75 Then
    Msg = "Afternoon"
  End If
  If Time >= 0.75 Then
    Msg = "Evening"
  End If
  MsgBox "Good " & Msg
End Sub
```

e. ElseIf example

In the previous examples, every statement in the routine is executed — even in the morning. A more efficient structure would exit the routine as soon as a condition is found to be true. In the morning, for example, the procedure should display the Good Morning message and then exit — without evaluating the other superfluous conditions.

With a tiny routine like this, you don't have to worry about execution speed.

But for larger applications in which speed is important, you should know about another syntax for the If-Then structure. The ElseIf syntax follows:

```
If condition Then
  [statements]
[ElseIf condition-n Then
  [elseifstatements]]
[Else
  [elsestatements]]
End If
```

Here's how you can rewrite the GreetMe routine by using this syntax:

```
Sub GreetMe7()
  Dim Msg As String
  If Time < 0.5 Then
    Msg = "Morning"
  ElseIf Time >= 0.5 And Time < 0.75 Then
    Msg = "Afternoon"
  Else
    Msg = "Evening"
  End If
  MsgBox "Good " & Msg
End Sub
```

When a condition is true, VBA executes the conditional statements and the If structure ends. In other words, VBA doesn't waste time evaluating the extraneous conditions, which makes this procedure a bit more efficient than the previous examples. The trade-off is that the code is more difficult to understand.

f. Another If-Then example

Here's another example that uses the simple form of the If-Then structure.

This procedure prompts the user for a quantity and then displays the appropriate discount, based on the quantity the user enters:

```
Sub ShowDiscount()
  Dim Quantity As Integer
  Dim Discount As Double
  Quantity = InputBox("Enter Quantity:")
  If Quantity > 0 Then Discount = 0.1
  If Quantity >= 25 Then Discount = 0.15
  If Quantity >= 50 Then Discount = 0.2
  If Quantity >= 75 Then Discount = 0.25
  MsgBox "Discount: " & Discount
End Sub
```

Notice that each If-Then statement in this routine is executed, and the value for Discount can change as the statements are executed. However, the routine ultimately displays the correct value for Discount because I put the If-Then statements in order of ascending Discount values.

The following procedure performs the same tasks by using the alternative ElseIf syntax. In this case, the routine ends immediately after executing the statements for a true condition.

```
Sub ShowDiscount2()
  Dim Quantity As Integer
  Dim Discount As Double
  Quantity = InputBox("Enter Quantity: ")
  If Quantity > 0 And Quantity < 25 Then
    Discount = 0.1
  ElseIf Quantity >= 25 And Quantity < 50 Then
    Discount = 0.15
  ElseIf Quantity >= 50 And Quantity < 75 Then
    Discount = 0.2
  ElseIf Quantity >= 75 Then
    Discount = 0.25
  End If
End Sub
```

8. THE SELECT CASE STRUCTURE

The Select Case structure is useful for decisions involving three or more options (although it also works with two options, providing an alternative to the If-Then-Else structure). The syntax for the Select Case structure follows:

```
Select Case testexpression
[Case expressionlist-n [statements-n]] . . .
[Case Else [elsestatements]]
End Select
```

Don’t be scared off by this official syntax. Using the Select Case structure is quite easy.

a. A Select Case example

The following example shows how to use the Select Case structure. This also shows another way to code the examples presented in the previous section:

```
Sub ShowDiscount3()
  Dim Quantity As Integer
  Dim Discount As Double
  Quantity = InputBox(“Enter Quantity: “)
  Select Case Quantity
    Case 0 To 24
      Discount = 0.1
    Case 25 To 49
      Discount = 0.15
    Case 50 To 74
      Discount = 0.2
    Case Is >= 75
      Discount = 0.25
  End Select
  MsgBox “Discount: “ & Discount
End Sub
```

In this example, the Quantity variable is being evaluated. The routine is checking for four different cases (0–24, 25–49, 50–74, and 75 or greater).

Any number of statements can follow each Case statement, and they all are executed if the case is true. If you use only one statement, as in this example, you can put the statement on the same line as the Case keyword, preceded by a colon — the VBA statement separator character. In my opinion, this makes the code more compact and a bit clearer.

Here's how the routine looks, using this format:

```
Sub ShowDiscount4 ()
  Dim Quantity As Integer
  Dim Discount As Double
  Quantity = InputBox("Enter Quantity: ")
  Select Case Quantity
    Case 0 To 24: Discount = 0.1
    Case 25 To 49: Discount = 0.15
    Case 50 To 74: Discount = 0.2
    Case Is >= 75: Discount = 0.25
  End Select
  MsgBox "Discount: " & Discount
End Sub
```

When VBA executes a Select Case structure, the structure is exited as soon as VBA finds a true case and executes the statements for that case.

b. A nested Select Case example

As demonstrated in the following example, you can nest Select Case structures. This routine examines the active cell and displays a message describing the cell's contents. Notice that the procedure has three Select Case structures and each has its own End Select statement.

```
Sub CheckCell()
  Dim Msg As String
  Select Case IsEmpty(ActiveCell)
    Case True
      Msg = "is blank."
    Case Else
      Select Case ActiveCell.HasFormula
        Case True
          Msg = "has a formula"
        Case False
          Select Case IsNumeric(ActiveCell)
            Case True
              Msg = "has a number"
            Case Else
              Msg = "has text"
          End Select
      End Select
  End Select
  MsgBox "Cell " & ActiveCell.Address & " " & Msg
End Sub
```

The logic goes something like this:

1. Find out whether the cell is empty.

2. If it's not empty, see whether it contains a formula.

3. If there's no formula, find out whether it contains a numeric value or text.

When the routine ends, the Msg variable contains a string that describes the cell's contents. You can nest Select Case structures as deeply as you need to, but make sure that each Select Case statement has a corresponding End Select statement.

9. GOTO STATEMENT

A GoTo statement offers the most straightforward means for changing a program's flow. The GoTo statement simply transfers program execution to a new statement, which is preceded by a label.

Your VBA routines can contain as many labels as you like. A label is just a text string followed by a colon.

The following procedure shows how a GoTo statement works:

```
Sub GoToDemo()
  UserName = InputBox("Enter Your Name: ")
  If UserName <> "Bill Gates" Then GoTo WrongName
  MsgBox ("Welcome Bill...")
  ' ...[More code here] ...
  Exit Sub
  WrongName:
    MsgBox "Sorry. Only Bill Gates can run this."
End Sub
```

The procedure uses the InputBox function to get the user's name. If the user enters a name other than Bill Gates, the program flow jumps to the WrongName label, displays an apologetic message, and the procedure ends.

On the other hand, if Mr. Gates runs this procedure and uses his real name, the procedure displays a welcome message and then executes some additional code (not shown in the example). Notice that the Exit Sub statement ends the procedure before the second MsgBox function has a chance to work.

This simple routine works, but VBA provides several better (and more structured) alternatives than GoTo. In general, you should use GoTo only when you have no other way to perform an action. In real life, the only time you must use a GoTo statement is for trapping errors. For example :

```
 On Error GoTo ErrHandler:

N = 1 / 0

' code that is skipped if an error occurs

Label1:

' more code to execute

Exit Sub

ErrHandler:

' go back to the line at Label1:
```

```
Resume Label1:
```

CONTENTS

1. FOR-NEXT LOOPS

The simplest type of loop is a For-Next loop. Here's the syntax for this structure:

```
For counter = start To end [Step stepval]
[statements]
[Exit For]
[statements]
Next [counter]
```

The looping is controlled by a counter variable, which starts at one value and stops at another value. The statements between the For statement and the Next statement are the statements that get repeated in the loop. To see how this works, keep reading.

a. A For-Next example

The following example shows a For-Next loop that doesn't use the optional Step value or the optional Exit For statement. This routine loops 20 times and uses the VBA Rnd function to enter a random number into 20 cells, beginning with the active cell:

```
Sub FillRange()
  Dim Count As Long
  For Count = 0 To 19
    ActiveCell.Offset(Count, 0) = Rnd
  Next Count
End Sub
```

In this example, Count (the loop counter variable) starts with a value of 0 and increases by 1 each time through the loop. Because I didn't specify a Step value, VBA uses the default value (1). The Offset method uses the value of Count as an argument. The first time through the loop, Count is 0 and the procedure enters a number into the active cell offset by zero rows. The second time through (Count = 1), the procedure enters a number into the active cell offset by one row, and so on.

Because the loop counter is a normal variable, you can write code to change its value within the block of code between the For and the Next statements.

This, however, is a very bad practice. Changing the counter within the loop can have unpredictable results. Take special precautions to ensure that your code does not directly change the value of the loop counter.

b. A For-Next example with a Step

You can use a Step value to skip some values in a For-Next loop. Here's the same procedure as in the preceding section, rewritten to insert random numbers into every other cell:

```
Sub FillRange2()
  Dim Count As Long
  For Count = 0 To 19 Step 2
    ActiveCell.Offset(Count, 0) = Rnd
  Next Count
End Sub
```

This time, Count starts out as 0 and then takes on a value of 2, 4, 6, and so on. The final Count value is 18. The Step value determines how the counter is incremented. Notice that the upper loop value (19) is not used because the highest value of Count after 18 would be 20, and 20 is larger than 19.

c. A For-Next example with an Exit For statement

A For-Next loop can also include one or more Exit For statements within the loop. When VBA encounters this statement, the loop terminates immediately. This routine identifies which of the active worksheet's cells in column A has the largest value:

```
Sub ExitForDemo()
  Dim MaxVal As Double
  Dim Row As Long
  MaxVal = WorksheetFunction.Max(Range("A:A"))
  For Row = 1 To Rows.Count
    If Range("A1").Offset(Row-1, 0).Value = MaxVal Then
      Range("A1").Offset(Row-1, 0).Activate
      MsgBox "Max value is in Row " & Row
      Exit For
    End If
  Next Row
End Sub
```

The routine calculates the maximum value in the column by using Excel's MAX function and assigns the result to the MaxVal variable. The For-Next loop then checks each cell in the column. If the cell being checked is equal to MaxVal, the routine doesn't need to continue looping (its job is finished), so the Exit For statement terminates the loop.

Before terminating the loop, the procedure activates the cell with the maximum value and informs the user of its location.

Notice that I use Rows.Count in the For statement. The count property of the Rows object returns the number of rows in the worksheet. Therefore, you can use this procedure with earlier versions of Excel (which have fewer rows).

d. A nested For-Next example

So far, all examples use relatively simple loops. However, you can have any number of statements in the loop and nest For-Next loops inside other For-Next loops.

The following example uses a nested For-Next loop to insert random numbers into a 12-row-x-5-column range of cells. Notice that the routine executes the inner loop (the loop with the Row counter) once for each iteration of the outer loop (the loop with the Col counter). In other words, the routine executes the Cells(Row, Col) = Rnd statement 60 times.

```
Sub FillRange2()
  Dim Col As Long
  Dim Row As Long
  For Col = 1 To 5
    For Row = 1 To 12
      Cells(Row, Col) = Rnd
    Next Row
  Next Col
End Sub
```

The next example uses nested For-Next loops to initialize a three-dimensional array with the value 100. This routine executes the statement in the middle of all the loops (the assignment statement) 1,000 times (10 * 10 * 10), each time with a different combination of values for i, j, and k:

```
Sub NestedLoops()
  Dim MyArray(10, 10, 10)
  Dim i As Integer
  Dim j As Integer
  Dim k As Integer
  For i = 1 To 10
    For j = 1 To 10
      For k = 1 To 10
        MyArray(i, j, k) = 100
      Next k
    Next j
  Next i
  'Other statements go here
End Sub
```

2. FOR EACH - LOOPING THROUGH A COLLECTION

VBA supports yet another type of looping — looping through each object in a collection of objects. Recall that a collection consists of a number of objects of the same type. For example, Excel has a collection of all open workbooks (the Workbooks collection), and each workbook has a collection of worksheets (the Worksheets collection).

When you need to loop through each object in a collection, use the For Each-Next structure. The syntax is:

```
For Each element In collection
[statements]
[Exit For]
[statements]
Next [element]
```

The following example loops through each worksheet in the active workbook and deletes the first row of each worksheet:

```
Sub DeleteRow1()
  Dim WkSht As Worksheet
  For Each WkSht In ActiveWorkbook.Worksheets
    WkSht.Rows(1).Delete
  Next WkSht
End Sub
```

In this example, the variable WkSht is an object variable that represents each worksheet in the workbook. Nothing is special about the variable name WkSht — you can use any variable name that you like.

The example that follows loops through the cells in a range, and checks each cell. The code switches the sign of the values (negative values are made positive; positive values are made negative). It does this by multiplying each value times –1. Note that I used an If-Then construct, along with the VBA IsNumeric function, to ensure that the cell contains a numeric value:

```
Sub ChangeSign()
  Dim Cell As Range
  For Each Cell In Range(“A1:E50”)
   If IsNumeric(Cell.Value) Then
     Cell.Value = Cell.Value * -1
   End If
  Next Cell
End Sub
```

The preceding code sample has a problem: It changes any formulas in the range it loops through to values, zapping your formulas. That's probably not what you want. Here's another version of the Sub that skips formula cells. It checks whether the cell has a formula by accessing the HasFormula property:

```
Sub ChangeSign2()
  Dim Cell As Range
  For Each Cell In Range("A1:E50")
    If Not Cell.HasFormula Then
      If IsNumeric(Cell.Value) Then
        Cell.Value = Cell.Value * -1
      End If
    End If
  Next Cell
End Sub
```

And here's one more For Each-Next example. This procedure loops through each chart on Sheet1 (that is, each member of the ChartObjects collection) and changes each chart to a line chart. In this example, Cht is a variable that represents each ChartObject. If Sheet1 has no ChartObjects, nothing happens.

```
Sub ChangeCharts()
  Dim Cht As ChartObject
  For Each Cht In Sheets("Sheet1").ChartObjects
    Cht.Chart.ChartType = xlLine
  Next Cht
End Sub
```

To write a procedure like ChangeCharts, you need to know something about the object model for charts. You can get that information by recording a macro to find out which objects are involved and then checking the Help system for details.

Excel 2007 users are out of luck here: The macro recorder in Excel 2007 does not record all chart changes you make.

3. DO-WHILE LOOP

VBA supports another type of looping structure known as a Do-While loop. Unlike a For-Next loop, a Do-While loop continues until a specified condition is met. Here's the Do-While loop syntax:

```
Do [While condition]
[statements]
[Exit Do]
[statements]
Loop
```

The following example uses a Do-While loop. This routine uses the active cell as a starting point and then travels down the column, multiplying each cell's value by 2. The loop continues until the routine encounters an empty cell.

```
Sub DoWhileDemo()
  Do While ActiveCell.Value <> Empty
    ActiveCell.Value = ActiveCell.Value * 2
    ActiveCell.Offset(1, 0).Select
  Loop
End Sub
```

Some people prefer to code a Do-While loop as a Do-Loop While loop. This example performs exactly as the previous procedure but uses a different loop syntax:

```
Sub DoLoopWhileDemo()
  Do
    ActiveCell.Value = ActiveCell.Value * 2
    ActiveCell.Offset(1, 0).Select
  Loop While ActiveCell.Value <> Empty
End Sub
```

Here's the key difference between the Do-While and Do-Loop While loops: The Do-While loop always performs its conditional test first. If the test is not true, the instructions inside the loop are never executed. The Do-Loop While loop, on the other hand, always performs its conditional test after the instructions inside the loop are executed. Thus, the loop instructions are always executed at least once, regardless of the test. This difference can sometimes have a big effect on how your program functions.

4. DO-UNTIL LOOP

The Do-Until loop structure is similar to the Do-While structure. The two structures differ in their handling of the tested condition. A program continues to execute a Do-While loop while the condition remains true. In a Do-Until loop, the program executes the loop until the condition is true.

Here's the Do-Until syntax:

```
Do [Until condition]
statements]
[Exit Do]
[statements]
Loop
```

The following example is the same one presented for the Do-While loop but recoded to use a Do-Until loop:

```
Sub DoUntilDemo()
  Do Until IsEmpty(ActiveCell.Value)
    ActiveCell.Value = ActiveCell.Value * 2
    ActiveCell.Offset(1, 0).Select
  Loop
End Sub
```

Just like with the Do-While loop, you may encounter a different form of the Do-Until loop — a Do-Loop Until loop. The following example, which has the same effect as the preceding procedure, demonstrates an alternate syntax for this type of loop:

```
Sub DoLoopUntilDemo()
  Do
    ActiveCell.Value = ActiveCell.Value * 2
    ActiveCell.Offset(1, 0).Select
  Loop Until IsEmpty(ActiveCell.Value)
End Sub
```

There is a subtle difference in how the Do-Until loop and the Do-Loop Until loop operate. In the former, the test is performed at the beginning of the loop, before anything in the body of the loop is executed. This means that it is possible that the code in the loop body will not be executed if the test condition is met. In the latter version, the condition is tested at the end of the loop.

Therefore, at a minimum, the Do-Loop Until loop always results in the body of the loop being executed once.

Another way to think about it is like this: The Do-While loop keeps looping as long as the condition is True. The Do-Until loop keeps looping as long as the condition is False.

5. HANDLING ERRORS

a. Introduction

Error handling refers to the programming practice of anticipating and coding for error conditions that may arise when your program runs. Errors in general come in three flavors:

- compiler errors such as undeclared variables that prevent your code from compiling
- user data entry error such as a user entering a negative value where only a positive number is acceptable
- run time errors, that occur when VBA cannot correctly execute a program statement.

We will concern ourselves here only with run time errors. Typical run time errors include attempting to access a non-existent worksheet or workbook, or attempting to divide by zero. The example code in this article will use the division by zero error (Error 11) when we want to deliberately raise an error.

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

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

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

b. The On Error Statement

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

```
On Error Goto 0

On Error Resume Next

On Error Goto <label>:
```

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

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

```
On Error Resume Next

N = 1 / 0    ' cause an error

If Err.Number <> 0 Then

    N = 1

End If
```

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

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

```
        On Error Goto ErrHandler:

        N = 1 / 0

        Exit Sub

    ErrHandler:

        ' error handling code

        Resume Next

    End Sub
```

c. Enabled And Active Error Handlers

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

Error Handling Blocks And On Error Goto

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

```
    On Error GoTo Err1:

    Debug.Print 1 / 0

    ' more code

Err1:

    On Error GoTo Err2:

    Debug.Print 1 / 0

    ' more code

Err2:
```

When the first error is raised, execution transfers to the line following Err1:. The error hander is still active when the second error occurs, and therefore the second error is not trapped by the On Error statement.

d. The Resume Statement

The Resume statement instructs VBA to resume execution at a specified point in the code. You can use Resume only in an error handling block; any other use will cause an error. Moreover, Resume is the only way, aside from exiting the procedure, to get out of an error handling block. Do not use the Goto statement to direct code execution out of an error handling block. Doing so will cause strange problems with the error handlers.

The Resume statement takes three syntactic forms:

```
Resume

Resume Next

Resume <label>
```

Used alone, Resume causes execution to resume at the line of code that caused the error. In this case you must ensure that your error handling block fixed the problem that caused the initial error. Otherwise, your code will enter an endless loop, jumping between the line of code that caused the error and the error handling block. The following code attempts to activate a worksheet that does not exist. This causes an error (9 - Subscript Out Of Range), and the code jumps to the error handling block which creates the sheet, correcting the problem, and resumes execution at the line of code that caused the error.

```
On Error GoTo ErrHandler:

Worksheets("NewSheet").Activate

Exit Sub



ErrHandler:

If Err.Number = 9 Then

    ' sheet does not exist, so create it

    Worksheets.Add.Name = "NewSheet"

    ' go back to the line of code that caused the problem
    Resume

End If
```

The second form of Resume is Resume Next . This causes code execution to resume at the line immediately following the line which caused the error. The following code causes an error (11 - Division By Zero) when attempting to set the value of N. The error handling block assigns 1 to the variable N, and then causes execution to resume at the statement after the statement that caused the error.

```
On Error GoTo ErrHandler:

N = 1 / 0

Debug.Print N

Exit Sub

ErrHandler:

N = 1

' go back to the line following the error

Resume Next
```

The third form of Resume is Resume <label>: . This causes code execution to resume at a line label. This allows you to skip a section of code if an error occurs. For example :

```
 On Error GoTo ErrHandler:

N = 1 / 0

' code that is skipped if an error occurs

Label1:

' more code to execute

Exit Sub

ErrHandler:

' go back to the line at Label1:

Resume Label1:
```

All forms of the Resume clear or reset the Err object.

e. Error Handling With Multiple Procedures

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

6. THE ERR OBJECT

a. Introduction

To assist you with handling errors, the Visual Basic language provides a class named Err. You don't have to declare a variable for this class. An Err object is readily available as soon as you you start working on VBA code and you can directly access its members.

b. The Error Number

As mentioned already, there are various types of errors that can occur to your program. To assist you with identifying them, the Err object is equipped with a property named Number. This property holds a specific number to most errors that can occur to your program. When your program runs and encounters a problem, it may stop and display the number of the error.

Because there are many types of errors, there are also many numbers, so much that we cannot review all of them. We can only mention some of them when we encounter them.

When a program runs, to find out what type of error occurred, you can question the Number property of the Err object to find out whether the error that has just occurred holds this or that number. To do this, you can use an If...Then conditional statement to check the number. You can then display the necessary message to the user. Here is an example:

```
Private Sub cmdCalculate_Click()

    On Error GoTo WrongValue

    Dim HourlySalary As Double, WeeklyTime As Double

    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as

    ' if the user types an invalid number

    HourlySalary = CDbl(txtHourlySalary)

    WeeklyTime = CDbl(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label

    WeeklySalary = HourlySalary * WeeklyTime
    txtWeeklySalary = FormatNumber(WeeklySalary)

    Exit Sub

WrongValue:

    If Err.Number = 13 Then

        MsgBox "You typed an invalid value"

        HourlySalary = 0

        WeeklyTime = 0

        Resume Next

    End If

End Sub
```

c. The Error Message

As mentioned already, there are many errors and therefore many numbers held by the Number property of the Err object. As a result, just knowing an error number can be vague. To further assist you with decrypting an error, the Err object provides a property named Description. This property holds a (usually short) message about the error number. This property works along with the Number property holding the message corresponding to the Number property.

To get the error description, after inquiring about the error number, you can get the equivalent Description value. Here is an example:

```
Private Sub cmdCalculate_Click()

    On Error GoTo WrongValue

    Dim HourlySalary As Double, WeeklyTime As Double

    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as

    ' if the user types an invalid number

    HourlySalary = CDbl(txtHourlySalary)

    WeeklyTime = CDbl(txtWeeklyTime)
    ' If there was an error, the flow would jump to the label

    WeeklySalary = HourlySalary * WeeklyTime

    txtWeeklySalary = FormatNumber(WeeklySalary)

    Exit Sub

WrongValue:

    If Err.Number = 13 Then

        MsgBox Err.Description

        HourlySalary = 0

        WeeklyTime = 0

        Resume Next

    End If

End Sub
```

In some cases, the error message will not be explicit enough, especially if a user simply reads it to you over the phone. The alternative is to create your own message in the language you easily understand, as we did earlier. If you want, you can also display a message that combines both the error description and your own message. Here is an example:

```
Private Sub cmdCalculate_Click()

    On Error GoTo WrongValue

    Dim HourlySalary As Double, WeeklyTime As Double

    Dim WeeklySalary As Double

    ' One of these two lines could produce an error, such as

    ' if the user types an invalid number

    HourlySalary = CDbl(txtHourlySalary)

    WeeklyTime = CDbl(txtWeeklyTime)

    ' If there was an error, the flow would jump to the label

    WeeklySalary = HourlySalary * WeeklyTime
    txtWeeklySalary = FormatNumber(WeeklySalary)

    Exit Sub

WrongValue:

    If Err.Number = 13 Then

        MsgBox Err.Description & ": The value you typed cannot be accepted."

        HourlySalary = 0

        WeeklyTime = 0

        Resume Next

    End If

End Sub
```

d. The Source of the Error

Most of the time, you will know what caused an error, since you will have created the application. The project that causes an error is known as the source of error. In some cases, you may not be able to easily identify the source of error. To assist you with this, the Err object is equipped with a property named Source.

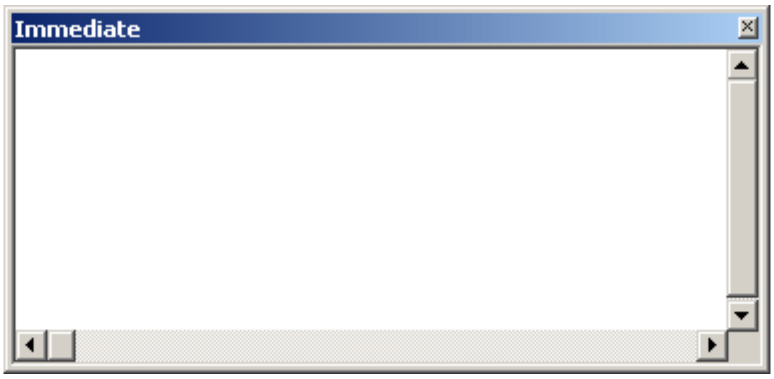
To identify the application that caused an error, you can inquire about the value of this property.

7. DEBUGGING AND THE IMMEDIATE WINDOW

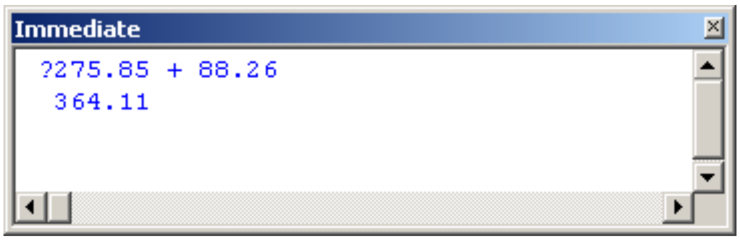
a. The Immediate Window

Debugging consists of examining and testing portions of your code or parts of your application to identify problems that may occur when somebody is using your database. Microsoft Visual Basic provides as many tools as possible to assist you with this task.

The Immediate window is an object you can use to test functions and expressions. To display the Immediate window, on the main menu of Microsoft Visual Basic, you can click View -> Immediate Window. It's a habit to keep the Immediate window in the bottom section of the Code Editor but you can move it from there by dragging its title bar:



Probably the simplest action you can perform in the Immediate window consists of testing an expression. For example, you can write an arithmetic operation and examine its result. To do this, in the Immediate window, type the question mark "?" followed by the expression and press Enter. Here is an example that tests the result of 275.85 + 88.26:

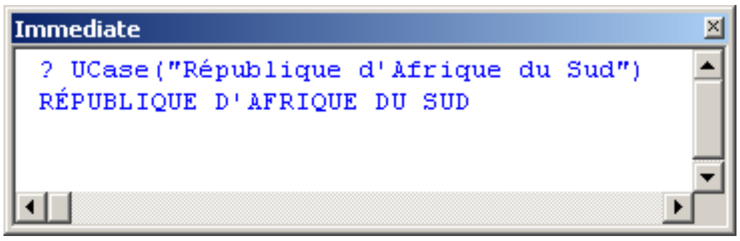


```
?275.85 + 88.26
 364.11
```

One of the most basic actions you can perform in the Immediate window consists of testing a built-in function. To do this, type ? followed by the name of the function and its arguments, if any. For example, to test the UCase$ function, in the Immediate window, you could type:

? UCase("République d'Afrique du Sud")

After typing the function and pressing Enter, the result would display in the next line:



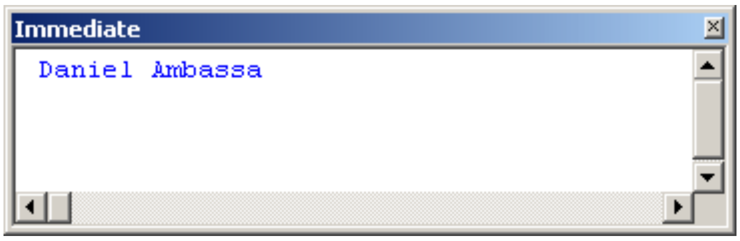
```
? UCase("République d'Afrique du Sud")
RÉPUBLIQUE D'AFRIQUE DU SUD
```

b. The Debug Object

The Immediate window is recognized in code as the Debug object. To programmatically display something, such as a string, in the Immediate window, the Debug object provides the Print method. The simplest way to use it consist of passing it a string. For example, imagine you create a button on a form, you name it cmdTestFullName and initialize it with a string. Here is an example of how you can display that string in the Immediate window:

```
Private Sub cmdTestFullName_Click()

    Dim strFullName$



    strFullName$ = "Daniel Ambassa"

    Debug.Print strFullName$

End Sub
```

When you click the button, the Immediate window would display the passed string:



```
Daniel Ambassa
```

In the same way, you can create a more elaborate expression and test its value in the Immediate window. You can also pass a value, such as a date, that can easily be converted to a string.

CONTENTS

1. EXCEL OBJECT MODEL

a. What is Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that represents concepts as "objects" that have data fields (attributes that describe the object) and associated procedures known as methods. **The attributes of the object hold its current information and the methods of the object determine what the object can do.**

An object type is called a class. A class is thus some type of object. Objects are instances of classes. All objects within a class hold the same kind of information (identical attributes) and can perform the same actions (identical methods).

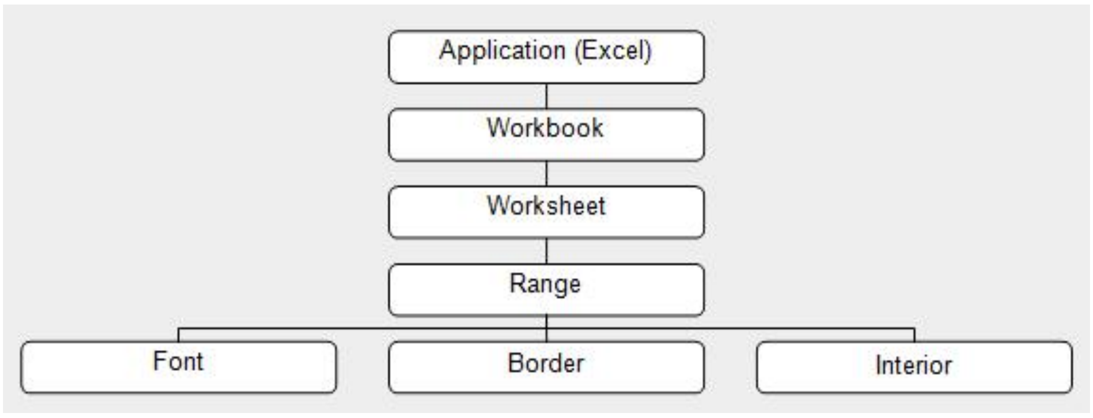
Object are used to interact with one another to design applications and computer programs.

b. Preview of Excel Object Model

Excel treats everything in a spreadsheet as objects. So, VBA lets us play with all these objects:

- Object that we are interested in: Workbooks, worksheets, cells, rows, ranges.
- There are over 200 different class (types of objects) in Excel.

Here a preview of the object hierarchy:



c. Application object

The word **Application** refers to the host (in this case Excel) and is deemed the top level object.

Use this object as the entry point (the gateway) to the Excel object model and is implicit which means that you can omit this keyword in your code as it's the default. The following two VBA commands do the same thing:

```
Application.ActiveSheet.Name = "January"
```

```
ActiveSheet.Name = "January"
```

The first example included the **Application** object keyword (as explicit) and the second one excluded (as implicit) it but produced the same result.

You only need to use this keyword if you are coding with other applications (that is not Excel) or wish to communicate to Excel from another application's environment (i.e. Microsoft Word). You will need to learn about object variables and set application objects to Excel.

The following code snippet creates an Excel object from outside of Excel (which uses VBA too) and opens a workbook called "Sales.xlsx":

```
Sub OpenExcelWorkbook()
      Dim xl As Object
      Set xl = CreateObject("Excel.Sheet")
      xl.Application.WorkBooks.Open("Sales.xlsx")
      'executed code continues...

End Sub
```

d. ActiveWorkbook and Workbooks objects

This object appears below the **Application** object along with other key objects including **Chart** and **Pivot Table** and control the tasks for any workbook from creating, opening, printing to saving and closing documents.

The singular keyword **Workbook** refers to the current or a single file you wish to control compared with the plural keyword **Workbooks** which is the collection of one or more documents you wish to control

Use the **Workbook** object referred in code as **ActiveWorkbook** to open, save, print, close and manipulate the documents attributes as required.

```
Sub WorkBookNameExample()
      MsgBox "Current workbook is " & ActiveWorkbook.Name
End Sub
```

Save a copy of the current workbook:

```
Sub SaveAsWorkBookExample1()
      ActiveWorkbook.SaveAs "VBA Workbook.xlsx"
End Sub
```

The above can also be expressed as follows:

```
Sub SaveAsWorkBookExample2()
      Workbooks(1).SaveAs "VBA Workbook.xlsx"
End Sub
```

Using the **Workbooks** keyword which is a collection of current workbooks, you can provide an index number (starting at 1 for the first document and incrementing by 1 for each open document) to execute code using the same identifiers as **ActiveWorkbook** object.

How many workbooks are currently open?

```
Sub WorkBookCount()
      MsgBox "There are currently " & Workbooks.Count & _
                                       " workbook(s) open"
End Sub
```

The **Workbooks** object doesn't have any parenthesis and an index number reference when dealing with a collection of many documents.

e. ActiveSheet and Worksheets objects

Most of the time, you will work with this object along the range object as the normal practice is worksheet management in a workbook when working with the Excel interface.

Again, the singular **Worksheet** object referred as **ActiveWorkSheet** controls the current or single worksheet objects including its name. The plural keyword **Worksheets** refers to one or more worksheets in a workbook which allows you to manipulate a collection of worksheets in one go.

Name a worksheet:

```
Sub RenameWorksheetExample1()
      ActiveWorkSheet.Name = "January"
End Sub
```

Or use

```
Sub RenameWorksheetExample2()
      WorkSheets(1).Name = "January"
End Sub
```

Assuming the first worksheet is to be renamed.

Insert a new worksheet and place it at the end of the current worksheets:

```
Sub InsertWorksheet1()
      Worksheets.Add After:=Worksheets(Worksheets.Count)
End Sub
```

Or it can shortened using the **Sheets** keyword instead:

```
Sub InsertWorksheet2()
      Sheets.Add After:=Sheets(Sheets.Count)
End Sub
```

f. 'Active' objects

Within the **Application** object you have other properties which act as shortcuts to the main objects directly below it. These include **ActiveCell**, **ActiveChart**, **ActivePrinter**, **ActiveSheet** **ActiveWindow** and **ActiveWorkbook**.

You use the above keywords as a direct implicit reference to the singular active object in the same way (as in the above already illustrated).

Remember, you can only have one active object when working in the Excel interface and therefore the VBA code is emulating the way users are conditioned to work. Even when a range of cells is selected (Selection object) only on cell is active (the white cell).

```
Sub PrinterName()
      MsgBox "Printer currently set is " & ActivePrinter
End Sub
```

2. RANGE & SELECTION OBJECTS

Range is one of the most widely used objects in Excel VBA, as it allows the manipulation of a row, column, cell or a range of cells in a spreadsheet.

When recording absolute macros, a selection of methods and properties use this object:

```
Range("A1").Select

Range("A1").FormulaR1C1 = 10
```

A generic global object known as **Selection** can be used to determine the current selection of a single or range cells.

When recording relative macros, a selection of methods and properties use this object:

```
Selection.Clear

Selection.Font.Bold = True
```

There are many properties and methods that are shared between **Range** and **Selection** objects and below are some illustrations (my choice of commonly used identifiers):

g. ADDRESS Property

Returns or sets the reference of a selection.

```
Sub AddressExample()
    MsgBox Selection.Address '$A$1 (default) - absolute
    MsgBox Selection.Address(False, True) '$A1 - column absolute
    MsgBox Selection.Address(True, False) 'A$1 - row absolute
    MsgBox Selection.Address(False, False) 'A1 - relative

End Sub
```

h. AREAS Property

Use this property to detect how many ranges (*non-adjacent*) are selected.

```
'Selects three non-adjacent ranges
Sub AreaExample()
    Range("A1:B2", E4, G10:J25").Select
    MsgBox Selection.Area.Count 'Number '3' - ranges returned
End Sub
```

The **Count** method returns the number selected as the **Areas** is a property only.

```
'Check for multiple range selection
Sub AreaExample2()
   If Selection.Areas.Count > 1 Then
       MsgBox "Cannot continue, only one range must be selected."
       Exit Sub
    End If

    [Code continues here...]
End Sub
```

Use the **Areas** property to check the state of a spreadsheet. If the system detects multiple ranges, a prompt will appear.

i. CELLS Property

This property can be used as an alternative to the absolute range property and is generally more flexible to work with, as variables are easier to pass into it.

There are two optional arguments: `Cells([row] [,column])`

Leaving the arguments empty (*no brackets*), it will detect the current selection as the active range.

Adding an argument to either row or column with a number will refer to the co-ordination of the number passed.

Adding both arguments will explicitly locate the single cell's co-ordinate.

```
'Examples of the Cells property
Sub CellsExample()
    Cells.Clear 'clears active selection
    Cells(1).Value = "This is A1 - row 1"
    Cells(, 1).Value = "This is A1 - col 1"
    Cells(1, 1).Value = "This is A1 - explicit"
    Cells(3, 3).Value = "This is C3"
    Cells(5, 3).Font.Bold = True
End Sub
```

Variables can be passed into the **Cells** property and then nested into the **Range** object as in the following example:

```
'Two InputBoxes for rows and columns
Sub CellsExample2()
    On Error GoTo handler
    Dim intRows As Integer
    Dim intCols As Integer
    intRows = CInt(InputBox("How many rows to populate?"))
    intCols = CInt(InputBox("How many columns to populate?"))
    'starts at cell A1 to the number of rows and columns passed
    Range(Cells(1, 1), Cells(intRows, intCols)).Value = "X"
    Exit Sub
handler:
    'Error code is handled here...
End Sub
```

By wrapping a range property around two cell properties, the flexibility of passing variables becomes apparent : `Range(Cells(1, 1), Cells(intRows, intCols))`

j. Column(s) and Row(s) Properties

Four properties that return the column or row number for the focused range.

The singular (***Column*** *or* ***Row***) returns the active cell's co-ordinate and the plural (***Columns*** *or* ***Rows***) can be used to count the current selections configuration.

```
Sub ColRowExample()
    MsgBox "Row " & ActiveCell.Row & _
        " : Column " & ActiveCell.Column
End Sub
```

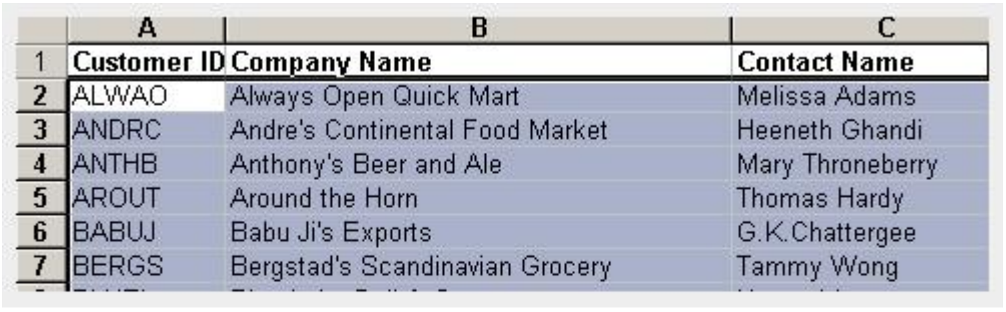
```
Sub ColsRowsCountExample()
    MsgBox Selection.Rows.Count & " rows by " _
        & Selection.Columns.Count & " columns selected"
End Sub
```

k. CURRENTREGION Property

Selects from the active cell's position all cells that are adjacent (*known as a region*) until a blank row and blank column breaks the region.

Use this statement to select a region: `Selection.CurrentRegion.Select`

To select a region of data and exclude the top row for a data list:



	A	B	C
1	Customer ID	Company Name	Contact Name
2	ALWAO	Always Open Quick Mart	Melissa Adams
3	ANDRC	Andre's Continental Food Market	Heeneth Ghandi
4	ANTHB	Anthony's Beer and Ale	Mary Throneberry
5	AROUT	Around the Horn	Thomas Hardy
6	BABUJ	Babu Ji's Exports	G.K.Chattergee
7	BERGS	Bergstad's Scandinavian Grocery	Tammy Wong

Run this piece of code:

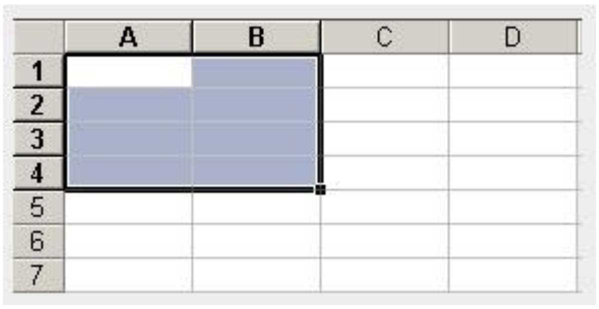
```
Sub RegionSelection()
    ActiveCell.CurrentRegion.Offset(1, 0).Resize( _
        ActiveCell.CurrentRegion.Rows.Count - 1, _
            ActiveCell.CurrentRegion.Columns.Count).Select
End Sub
```

Make sure the active cell is in the region of data you wish to capture before running the above procedure.

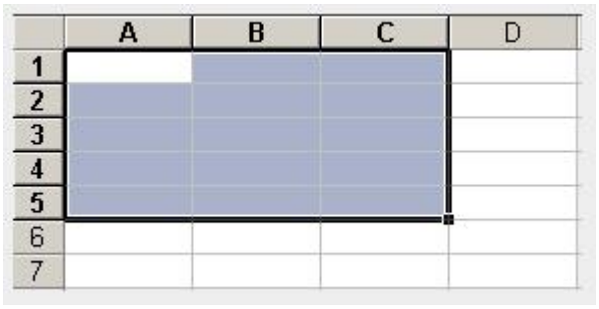
l. RESIZE Property

This property is useful for extending or re-defining a new size range.

To extend this range



By one row and one column to



Use the code snippet below:

```
Sub ResizeRange()
    Dim rows As Integer
    Dim cols As Integer
    cols = Selection.Columns.Count
    rows = Selection.rows.Count
    Selection.Resize(rows + 1, cols + 1).Select
End Sub
```

Resizing a range can be increased, decreased or change the configuration (*shape*) by combining positive and negative values inside the **Resize** property's arguments.

m. OFFSET Property

This property is used in many procedures as it controls references to other cells and navigation.

Two arguments are passed into this property that is then compounded with either another property or a method.

```
Selection.OffSet(1, 2).Select
```

```
ActiveCell.OffSet(0, -1).Value = "X"
```

Consider referring to an offset position rather than physically navigating to it – this will speed up the execution of code particularly while iterating.

For example:

```
Sub OffSetExample1()
    Dim intCount As Integer
    Do Until intCount = 10
        ActiveCell.Offset(intCount, 0).Value = "X"
        intCount = intCount + 1
    Loop
End Sub
```

is quicker to execute than:

```
Sub OffSetExample2()
    Dim intCount As Integer
    Do Until intCount = 10
        ActiveCell.Value = "X"
        ActiveCell.Offset(1, 0).Select
        intCount = intCount + 1
    Loop
End Sub
```

The above two examples produce the same result but instead of telling Excel to move to the active cell and then enter a value, it is more efficient to refer (*or point*) to the resulting cell and remain in the same position.

- A positive value for the row argument refers to a row downwards.
- A positive value for the column argument refers to a column to its right.
- A negative value for the row argument refers to a row upwards.
- A negative value for the column argument refers to a column to its left.

n. ACTIVATE Method

This method should not be confused with the **Select** method as commonly used in VBA.

The **Select** method means go and navigate to it.

```
Range(“A1”).Select.

Range(“A1:C10”).Select
```

The **Activate** method selects a cell within a selection.

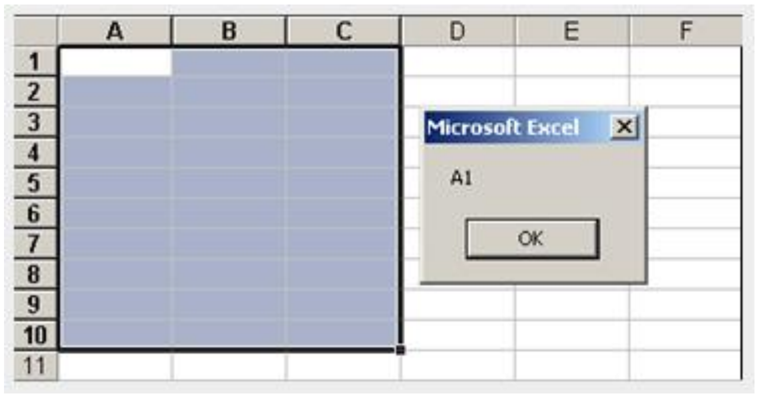
By default, in a selection of cells, the first (*top left position*) is the active cell *(white cell in a block*).

Example:

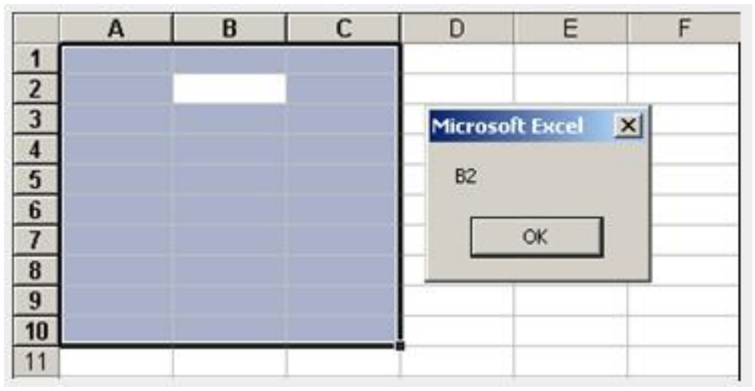
```
Sub ActivateMethodExample()
    'select a fixed range
    Range("A1:C10").Select
    MsgBox ActiveCell.Address(False, False)
    Range("B2").Activate
    MsgBox ActiveCell.Address(False, False)
End Sub
```

The above procedure selects a fixed range of cells with a message box confirming the address of the active cell. Then, using the **Activate** method, move the active cell to address B2.

From



to



o. CLEAR Methods

There are six variations of this method:

1. **Clear** – all attributes are cleared and reset to default
2. **ClearComments** – clear comments only
3. **ClearContents** – clear contents only (delete key command)
4. **ClearFormats** – clear formats only (revert to general format)
5. **ClearNotes** – clear comments and sound notes only
6. **ClearOutline** – clear on outlines implemented

Simply locate the object and use of the above methods:

```
'Different ways to refer to a selection
Sub ClearMethodsExamples()
    Range("A1:C10").Clear
    Selection.ClearComments
    Selection.CurrentRegion.ClearContents
    ActiveCell.ClearFormats
    Range(Cells(1, 1), Cells(5, 3)).ClearNotes
    Columns("A:E").ClearOutline
End Sub
```

p. CUT, COPY and PASTESPECIAL Methods

These methods simulate the windows clipboard cut, copy and paste commands.

There are a few different types of these methods where most arguments are optional and by changing the argument settings, will change the behavior of the method.

Some examples:

```
'Simple Copy and Paste
Sub CopyPasteData1()
    Range("A1").Copy
    Range("B1").PasteSpecial xlPasteAll
End Sub
```

```
'Copy and Paste Values only (no format)
Sub CopyPasteData2()
    Range("A1").Copy
    Range("B1").PasteSpecial xlPasteValues
End Sub
```

```
'Simple Cut and Paste
Sub CutPasteData()
    Range("A1").Cut Range("B1")
End Sub
```

If the copy and cut methods omit the argument **Destination**, the item is copied to the windows clipboard.

q. INSERT and DELETE Methods

These methods can add or remove cells, rows or columns and is best used with other properties to help establish which action to execute.

Some examples:

```
'Inserts an entire row at the active cell
Sub InsertRow()
    ActiveCell.EntireRow.Insert 'or EntireColumn
End Sub
```

```
'Deletes an entire row at the active cell
Sub DeleteRow()
    ActiveCell.EntireRow.Delete 'or EntireColumn
End Sub
```

```
'Inserts an entire row at row 4
Sub InsertAtRow4()
    ActiveSheet.rows(4).Insert
End Sub
```

```
'Insert columns and move data to the right
Sub InsertColumns()
    Range("A1:C5").Insert Shift:=xlShiftToRight
End Sub
```

r. Using the SET Keyword Command

Users can create and set a range object instead and like all other object declarations, use the **Set** command:

```
'Alternative way of referring to a range
Sub RangeObject()
    Dim rng As Range
    Set rng = Range("A1:B2")
    With rng
        .Value = "X"
        .Font.Bold = True
        .Borders.LineStyle = xlDouble
        'any other properties.........
    End With
    Set rng = Nothing
End Sub
```

This is an alternative way of working with the range object and is sometimes preferred as it exposes more members.

For example, using a **For...Next**, iterating in a collection is carried out by declaring and setting a variable as a **Range** object:

```
'Loops through the elements of the Range object
Sub IterateRangeObject()
    Dim r1 As Range
    Dim c As Object
    Set r1 = Range("A1:C10")
    For Each c In r1
        If c.Value = Empty Then
            c.Value = "0"
        End If
    Next c
End Sub
```

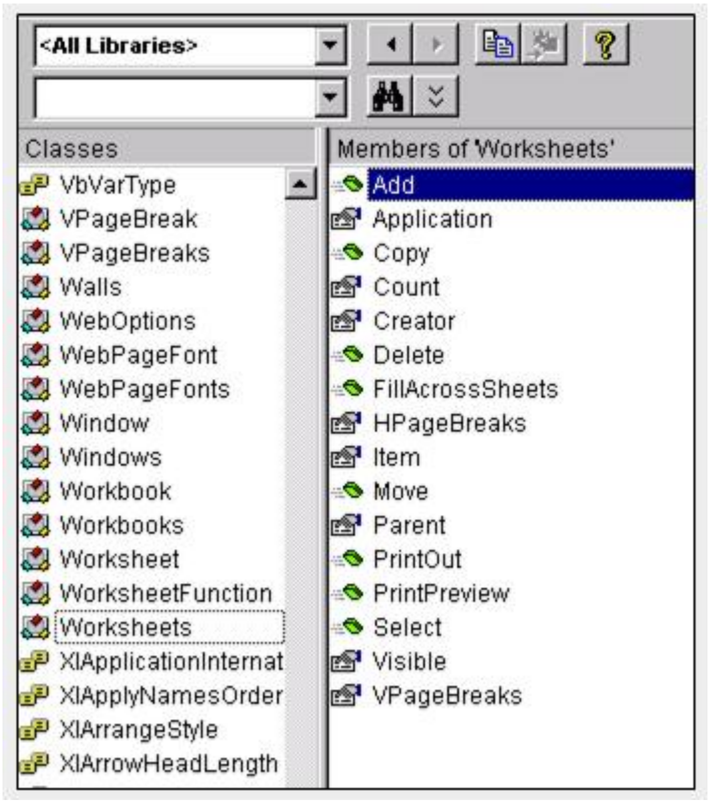
The above procedure checks each cell in a fixed range (*A1 to C10*) and determines its value, placing a 0 (*zero*) if the cell is empty.

3. OBJECT BROWSER

The Object Browser enables you to see a list of all the different objects with their methods, properties, events and constants.

In the VBE editor:

1. Insert menu and select a Module.
2. Click on the **View** menu and select **Object Browser** (shortcut key: **F2**).
3. Make sure it's set to **'<All Libraries>'**.



Notice **Add**([*Before*], [*After*], [*Count*], [*Type*]) is one of the examples previously seen.

The main two panes contain on the left **Classes** (also known as Objects) and on the right **Members** (also known as Identifiers).

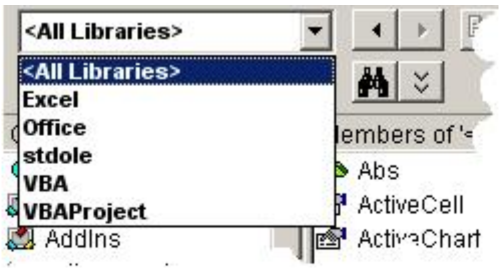
By selecting a class, you display its members which are of three key types; **Properties**, **Methods** and **Events**.

s. Libraries

Libraries are the application divisions of a collection of classes (objects). Therefore, you will have a class for **Excel**, **VBA** and other applications you wish to have a reference to. The **Excel** (the host), **VBA** and **VBAProject** are mandatory and cannot disabled. All other library files can be switched on or off as required.

In order to code to another applications (for example, Microsoft Word) you will need to load its library first.

To switch between libraries or show all libraries, choose the '*Project/Library'* drop down box:



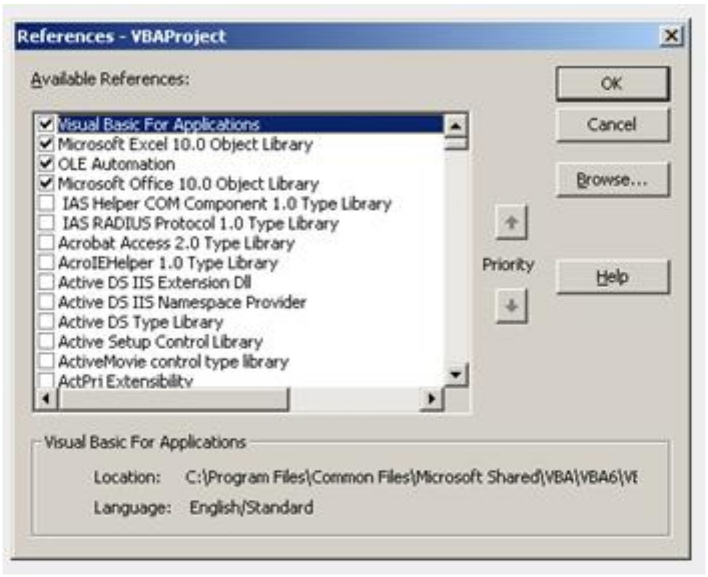
The default libraries available:

1. **Excel** – A collection of classes available in Excel i.e. workbook, worksheet, range, chart, etc…
2. **Office** – A collection of classes generic to all office applications i.e. command bar, command icon, help assistance, etc…
3. **stdole** – A collection of standard OLE classes which allow other OLE applications to share information (Not covered in this manual).
4. **VBA** – A collection of classes which allow generic functions to be used i.e. MsgBox, InputBox, conversion functions, string functions, etc…
5. **VBAProject** – A collection of classes local to the active workbook project, which includes sheets, workbook and any user, defined classes.

Other libraries are also available but require to be enabled before they can be used which include **Word**, **Outlook**, **DAO, ADODB** and many others.

By enabling the additional libraries, developers can start to code and communicate with other applications and processes, which start to reveal the potential power of Visual Basic (VBA).

To enable a library, from the Visual Basic Editor, choose **Tools** menu and select **References…**


References - VBAProject

Available References:

- ✓ Visual Basic For Applications
- ✓ Microsoft Excel 10.0 Object Library
- ✓ OLE Automation
- ✓ Microsoft Office 10.0 Object Library
- ☐ IAS Helper COM Component 1.0 Type Library
- ☐ IAS RADIUS Protocol 1.0 Type Library
- ☐ Acrobat Access 2.0 Type Library
- ☐ AcroIEHelper 1.0 Type Library
- ☐ Active DS IIS Extension Dll
- ☐ Active DS IIS Namespace Provider
- ☐ Active DS Type Library
- ☐ Active Setup Control Library
- ☐ ActiveMovie control type library
- ☐ ActPri Extensibility

Visual Basic For Applications

Location: C:\Program Files\Common Files\Microsoft Shared\VBA\VBA6\VE

Language: English/Standard

Scroll down to find the required library or choose the **Browse...** button to locate the required library.

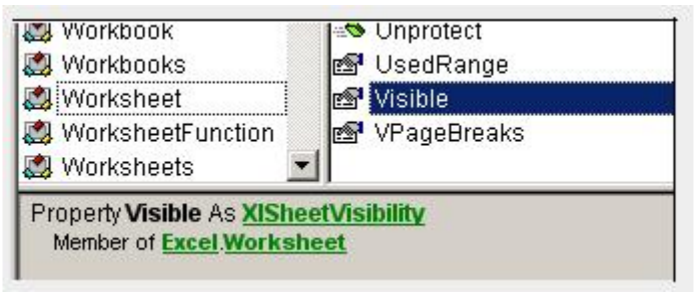
Excluding the top two libraries, a library priority order matters that is why users can re-arrange the order using the **Priority** buttons. The way the system works is when a procedure is executed, it checks to see which library is required in order to execute the line-by-line code. In some cases, a method or class can be duplicated between libraries and it is therefore important to be able to call the correct method or class first superseding the lower level references.

t. Structure of a Library

Each **Library** will typically have a collection of **classes**. A **class** or **object class** is in essence the object i.e. Worksheet.

Each object class will have a collection of **members**, which could be a collection of properties, methods and events.

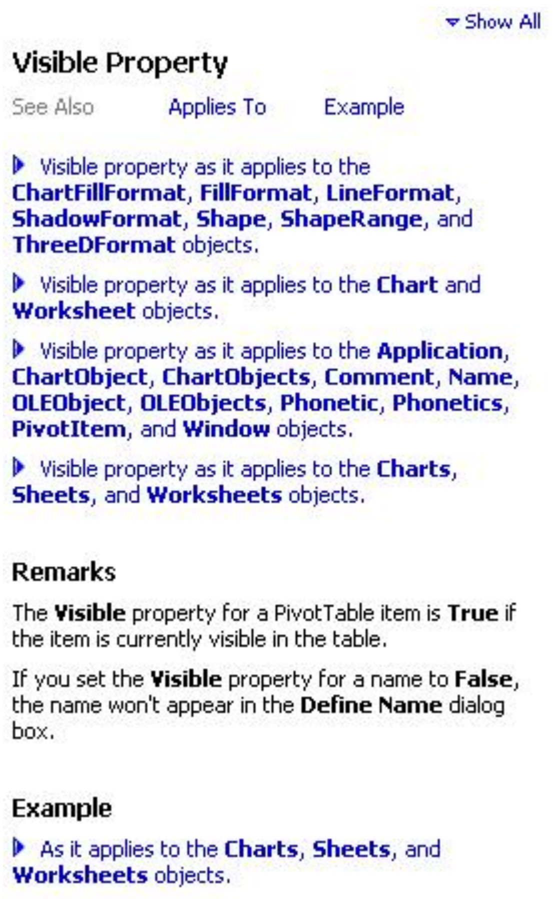
When looking at the Object Browser, users will see on the left hand side many classes to the active library. To the right hand pane, all members of the selected class will reveal properties, methods and events.



Property **Visible** As XlSheetVisibility
Member of Excel.Worksheet

The above illustration shows the **Excel** library, Worksheet class and the **Visible** property highlighted (**Excel.Worksheet.Visible**).

Right mouse click the selected item to choose the **Help** command and go straight to the offline help documentation.



▾ Show All

Visible Property

See Also Applies To Example

▸ Visible property as it applies to the **ChartFillFormat**, **FillFormat**, **LineFormat**, **ShadowFormat**, **Shape**, **ShapeRange**, and **ThreeDFormat** objects.

▸ Visible property as it applies to the **Chart** and **Worksheet** objects.

▸ Visible property as it applies to the **Application**, **ChartObject**, **ChartObjects**, **Comment**, **Name**, **OLEObject**, **OLEObjects**, **Phonetic**, **Phonetics**, **PivotItem**, and **Window** objects.

▸ Visible property as it applies to the **Charts**, **Sheets**, and **Worksheets** objects.

Remarks

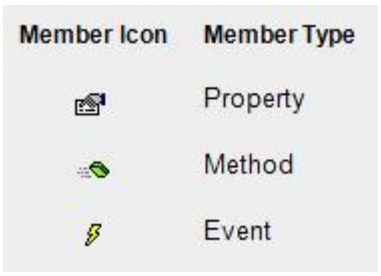
The **Visible** property for a PivotTable item is **True** if the item is currently visible in the table.

If you set the **Visible** property for a name to **False**, the name won't appear in the **Define Name** dialog box.

Example

▸ As it applies to the **Charts**, **Sheets**, and **Worksheets** objects.

Browsing the right hand and pane of the Object Browser, users will see three different icons to help identify the member type:

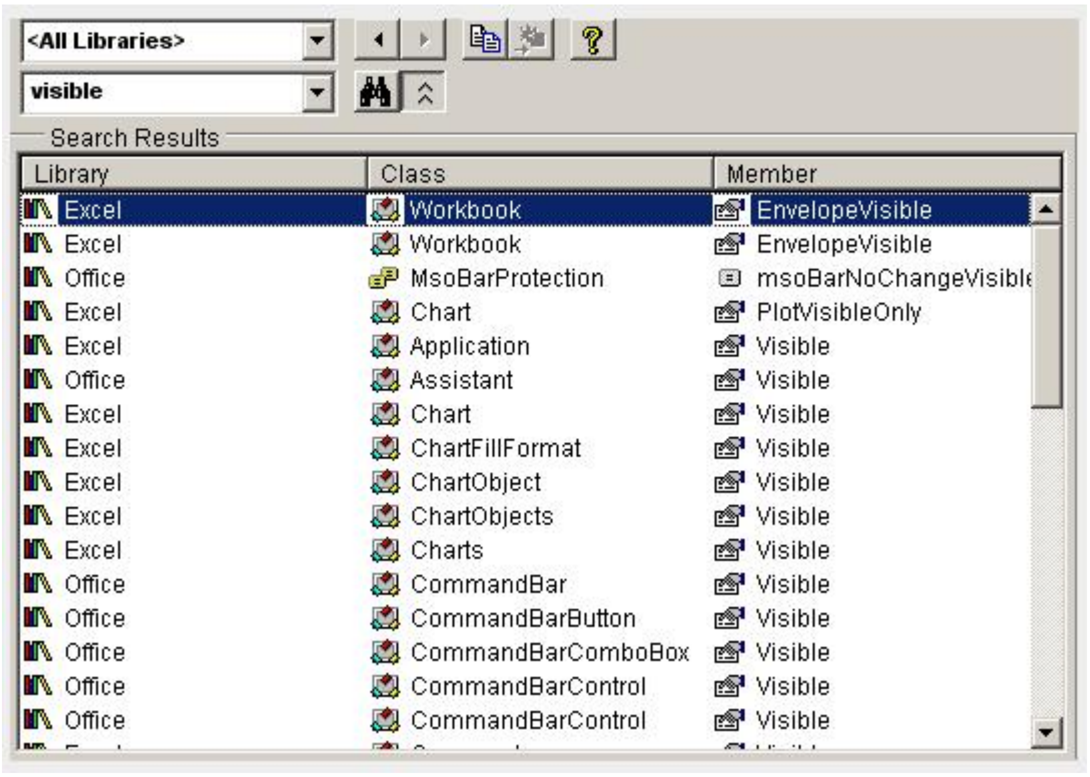


Member Icon	Member Type
🖹	Property
●	Method
⚡	Event

u. Browser Searching

The search tool allows users to locate if a keyword firstly exists and secondly where it could possibly reside

At the top half of the browser window, type the desired keyword and then click the search button:



Library	Class	Member
Excel	Workbook	EnvelopeVisible
Excel	Workbook	EnvelopeVisible
Office	MsoBarProtection	msoBarNoChangeVisible
Excel	Chart	PlotVisibleOnly
Excel	Application	Visible
Office	Assistant	Visible
Excel	Chart	Visible
Excel	ChartFillFormat	Visible
Excel	ChartObject	Visible
Excel	ChartObjects	Visible
Excel	Charts	Visible
Office	CommandBar	Visible
Office	CommandBarButton	Visible
Office	CommandBarComboBox	Visible
Office	CommandBarControl	Visible
Office	CommandBarControl	Visible

The above example looked for the keyword **visible** across all libraries.

After locating the correct item (*selecting the item*), users can use the copy button function and then paste into a code window.

4. GETOPENFILENAME METHOD

Displays the standard **Open** dialog box and gets a file name from the user without actually opening any files.

v. Syntax

expression **.GetOpenFilename(*FileFilter*, *FilterIndex*, *Title*, *ButtonText*, *MultiSelect*)**

expression Required. An expression that returns an **Application** object.

FileFilter Optional **Variant**. A string specifying file filtering criteria.

This string consists of pairs of file filter strings followed by the MS-DOS wildcard file filter specification, with each part and each pair separated by commas. Each separate pair is listed in the **Files of type** drop-down list box. For example, the following string specifies two file filters—text and addin: "Text Files (*.txt),*.txt,Add-In Files (*.xla),*.xla".

To use multiple MS-DOS wildcard expressions for a single file filter type, separate the wildcard expressions with semicolons; for example, "Visual Basic Files (*.bas; *.txt),*.bas;*.txt".

If omitted, this argument defaults to "All Files (*.*),*.*".

FilterIndex Optional **Variant**. Specifies the index numbers of the default file filtering criteria, from 1 to the number of filters specified in ***FileFilter***. If this argument is omitted or greater than the number of filters present, the first file filter is used.

Title Optional **Variant**. Specifies the title of the dialog box. If this argument is omitted, the title is "Open."

ButtonText Optional **Variant**. Macintosh only.

MultiSelect Optional **Variant**. **True** to allow multiple file names to be selected. **False** to allow only one file name to be selected. The default value is **False**

w. Remarks

This method returns the selected file name or the name entered by the user. The returned name may include a path specification. If ***MultiSelect*** is **True**, the return value is an array of the selected file names (even if only one filename is selected). Returns **False** if the user cancels the dialog box.

This method may change the current drive or folder.

x. Example

This example displays the **Open** dialog box, with the file filter set to text files. If the user chooses a file name, the code displays that file name in a message box.

```
fileToOpen = Application.GetOpenFilename("Text Files (*.txt), *.txt")

If fileToOpen <> False Then

    MsgBox "Open " & fileToOpen

End If
```

5. GETSAVEASFILENAME METHOD

Displays the standard **Save As** dialog box and gets a file name from the user without actually saving any files.

y. Syntax

expression **.GetSaveAsFilename(*InitialFilename*, *FileFilter*, *FilterIndex*, *Title*, *ButtonText*)**

expression Required. An expression that returns an **Application** object.

InitialFilename Optional **Variant**. Specifies the suggested file name. If this argument is omitted, Microsoft Excel uses the active workbook's name.

FileFilter Optional **Variant**. A string specifying file filtering criteria.

This string consists of pairs of file filter strings followed by the MS-DOS wildcard file filter specification, with each part and each pair separated by commas. Each separate pair is listed in the **Files of type** drop-down list box. For example, the following string specifies two file filters, text and addin: "Text Files (*.txt), *.txt, Add-In Files (*.xla), *.xla".

To use multiple MS-DOS wildcard expressions for a single file filter type, separate the wildcard expressions with semicolons; for example, "Visual Basic Files (*.bas; *.txt),*.bas;*.txt".

If omitted, this argument defaults to "All Files (*.*),*.*".

FilterIndex Optional **Variant**. Specifies the index number of the default file filtering criteria, from 1 to the number of filters specified in ***FileFilter***. If this argument is omitted or greater than the number of filters present, the first file filter is used.

Title Optional **Variant**. Specifies the title of the dialog box. If this argument is omitted, the default title is used.

ButtonText Optional **Variant**. Macintosh only.

z. Remarks

This method returns the selected file name or the name entered by the user. The returned name may include a path specification. Returns **False** if the user cancels the dialog box.

This method may change the current drive or folder.

aa. Example

This example displays the **Save As** dialog box, with the file filter set to text files. If the user chooses a file name, the example displays that file name in a message box.

```
fileSaveName = Application.GetSaveAsFilename( fileFilter:="Text Files
(*.txt), *.txt")

If fileSaveName <> False Then

    MsgBox "Save as " & fileSaveName

End If
```

6. APPLICATION.FILEDIALOG

Returns a FileDialog object representing an instance of the file dialog.

bb. Syntax

expression .FileDialog(fileDialogType)

expression A variable that represents an Application object.

cc. Parameters

Name	Required/Optional	Data Type	Description
fileDialogType	Required	MsoFileDialogType	The type of file dialog.

dd. Remarks

MsoFileDialogType can be one of these MsoFileDialogType constants:

- msoFileDialogFilePicker. Allows user to select a file.
- msoFileDialogFolderPicker. Allows user to select a folder.
- msoFileDialogOpen. Allows user to open a file.
- msoFileDialogSaveAs. Allows user to save a file.

ee. Example

In this example, Microsoft Excel opens the file dialog allowing the user to select one or more files. Once these files are selected, Excel displays the path for each file in a separate message.

```
Sub UseFileDialogOpen()

    Dim lngCount As Long

    ' Open the file dialog

    With Application.FileDialog(msoFileDialogOpen)

        .AllowMultiSelect = True

        .Show

        ' Display paths of each file selected

        For lngCount = 1 To .SelectedItems.Count
```

```
            MsgBox .SelectedItems(lngCount)

        Next lngCount

    End With

End Sub
```

7. COMMANDBARS.EXECUTEMSO METHOD

Executes the control identified by the idMso parameter.

ff. Syntax

```
expression.ExecuteMso(idMso)
```

expression An expression that returns a CommandBars object.

gg. Parameters

Name	Required/Optional	Data Type	Description
idMso	Required	String	Identifier for the control.

hh. Remarks

This method is useful in cases where there is no object model for a particular command. Works on controls that are built-in buttons, toggleButtons and splitButtons. On failure it returns E_InvalidArg for an invalid IdMso, and E_Fail for controls that are not enabled or not visible.

ii. Example

The following sample executes the Copy button.

```
Application.CommandBars.ExecuteMso(“Copy”)
```

1. CONTENTS

2. MAJOR VBA FUNCTIONS

a. CONVERSION Functions

This class contains various functions to help cast and convert values from one data type to another.

Some functions in this class:

CBool(*expression***)** – convert to a Boolean (true/false) value
CByte(*expression***)** – convert to a Byte value
CCur(*expression***)** – convert to a Currency value
CDate(*expression***)** – convert to Date value
CDbl(*expression***)** – convert to a Double value
CDec(*expression***)** – convert to a Decimal value
CInt(*expression***)** – convert to an Integer value
CLng(*expression***)** – convert to a Long value
CSng(*expression***)** – convert to a Single value
CStr(*expression***)** – convert to a String value
CVar(*expression***)** – convert to Variant value

The expression can be either a string or numeric value, which is converted to one of the above data types.

```
Sub ConvertValue()
    Dim strInput As String
    Dim intNumber As Integer
    strInput = InputBox("Enter a Number:")
    intNumber = CInt(strInput)
    .....................
End Sub
```

The above example takes a string variable, which the InputBox function returns as a string and converts it to an integer value and stores to the integer, variable.

b. DATETIME Functions

This class is a collection of date/time conversions and interrogations.

Some functions in this class:

Date – return/sets the system's date

Now – returns/sets the system's date and time

Day(*Date***)** – returns the day element of the date

Month(*Date***)** – returns the month element of the date

Year(*Date***)** – returns the year element of the date

DateDiff(*interval, date1, date2* [,*firstdayofweek*] [,*firstweekofyear*]**)**

– returns the difference between two dates driven by the interval

DateSerial(*Year, Month, Day***)** – returns a valid date from 3 separate values

DateValue(*Date***)** – converts a string date into a date data type date

Weekday(*Date*, [*firstdayofweek*]**)** – Returns a string day of the week

```
'Converts a string date to date date type date
Sub DateExample1()
    Dim strDate As String
    strDate = "10 May 2010"
    MsgBox DateValue(strDate)
End Sub
```

```
'Works out the difference between two dates
'returns the number of months (interval)
Sub DateExample2()
    Dim dtmStartDate As Date
    dtmStartDate = #5/2/2010#
    MsgBox DateDiff("m", dtmStartDate, Date) & " Months"
End Sub
```

c. INFORMATION Functions

This class is a collection of status functions to help evaluate conditions of variables and objects alike.

Some functions in the class:

IsArray(*variant***)** – returns true or false

IsDate(*expression***)** – returns true or false

IsError(*expression***)** – returns true or false

IsEmpty(*expression***)** – returns true or false

IsMissing(*variant***)** – returns true or false

IsNull(*expression***)** – returns true or false

IsNumeric(*expression***)** – returns true or false

IsObject(*expression***)** – returns true or false

An expression or variant is the variable being tested to see if it is **True** or **False**.

d. MATH Functions

This class is a collection of mathematical functions that can be used to change variables with ease and without having to create your own functions.

Some functions in the class:

Abs(*Number***)** – returns the absolute number (always a positive value)

Rnd([*Number*]**)** – returns a random value

Round(*Number, [NumDigitsAfterDecimal]***)** – returns a rounded value

Sqr(*Number***)** – returns a square value (x^2)

```
'Generates a random value between 1 and 100.
Sub RandomNumber()
    Dim intNumber As Integer
    intNumber = Int((100 * Rnd) + 1)
    MsgBox intNumber
End Sub
```

e. STRING Functions

This class is a collection of text (*string*) based functions that include conversion, extractions and concatenation.

Some functions in the class:

Asc(*String***)** – returns the numeric ASCII value of the character string

Chr(*CharCode***)** – returns the character string from the code supplied

Format(*Expression*, [*Format*], [*FirstDayOfWeek*], [*FirstWeekOfYear*]**)**

- returns the format presentation of the expression

InStr([*Start*], [*String1*], [*String2*], [*Compare*]**)** – returns the numeric position of the first character found from left to right

InStrRev(*StringCheck*, *StringMatch*, [*Start*], [*Compare*]**)** – returns the numeric position of the first character found from right to left

LCase(*String***)** – returns the string in lowercase

UCase(*String***)** – returns the string in uppercase

Left(*String*, *Length***)** – returns the remaining characters from the left of the length supplied

Right(*String*, *Length***)** – returns the remaining characters from the right of the length supplied

Len(*Expression***)** – returns a value of the length of characters supplied

Mid(*String*, *Start*, [*Length*]**)** – returns the portion of characters as determined by the start and end parameters supplied

Trim(*String***)** – removes unwanted spaces from left and right of the string supplied and extra spaces in between multiple strings

```
Sub StringExample1()
    Dim strString As String
    strString = "Microsoft Excel VBA"
    'Returns 17 (17th character starting from first character)
    MsgBox InStr(1, strString, "V", vbTextCompare)
    'Returns 7 (7th character from left starting 'at the sixth position)
    MsgBox InStr(6, strString, "o", vbTextCompare)
End Sub
```

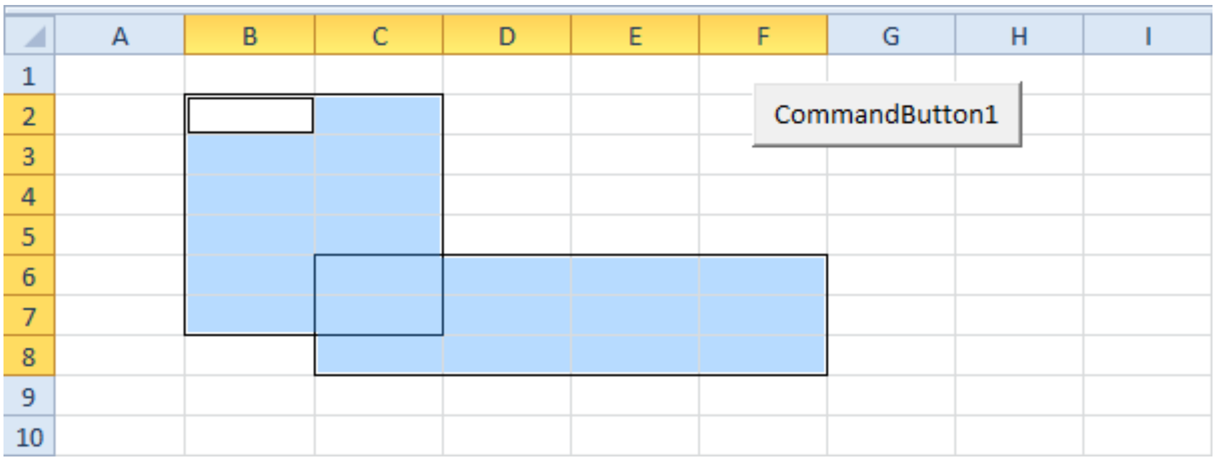
```
Sub StrngExample2()
    MsgBox Format(12.5 * 1.175, "£0.00")
End Sub
```

3. UNION AND INTERSECT METHIDS

The Union method in Excel VBA returns a Range object that represents the union of two or more ranges (borders below for illustration only).
Code line:

```
Union(Range("B2:C7"), Range("C6:F8")).Select
```

Result:

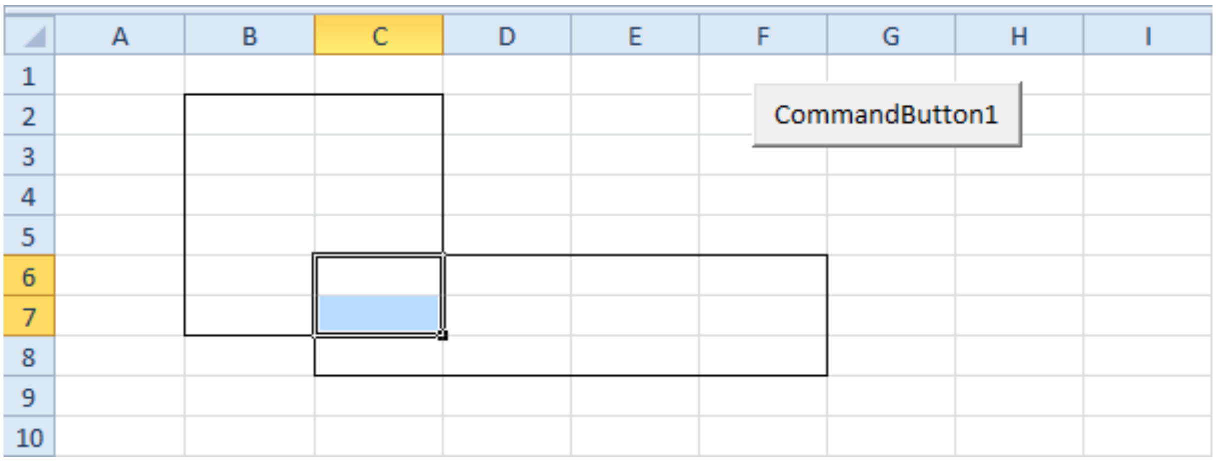


Note: the Union method doesn't return the mathematical union (cell C6 and cell C7 are included twice).

The Intersect method in Excel VBA returns a Range object that represents the intersection of two or more ranges (borders below for illustration only).
Code line:

```
Intersect(Range("B2:C7"), Range("C6:F8")).Select
```

Result:



4. ARRAYS

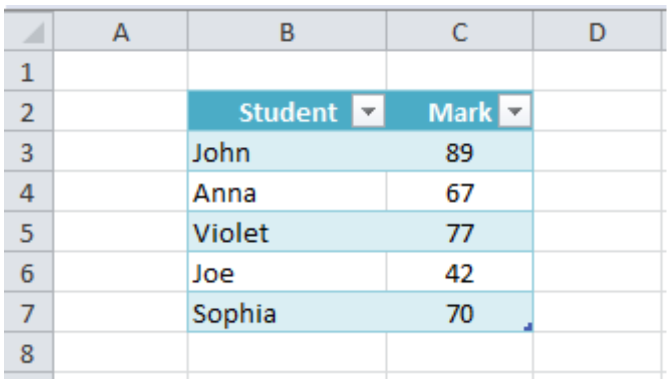
A VBA array is a type of variable. It is used to store lists of data of the same type. An example would be storing a list of countries or a list of weekly totals.

In VBA a normal variable can store only one value at a time. The following example shows a variable being used to store the marks of a student.

```
' Can only store 1 value at a time

Dim Student1 As Integer

Student1 = 55
```

If we wish to store the marks of another student then we need to create a second variable.

In the following example we have the marks of five students



	A	B	C	D
1				
2		Student	Mark	
3		John	89	
4		Anna	67	
5		Violet	77	
6		Joe	42	
7		Sophia	70	
8				

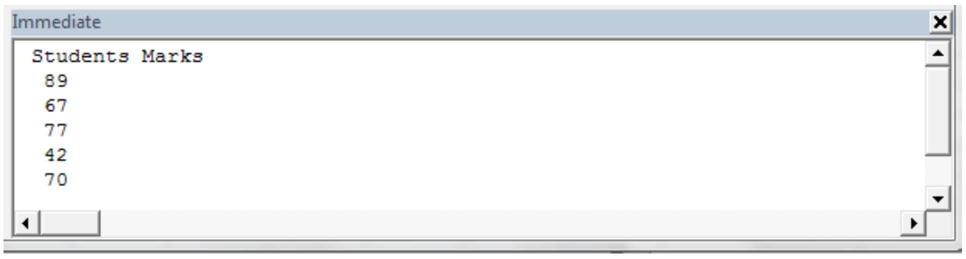
We are going to read these marks and write them to the Immediate Window.

As you can see in the following example we are writing the same code five times – once for each student

```
Public Sub StudentMarks()

   With ThisWorkbook.Worksheets("Sheet1")

      ' Declare variable for each student

      Dim Student1 As Integer

      Dim Student2 As Integer

      Dim Student3 As Integer
```

```
        Dim Student4 As Integer

        Dim Student5 As Integer



        ' Read student marks from cell

        Student1 = .Range("C2").Offset(1)

        Student2 = .Range("C2").Offset(2)

        Student3 = .Range("C2").Offset(3)

        Student4 = .Range("C2").Offset(4)

        Student5 = .Range("C2").Offset(5)



        ' Print student marks

        Debug.Print "Students Marks"

        Debug.Print Student1

        Debug.Print Student2

        Debug.Print Student3

        Debug.Print Student4

        Debug.Print Student5



    End With

End Sub
```

The following is the output from the example



```
Students Marks
 89
 67
 77
 42
 70
```

The problem with using one variable per student is that you need to add code for each student. Therefore if you had a thousand students in the above example you would need three thousand lines of code!

Luckily we have arrays to make our life easier. Arrays allow us to store a list of data items in one structure.

The following code shows the above student example using an array

```
Public Sub StudentMarksArr()



  With ThisWorkbook.Worksheets("Sheet1")



    ' Declare an array to hold marks for 5 students

    Dim Students(1 To 5) As Integer



    ' Read student marks from cells C3:C7 into array

    Dim i As Integer

    For i = 1 To 5

      Students(i) = .Range("C2").Offset(i)

    Next i



    ' Print student marks from the array

    Debug.Print "Students Marks"
```

```
        For i = LBound(Students) To UBound(Students)

            Debug.Print Students(i)

        Next i



    End With



End Sub
```

The advantage of this code is that it will work for any number of students. If we have to change this code to deal with 1000 students we only need to change the **(1 To 5)** to **(1 To 1000)** in the declaration. In the prior example we would need to add approximately five thousand lines of code.

Let’s have a quick comparison of variables and arrays. First we compare the declaration

```
' Variable

Dim Student As Integer

Dim Country As String



' Array

Dim Students(1 to 3) As Integer

Dim Countries(1 to 3) As String
```

Next we compare assigning a value

```
' assign value to variable

Student1 = .Cells(1, 1)

' assign value to first item in array

Students(1) = .Cells(1, 1)
```

Lastly we look at writing the values

```
' Print variable value

Debug.Print Student1



' Print value of first student in array

Debug.Print Students(1)
```

As you can see, using variables and arrays is quite similar.

The fact that arrays use an index(also called a subscript) to access each item is important. It means we can easily access all the items in an array using a For Loop.

Now that you have some background on why arrays are useful lets go through them step by step.

a. Types of VBA Arrays

There are two types of arrays in VBA

1. Static – an array of fixed size.
2. Dynamic – an array where the size is set at run time.

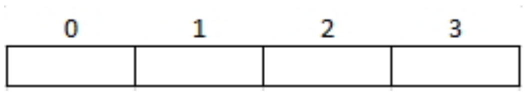
The difference between these arrays mainly in how they are created. Accessing values in both array types is exactly the same. In the following sections we will cover both types.

b. Declaring an Array

A static array is declared as follows

```
Public Sub DecArrayStatic()



   ' Create array with locations 0,1,2,3

   Dim arrMarks1(0 To 3) As Long



   ' Defaults as 0 to 3 i.e. locations 0,1,2,3

   Dim arrMarks2(3) As Long
```

```
    ' Create array with locations 1,2,3,4,5

    Dim arrMarks1(1 To 5) As Long



    ' Create array with locations 2,3,4 ' This is rarely used

    Dim arrMarks3(2 To 4) As Long



End Sub
```



As you can see the size is specified when you declare a static array. The problem with this is that you can never be sure in advance the size you need. Each time you run the Macro you may have different size requirements.

If you do not use all the array locations then the resources are being wasted. If you need more locations you can used ReDim but this is essentially creating a new static array.

The dynamic array does not have such problems. You do not specify the size when you declare it. Therefore you can then grow and shrink as required.

```
Public Sub DecArrayDynamic()



    ' Declare  dynamic array

    Dim arrMarks() As Long



    ' Set the size of the array when you are ready

    ReDim arrMarks(0 To 5)
```

```
End Sub
```

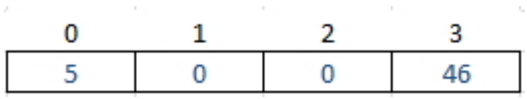
The dynamic array is not allocated until you use the ReDim statement. The advantage is you can wait until you know the number of items before setting the array size. With a static array you have to give the size up front.

To give an example. Imagine you were reading worksheets of student marks. With a dynamic array you can count the students on the worksheet and set an array to that size. With a static array you must set the size to the largest possible number of students.

c. Assigning Values to an Array

To assign values to an array you use the number of the location. You assign value for both array types the same way.

```
Public Sub AssignValue()

   ' Declare  array with locations 0,1,2,3

   Dim arrMarks(0 To 3) As Long



   ' Set the value of position 0

   arrMarks(0) = 5



   ' Set the value of position 3

   arrMarks(3) = 46



   ' This is an error as there is no location 4

   arrMarks(4) = 99



End Sub
```



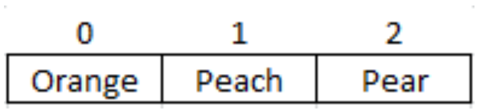
0	1	2	3
5	0	0	46

The number of the location is called the subscript or index. The last line in the example will give a "Subscript out of Range" error as there is no location 4 in the array example.

d. Using the Array and Split function

You can use the **Array** function to populate an array with a list of items. You must declare the array as a type Variant. The following code shows you how to use this function.

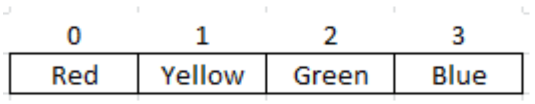
```
Dim arr1 As Variant

arr1 = Array("Orange", "Peach","Pear")



Dim arr2 As Variant

arr2 = Array(5, 6, 7, 8, 12)
```



0	1	2
Orange	Peach	Pear

The array created by the Array Function will start at index zero unless you use **Option Base 1**at the top of your module. Then it will start at index one. In programming it is generally considered poor practice to have your actual data in the code. However sometimes it is useful when you need to test some code quickly. The **Split** function is used to split a string into an array based on a delimiter. A delimiter is a character such as a comma or space that separates the items. The following code will split the string into an array of three elements.

```
Dim s As String

s = "Red,Yellow,Green,Blue"



Dim arr() As String

arr = Split(s, ",")
```



0	1	2	3
Red	Yellow	Green	Blue

The Split function is normally used when you read from a comma separated file or another source that provides a list of items separated by the same character.

e. Using Loops With Arrays

Using a For Loop allows quick access to all items in an array. This is where the power of using arrays becomes apparent. We can read arrays with ten values or ten thousand values using the same few lines of code. There are two functions in VBA called LBound and UBound. These functions return the smallest and largest subscript in an array. In an array arrMarks(0 to 3) the LBound will return 0 and UBound will return 3. The following example assigns random numbers to an array using a loop. It then prints out these numbers using a second loop.

```
Public Sub ArrayLoops()



  ' Declare  array

  Dim arrMarks(0 To 5) As Long



  ' Fill the array with random numbers

  Dim i As Long

  For i = LBound(arrMarks) To UBound(arrMarks)

      arrMarks(i) = 5 * Rnd

  Next i



  ' Print out the values in the array

  Debug.Print "Location", "Value"

  For i = LBound(arrMarks) To UBound(arrMarks)

      Debug.Print i, arrMarks(i)
```

```
    Next i



End Sub
```

The function LBound and UBound are very useful. Using them means our loops will work correctly with any array size. The real benefit is that if the size of the array changes we do not have to change the code for printing the values. A loop will work for an array of any size as long as you use these functions.

f. Using the For Each Loop

You can use the **For Each** loop with arrays. The important thing to keep in mind is that it is **Read-Only.** This means that you cannot change the value in the array. In the following code the value of **mark** changes but **<u>it does not change the value in the array.</u>**

```
For Each mark In arrMarks

    ' Will not change the array value

    mark = 5 * Rnd

Next mark
```

The For Each is loop is fine to use for reading an array. It is neater to write especially for a 2 dimensional array as we will see.

```
Dim mark As Variant

For Each mark In arrMarks

    Debug.Print mark

Next mark
```

g. Using Erase

The Erase function can be used on arrays but performs differently depending on the array type. For a static Array the Erase function resets all the values to the default. If the array is of integers then all the values are set to zero. If the array is of strings then all the strings are set to “” and so on. For a Dynamic Array the Erase function DeAllocates memory. That is, it deletes the array. If you want to use it again you must use ReDim to Allocate memory. Lets have a look an example for the static array. This example is the same as the ArrayLoops example in the last section with one difference – we use Erase after setting the values. When the value are printed out they will all be zero.

```
Public Sub EraseStatic()



    ' Declare  array

    Dim arrMarks(0 To 3) As Long



    ' Fill the array with random numbers

    Dim i As Long

    For i = LBound(arrMarks) To UBound(arrMarks)

        arrMarks(i) = 5 * Rnd

    Next i



    ' ALL VALUES SET TO ZERO

    Erase arrMarks



    ' Print out the values - there are all now zero

    Debug.Print "Location", "Value"

    For i = LBound(arrMarks) To UBound(arrMarks)

        Debug.Print i, arrMarks(i)

    Next i



End Sub
```

We will now try the same example with a dynamic. After we use Erase all the locations in the array have been deleted. We need to use ReDim if we wish to use the array again. If we try to access members of this array we will get a "Subscript out of Range" error.

```
Public Sub EraseDynamic()



   ' Declare  array

   Dim arrMarks() As Long

   ReDim arrMarks(0 To 3)



   ' Fill the array with random numbers

   Dim i As Long

   For i = LBound(arrMarks) To UBound(arrMarks)

      arrMarks(i) = 5 * Rnd

   Next i



   ' arrMarks is now deallocated. No locations exist.

   Erase arrMarks



End Sub
```

h. Passing an Array to a Sub or Function

Sometimes you will need to pass an array to a procedure. You declare the parameter using parenthesis similar to how you declare a dynamic array. Passing to the procedure using ByRef means you are passing a reference of the array. So if you change the array in the procedure it will be changed when you return. It is not possible to pass an array using ByVal.

```
' Passes array to a Function

Public Sub PassToProc()

    Dim arr(0 To 5) As String

    ' Pass the array to function

    UseArray arr

End Sub



Public Function UseArray(ByRef arr() As String)

    ' Use array

    Debug.Print UBound(arr)

End Function
```

i. Returning an Array from a Function

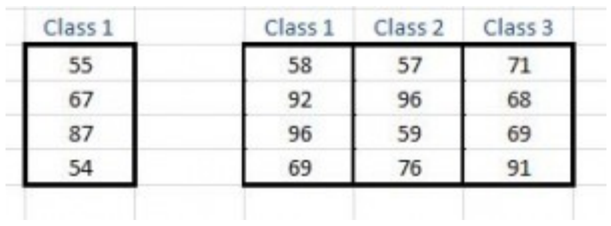
It is important to keep the following in mind. If you want to change an existing array in a procedure then you should pass it as a parameter using ByRef(see last section). You do not need to return the array from the procedure. The main reason for returning an array is when you use the procedure to create a new one. In this case you assign the return array to an array in the caller. This array cannot be already allocated. In other words you must use a dynamic array that has not been allocated. The following examples show this

```
Public Sub TestArray()



    ' Declare dynamic array - not allocated

    Dim arr() As String

    ' Return new array

    arr = GetArray

```

```
End Sub



Public Function GetArray() As String()



    ' Create and allocate new array

    Dim arr(0 To 5) As String

    ' Return array

    GetArray = arr



End Function
```

j. Two Dimensional Arrays

The arrays we have been looking at so far have been one dimensional arrays. This means the arrays are one list of items. A two dimensional array is essentially a list of lists. If you think of a single spreadsheet column as a single dimension then more than one column is two dimensional. In fact a spreadsheet is the equivalent of a 2 dimensional array. It has two dimensions – rows and columns. The following example shows two groups of data. The first is a one dimensional layout and the second is two dimensional.



Class 1
55
67
87
54

Class 1	Class 2	Class 3
58	57	71
92	96	68
96	59	69
69	76	91

To access an item in the first set of data(1 dimensional) all you need to do is give the row e.g. 1,2, 3 or 4. For the second set of data(2 dimensional) you need to give the row AND the column. So you can think of 1 dimensional being rows only and 2 dimensional as being rows and columns.

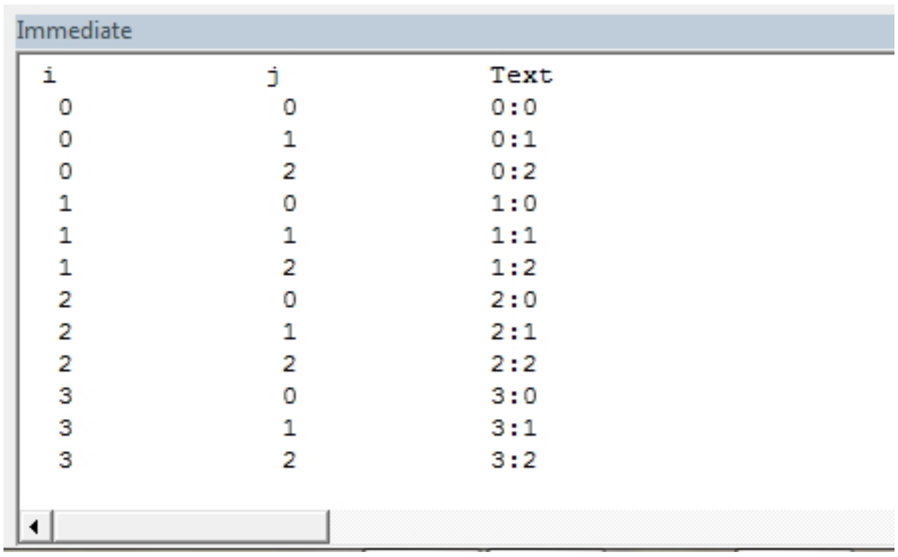
Note: It is possible to have more dimensions in an array. It is rarely required. If you are solving a problem using a 3+ dimensional array then there probably is a better way to do it. You declare a 2 dimensional array as follows

```
Dim ArrayMarks(0 to 2,0 to 3) As Long
```

The following example creates a random value for each item in the array and the prints the values to the Immediate Window.

```
Public Sub TwoDimArray()



    ' Declare a two dimensional array

    Dim arrMarks(0 To 3, 0 To 2) As String



    ' Fill the array with text made up of i and j values

    Dim i As Long, j As Long

    For i = LBound(arrMarks) To UBound(arrMarks)

        For j = LBound(arrMarks, 2) To UBound(arrMarks, 2)

            arrMarks(i, j) = CStr(i) + ":" + CStr(j)

        Next j

    Next i



    ' Print the values in the array to the Immediate Window

    Debug.Print "i", "j", "Value"

    For i = LBound(arrMarks) To UBound(arrMarks)

        For j = LBound(arrMarks, 2) To UBound(arrMarks, 2)

            Debug.Print i, j, arrMarks(i, j)
        Next j

    Next i



End Sub
```

You can see that we use a second For loop inside the first loop to access all the items. The output of the example looks like this:



```
i              j              Text
 0              0             0:0
 0              1             0:1
 0              2             0:2
 1              0             1:0
 1              1             1:1
 1              2             1:2
 2              0             2:0
 2              1             2:1
 2              2             2:2
 3              0             3:0
 3              1             3:1
 3              2             3:2
```

How this Macro works is as follows

- Enters the **i** loop
- **i** is set to 0
- Enters **j** loop
- **j** is set to 0
- **j** is set to 1
- **j** is set to 2
- Exit **j** loop
- **i** is set to 1
- **j** is set to 0
- **j** is set to 1
- **j** is set to 2
- And so on until **i**=3 and **j**=2

You may notice that LBound and UBound have a second argument of 2. This specifies that it is the upper or lower bound of the second dimension. That is the start and end location for **j**. The default value 1 which is why we do not need to specify it for the **i** loop.

k. Using the For Each Loop

Using a For Each is neater to use when reading from an array. Let’s take the code from above that writes out the two-dimensional array.

```
' Using For loop needs two loops

Debug.Print "i", "j", "Value"

For i = LBound(arrMarks) To UBound(arrMarks)

    For j = LBound(arrMarks, 2) To UBound(arrMarks, 2)

        Debug.Print i, j, arrMarks(i, j)

    Next j

Next i
```

Now let’s rewrite it using a For each loop. You can see we only need one loop and so it is much easier to write.

```
' Using For Each requires only one loop

Debug.Print "Value"

Dim mark As Variant

For Each mark In arrMarks

    Debug.Print mark

Next mark
```

Using the For Each loop gives us the array in one order only – from LBound to UBound. Most of the time this is all you need.

l. Reading from a Range of Cells to an Array

VBA has an extremely efficient way of reading from a Range of Cells to an Array and vice versa.

```
Public Sub ReadToArray()
```

```
    ' Create dynamic array

    Dim StudentMarks() As Variant



    ' Read values into array from first row

    StudentMarks = Range("A1:Z1").Value



    ' Write the values back to the third row

    Range("A3:Z3").Value = StudentMarks



End Sub
```

The dynamic array created in this example will be a two dimensional array. As you can see we can read from an entire range of cells to an array in just one line. The next example will read the sample student data below from C3:E6 of Sheet1 and print them to the Immediate Window.

```
Public Sub ReadAndDisplay()



    ' Get Range

    Dim rg As Range

    Set rg = ThisWorkbook.Worksheets("Sheet1").Range("C3:E6")



    ' Create dynamic array

    Dim StudentMarks() As Variant



    ' Read values into array from sheet1

    StudentMarks = rg.Value
```

```
 ' Print the array values

 Debug.Print "i", "j", "Value"

 Dim i As Long, j As Long

 For i = LBound(StudentMarks) To UBound(StudentMarks)

     For j = LBound(StudentMarks, 2) To UBound(StudentMarks, 2)

         Debug.Print i, j, StudentMarks(i, j)

     Next j

 Next i



End Sub
```

As you can see the first dimension(accessed using **i**) of the array is a row and the second is a column. To demonstrate this take a look at the value 44 in E4 of the sample data. This value is in row 2 column 3 of our data. You can see that 44 is stored in the array at **StudentMarks(2,3)**.

m. How To Make Your Macros Run at Super Speed

If your macros are running very slow then you may find this section very helpful. Especially if you are dealing with large amounts of data. The following is a well kept secret in VBA

Updating values in arrays is exponentially faster than updating values in cells.

In the last section, you saw how we can easily read from a group of cells to an array and vice versa. If we are updating a lot of values then we can do the following

1. Copy the data from the cells to an array.
2. Change the data in the array.
3. Copy the updated data from the array back to the cells.

For example, the following code would be much faster that the code below it

```
Public Sub ReadToArray()



    ' Read values into array from first row

    StudentMarks = Range("A1:Z20000").Value



    Dim c As Variant

    For Each c In StudentMarks

        ' Update values here

    Next c



    ' Write the new values back to the worksheet

    Range("A1:Z20000").Value = StudentMarks



End Sub
Sub UsingCellsToUpdate()



    Dim c As Variant

    For Each c In Range("A1:Z20000")

        ' Update values here

    Next c



End Sub
```

Assigning from one set of cells to another is also much faster than using Copy and Paste

```
' Assigning - this is faster

Range("A1:A10").Value = Range("B1:B10").Value



' Copy Paste - this is slower

Range("B1:B1").Copy Destination:=Range("A1:A10")
```

n. Conclusion

The following is a summary of the main points of this part.

1. Arrays are an efficient way of storing a **list of items** of the same type.
2. You can access an array item directly using the number of the location which is known as the **subscript or index**.
3. The common error "**Subscript out of Range**" is caused by accessing a location that does not exist.
4. There are two types of arrays: **Static** and **Dynamic**.
5. **Static** is used when the size of the array is always the same.
6. **Dynamic** arrays allow you to determine the size of an array at run time.
7. **LBound** and **UBound** provide a safe way of find the smallest and largest subscripts of the array.
8. The basic array is **one dimensional**. You can also have multi dimensional arrays.
9. You can only pass an array to a procedure using **ByRef**. You do this like this: ByRef arr() as long.
10. You can **return an array** from a function but the array, it is assigned to, must not be currently allocated.
11. A worksheet with it's rows and columns is essentially a **two dimensional** array.
12. You can read directly **from a worksheet range** into a two dimensional array in just one line of code.
13. You can also write from a two dimensional **array to a range** in just one line of code.

1. CONTENTS

2. BUILD IN EVENT ?

An event is basically something that happens in Excel. Following are a few examples of the types of events that Excel can deal with:

- A workbook is opened or closed.
- A window is activated.
- A worksheet is activated or deactivated.
- Data is entered into a cell or the cell is edited.
- A workbook is saved.
- A worksheet is calculated.
- An object, such as a button, is clicked.
- A particular key or key combination is pressed.
- A particular time of day occurs.
- An error occurs.

a. Workbook Events

Activate	The workbook is activated.
AddinInstall	An add-in is installed (relevant only for add-ins).
AddinUninstall	The add-in is uninstalled (relevant only for add-ins).
BeforeClose	The workbook is closed.
BeforePrint	The workbook is printed.
BeforeSave	The workbook is saved.
Deactivate	The workbook is deactivated.
NewSheet	A new sheet is added to the workbook.
Open	The workbook is opened.
SheetActivate	A sheet in the workbook is activated.
SheetBefore DoubleClick	A cell in the workbook is double-clicked.
SheetBefore RightClick	A cell in the workbook is right-clicked.
SheetCalculate	A sheet in the workbook is recalculated.
SheetChange	A change is made to a cell in the workbook.

SheetDeactivate	A sheet in the workbook is deactivated.
SheetFollowHyperlink	A hyperlink in a worksheet is clicked.
SheetSelectionChange	The selection is changed.
WindowActivate	The workbook window is activated.
WindowDeactivate	The workbook window is deactivated.
WindowResize	The workbook window is resized.

b. Worksheet events

Activate	The worksheet is activated.
BeforeDoubleClick	A cell in the worksheet is double-clicked.
BeforeRightClick	A cell in the worksheet is right-clicked.
Calculate	The worksheet is recalculated.
Change	A change is made to a cell in the worksheet.
Deactivate	The worksheet is deactivated.
FollowHyperlink	A hyperlink is activated.
SelectionChange	The selection is changed.

c. Are events useful?

At this point, you may be wondering how these events can be useful. Here's a quick example.

Suppose you have a workbook in which you enter data in column A. Your boss tells you that he needs to know exactly when each data point was entered. Entering data is an event — a WorksheetChange event. You can write a macro that responds to this event. That macro kicks in whenever the worksheet is changed. If the change was made in column A, the macro puts the date and time in column B, next to the data point that was entered.

Here is what such a macro would look like. (It should be in the Code module for the worksheet.)

```
Private Sub Worksheet_Change(ByVal Target As Range)
```

```
    If Target.Column = 1 Then
       Target.Offset(0, 1) = Now
    End If
End Sub
```

Just because your workbook contains procedures that respond to events doesn't guarantee that those procedures will actually run. As you know, it's possible to open a workbook with macros disabled. In such a case, all macros (even procedures that respond to events) are turned off. Keep this fact in mind when you create workbooks that rely on event-handler procedures.

d. Programming event-handler procedures

A VBA procedure that executes in response to an event is called an eventhandler procedure. These are always Sub procedures (as opposed to Function procedures). Writing these event-handlers is relatively straightforward after you understand how the process works. It boils down to a few steps:

1. Identify the event you want to trigger the procedure.
2. Press Alt+F11 to Activate the Visual Basic Editor.
3. In the VBE Project Window, double-click the appropriate object listed under Microsoft Excel Objects. For workbook-related events, the object is ThisWorkbook. For a worksheet related event, the object is a Worksheet object (such as Sheet1).
4. In the Code window for the object, write the event-handler procedure that is executed when the event occurs. This procedure will have a special name that identifies it as an eventhandler procedure.

e. Where Does the VBA Code Go?

It's very important to understand where your event-handler procedures go. They must reside in the Code window of an Object module. They do not go in a standard VBA module. If you put your event-handler procedure in the wrong place, it simply won't work. And you won't see any error messages either.

f. Writing an Event-Handler Procedure

The VBE helps you out when you're ready to write an event-handler procedure; it displays a list of all events for the selected object.

At the top of each Code window, you find two drop-down lists:

1. The Object drop-down list (the one on the left)
2. The Procedure drop-down list (the one on the right)

By default, the Object drop-down list in the Code window displays General. If you're writing an event handler for the ThisWorkbook object, you need to choose Workbook from the Object drop-down (it's the only other choice).

If you're writing an event-handler for a Sheet object, you need to choose Worksheet (again, the only other choice).

After you've made your choice from the Object drop-down list, then you can choose the event from the Procedure drop-down list.

When you select an event from the list, VBE automatically starts creating an event-handler procedure for you. This is a very useful feature, because you can verify that the proper arguments are used.

Here's a little quirk. When you first selected Workbook from the Object list, VBE always assumes that you want to create an event-handler procedure for the Open event and creates it for you. If you're actually creating a Workbook_Open procedure, that's fine. But if you're creating a different event-procedure, you need to delete the empty Workbook_Open Sub that Excel created.

You don't really have to use those two drop-downs, but it makes your job easier because the name of the event-handler procedure is critically important.

If you don't get the name exactly right, it won't work. Plus, some event handler procedures use one or more arguments in the Sub statement. There's no way you can remember what those arguments are. For example, if you select SheetActivate from the event list for a Workbook object, VBE writes the following Sub statement:

```
Private Sub Workbook_SheetActivate(ByVal Sh As Object)
```

In this case, Sh is the argument passed to the procedure and is a variable that represents the sheet in the activated workbook.

g. Order Of Events

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

```
Private Sub Worksheet_Change(ByVal Target As Range)

    Application.EnableEvents = False

    ' appropriate action here

    Application.EnableEvents = True

End Sub
```

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

h. Preventing Event Loops

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

```
Private Sub Worksheet_Change(ByVal Target As Range)

    Target.Value = Target.Value + 1

End Sub
```

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

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

```
Private Sub Worksheet_Change(ByVal Target As Range)

    Application.EnableEvents = False

    Target.Value = Target.Value + 1

    Application.EnableEvents = True

End Sub
```

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

```
    Public AbortChangeEvent As Boolean
```

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

```
Private Sub Worksheet_Change(ByVal Target As Range)

    If AbortChangeEvent = True Then

        Exit Sub

    End If

    '

    ' rest of code here

    '

End Sub
```

Finally, you would disable the Worksheet_Change event by setting the AbortChangeEvent variable to True. For example,

```
    AbortChangeEvent = True

    Range("A1").Value = 1234

    AbortChangeEvent = False
```

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

3. KEYBOARD EVENT

With Application.Onkey you can run a macro when you use a particular key or key combination or disable a particular key or key combination.

a. Examples of use

This example assigns "YourMacroName" to the key sequence SHIFT+CTRL+RIGHT ARROW

```
Application.OnKey "+^{RIGHT}", "YourMacroName"
```

This example returns SHIFT+CTRL+RIGHT ARROW to its normal meaning.

```
Application.OnKey "+^{RIGHT}"
```

This example disables the SHIFT+CTRL+RIGHT ARROW key sequence.

```
Application.OnKey "+^{RIGHT}", ""
```

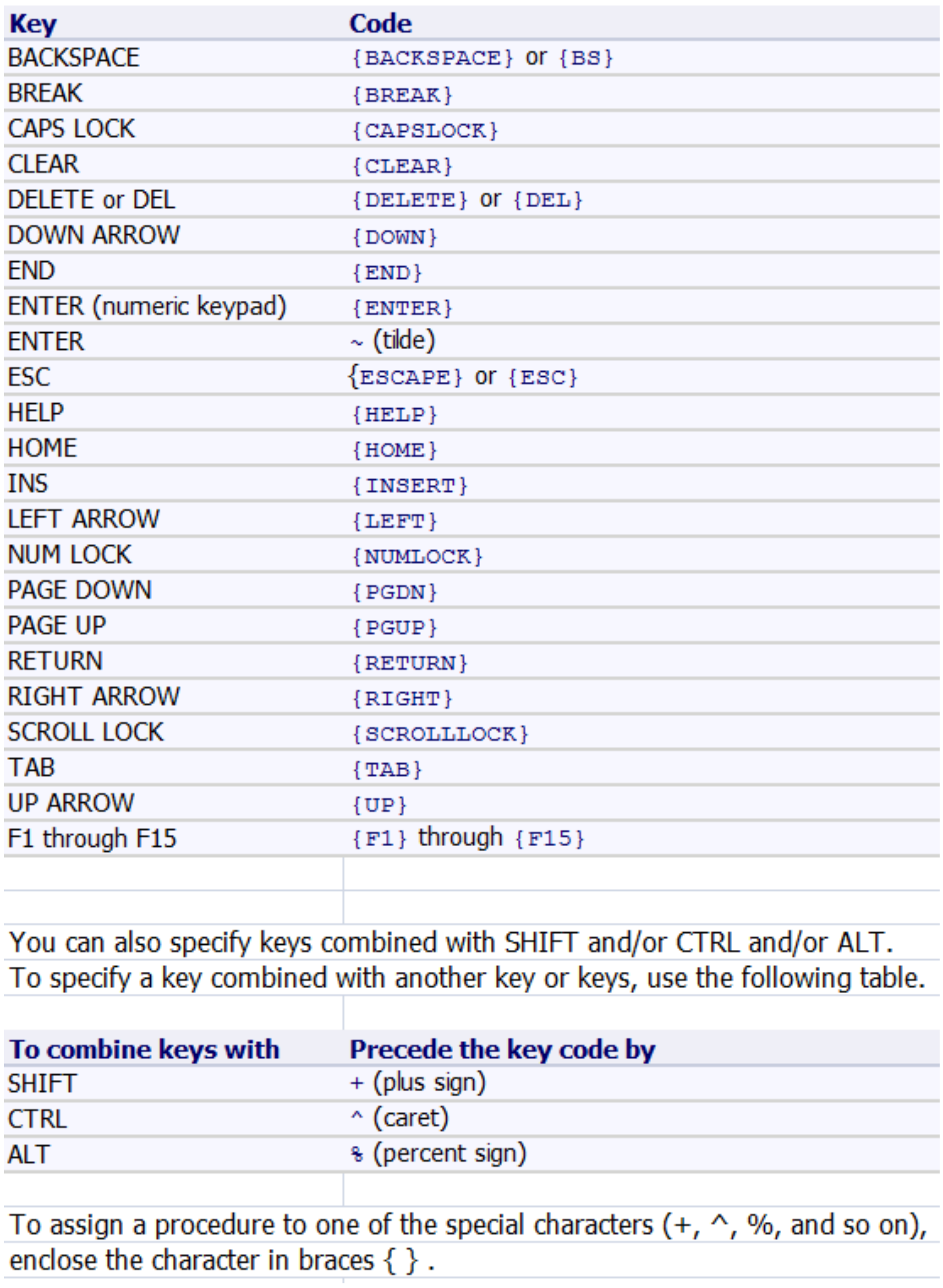
You can also use it to disable a built-in shortcut like Ctrl p that you can use to Print.

```
Application.OnKey "^p", ""
```

b. Key code

The Key argument can specify any single key combined with ALT, CTRL, or SHIFT, or any combination of these keys. Each key is represented by one or more characters, such as "a" for the character a, or "{ENTER}" for the ENTER key.

Shift key = "**+**" (plus sign)
Ctrl key = "**^**" (caret)
Alt key = "**%**" (percent sign)



Key	Code
BACKSPACE	{BACKSPACE} or {BS}
BREAK	{BREAK}
CAPS LOCK	{CAPSLOCK}
CLEAR	{CLEAR}
DELETE or DEL	{DELETE} or {DEL}
DOWN ARROW	{DOWN}
END	{END}
ENTER (numeric keypad)	{ENTER}
ENTER	~ (tilde)
ESC	{ESCAPE} or {ESC}
HELP	{HELP}
HOME	{HOME}
INS	{INSERT}
LEFT ARROW	{LEFT}
NUM LOCK	{NUMLOCK}
PAGE DOWN	{PGDN}
PAGE UP	{PGUP}
RETURN	{RETURN}
RIGHT ARROW	{RIGHT}
SCROLL LOCK	{SCROLLLOCK}
TAB	{TAB}
UP ARROW	{UP}
F1 through F15	{F1} through {F15}

You can also specify keys combined with SHIFT and/or CTRL and/or ALT.
To specify a key combined with another key or keys, use the following table.

To combine keys with	Precede the key code by
SHIFT	+ (plus sign)
CTRL	^ (caret)
ALT	% (percent sign)

To assign a procedure to one of the special characters (+, ^, %, and so on), enclose the character in braces { } .

Note: For some reason you can't disable every key combination with Onkey. I am not able to disable **Ctrl-**and **Ctrl+** for example but you can protect the worksheet to disable these two shortcuts that popup the Insert and Delete dialog.

Tip: If you want to use Onkey only for one workbook you can place the code in the Activate and Deactivate event in the ThisWorkbook module of that file.

```
Private Sub Workbook_Activate()
    Application.OnKey "+^{RIGHT}", "YourMacroName"
End Sub

Private Sub Workbook_Deactivate()
    Application.OnKey "+^{RIGHT}"
End Sub
```

1. CONTENTS

2. INPUTBOX

This function displays a dialog box for user input. Returns the information entered in the dialog box.

a. Syntax

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

expression Required. An expression that returns an **Application** object.

Prompt Required **String**. The message to be displayed in the dialog box. This can be a string, a number, a date, or a Boolean value (Microsoft Excel automatically coerces the value to a **String** before it's displayed).

Title Optional **Variant**. The title for the input box. If this argument is omitted, the default title is "Input."

Default Optional **Variant**. Specifies a value that will appear in the text box when the dialog box is initially displayed. If this argument is omitted, the text box is left empty. This value can be a Range object.

Left Optional **Variant**. Specifies an x position for the dialog box in relation to the upper-left corner of the screen, in points.

Top Optional **Variant**. Specifies a y position for the dialog box in relation to the upper-left corner of the screen, in points.

HelpFile Optional **Variant**. The name of the Help file for this input box. If the ***HelpFile*** and ***HelpContextID*** arguments are present, a Help button will appear in the dialog box.

HelpContextId Optional **Variant**. The context ID number of the Help topic in ***HelpFile***.

Type Optional **Variant**. Specifies the return data type. If this argument is omitted, the dialog box returns text. Can be one or a sum of the following values.

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.

b. Remarks

Use **InputBox** to display a simple dialog box so that you can enter information to be used in a macro. The dialog box has an **OK** button and a **Cancel** button. If you choose the **OK** button, **InputBox** returns the value entered in the dialog box. If you click the **Cancel** button, **InputBox** returns **False**.

If ***Type*** is 0, **InputBox** returns the formula in the form of text — for example, "=2*PI()/360". If there are any references in the formula, they are returned as A1-style references. (Use ConvertFormula to convert between reference styles.)

If ***Type*** is 8, **InputBox** returns a **Range** object. You must use the **Set** statement to assign the result to a **Range** object, as shown in the following example.

```
Set myRange = Application.InputBox(prompt := "Sample", type := 8)
```

If you don't use the **Set** statement, the variable is set to the value in the range, rather than the **Range** object itself.

If you use the **InputBox** method to ask the user for a formula, you must usethe **FormulaLocal** property to assign the formula to a **Range** object. The input formula will be in the user's language.

The **InputBox** method differs from the **InputBox** function in that it allows selective validation of the user's input, and it can be used with Microsoft Excel objects, error values, and formulas. Note that `Application.InputBox` calls the **InputBox** method; `InputBox` with no object qualifier calls the **InputBox** function.

c. Examples

This example prompts the user for a number.

```
myNum = Application.InputBox("Enter a number")
```

This example prompts the user to select a cell on Sheet1. The example uses the ***Type*** argument to ensure that the return value is a valid cell reference (a **Range** object).

```
Worksheets("Sheet1").Activate

Set myCell = Application.InputBox( Prompt:="Select a cell", Type:=8)
```

3. MSGBOX FUNCTION

Displays a message in a dialog box, waits for the user to click a button, and returns an **Integer** indicating which button the user clicked.

a. Syntax

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

The MsgBox function syntax has these named arguments:

Part	Description
prompt	Required. String expression displayed as the message in the dialog box. The maximum length of prompt is approximately 1024 characters, depending on the width of the characters used. If prompt consists of more than one line, you can separate the lines using a carriage return character (Chr(13)), a linefeed character (Chr(10)), or carriage return linefeed character combination (Chr(13) & Chr(10)) between each line.
buttons	Optional. Numeric expression that is the sum of values specifying the number and type of buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. If omitted, the default value for buttons is 0.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit title, the application name is placed in the title bar.
helpfile	Optional. String expression that identifies the Help file to use to provide context-sensitive Help for the dialog box. If helpfile is provided, context must also be provided.
context	Optional. Numeric expression that is the Help context number assigned to the appropriate Help topic by the Help author. If context is provided, helpfile must also be provided.

b. Settings

The ***buttons*** argument settings are:

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.
vbDefaultButton1	0	First button is default.
vbDefaultButton2	256	Second button is default.
vbDefaultButton3	512	Third button is default.
vbDefaultButton4	768	Fourth button is default.
vbApplicationModal	0	Application modal; the user must respond to the message box before continuing work in the current application.
vbSystemModal	4096	System modal; all applications are suspended until the user responds to the message box.
vbMsgBoxHelpButton	16384	Adds Help button to the message box
VbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window

vbMsgBoxRight	524288	Text is right aligned
vbMsgBoxRtlReading	1048576	Specifies text should appear as right-to-left reading on Hebrew and Arabic systems

The first group of values (05) describes the number and type of buttons displayed in the dialog box; the second group (16, 32, 48, 64) describes the icon style; the third group (0, 256, 512) determines which button is the default; and the fourth group (0, 4096) determines the modality of the message box. When adding numbers to create a final value for the ***buttons*** argument, use only one number from each group.

Note These constants are specified by Visual Basic for Applications. As a result, the names can be used anywhere in your code in place of the actual values.

c. Return Values

Constant	**Value**	**Description**
vbOK	1	**OK**
vbCancel	2	**Cancel**
vbAbort	3	**Abort**
vbRetry	4	**Retry**
vbIgnore	5	**Ignore**
vbYes	6	**Yes**
vbNo	7	**No**

d. Remarks

When both ***helpfile*** and ***context*** are provided, the user can press F1 to view the Help topic corresponding to the **context**. Some host applications, for example, Microsoft Excel, also automatically add a **Help** button to the dialog box.

If the dialog box displays a **Cancel** button, pressing the ESC key has the same effect as clicking **Cancel**. If the dialog box contains a **Help** button, context-sensitive Help is provided for the dialog box. However, no value is returned until one of the other buttons is clicked.

Note To specify more than the first named argument, you must use **MsgBox** in an expression. To omit some positional arguments, you must include the corresponding comma delimiter.

4. GETOPENFILENAME METHOD

Displays the standard **Open** dialog box and gets a file name from the user without actually opening any files.

a. Syntax

expression **.GetOpenFilename(*FileFilter*, *FilterIndex*, *Title*, *ButtonText*, *MultiSelect*)**

expression Required. An expression that returns an **Application** object.

FileFilter Optional **Variant**. A string specifying file filtering criteria.

This string consists of pairs of file filter strings followed by the MS-DOS wildcard file filter specification, with each part and each pair separated by commas. Each separate pair is listed in the **Files of type** drop-down list box. For example, the following string specifies two file filters—text and addin: "Text Files (*.txt),*.txt,Add-In Files (*.xla),*.xla".

To use multiple MS-DOS wildcard expressions for a single file filter type, separate the wildcard expressions with semicolons; for example, "Visual Basic Files (*.bas; *.txt),*.bas;*.txt".

If omitted, this argument defaults to "All Files (*.*),*.*".

FilterIndex Optional **Variant**. Specifies the index numbers of the default file filtering criteria, from 1 to the number of filters specified in ***FileFilter***. If this argument is omitted or greater than the number of filters present, the first file filter is used.

Title Optional **Variant**. Specifies the title of the dialog box. If this argument is omitted, the title is "Open."

ButtonText Optional **Variant**. Macintosh only.

MultiSelect Optional **Variant**. **True** to allow multiple file names to be selected. **False** to allow only one file name to be selected. The default value is **False**

b. Remarks

This method returns the selected file name or the name entered by the user. The returned name may include a path specification. If ***MultiSelect*** is **True**, the return value is an array of the selected file names (even if only one filename is selected). Returns **False** if the user cancels the dialog box.

This method may change the current drive or folder.

c. Example

This example displays the **Open** dialog box, with the file filter set to text files. If the user chooses a file name, the code displays that file name in a message box.

```
fileToOpen = Application.GetOpenFilename("Text Files (*.txt), *.txt")

If fileToOpen <> False Then

    MsgBox "Open " & fileToOpen

End If
```

5. GETSAVEASFILENAME METHOD

Displays the standard **Save As** dialog box and gets a file name from the user without actually saving any files.

a. Syntax

expression **.GetSaveAsFilename(*InitialFilename*, *FileFilter*, *FilterIndex*, *Title*, *ButtonText*)**

expression Required. An expression that returns an **Application** object.

InitialFilename Optional **Variant**. Specifies the suggested file name. If this argument is omitted, Microsoft Excel uses the active workbook's name.

FileFilter Optional **Variant**. A string specifying file filtering criteria.

This string consists of pairs of file filter strings followed by the MS-DOS wildcard file filter specification, with each part and each pair separated by commas. Each separate pair is listed in the **Files of type** drop-down list box. For example, the following string specifies two file filters, text and addin: "Text Files (*.txt), *.txt, Add-In Files (*.xla), *.xla".

To use multiple MS-DOS wildcard expressions for a single file filter type, separate the wildcard expressions with semicolons; for example, "Visual Basic Files (*.bas; *.txt),*.bas;*.txt".

If omitted, this argument defaults to "All Files (*.*),*.*".

FilterIndex Optional **Variant**. Specifies the index number of the default file filtering criteria, from 1 to the number of filters specified in ***FileFilter***. If this argument is omitted or greater than the number of filters present, the first file filter is used.

Title Optional **Variant**. Specifies the title of the dialog box. If this argument is omitted, the default title is used.

ButtonText Optional **Variant**. Macintosh only.

b. Remarks

This method returns the selected file name or the name entered by the user. The returned name may include a path specification. Returns **False** if the user cancels the dialog box.

This method may change the current drive or folder.

c. Example

This example displays the **Save As** dialog box, with the file filter set to text files. If the user chooses a file name, the example displays that file name in a message box.

```
fileSaveName = Application.GetSaveAsFilename( fileFilter:="Text Files
(*.txt), *.txt")

If fileSaveName <> False Then

    MsgBox "Save as " & fileSaveName

End If
```

6. APPLICATION.FILEDIALOG

Returns a FileDialog object representing an instance of the file dialog.

a. Syntax

expression .FileDialog(fileDialogType)

expression A variable that represents an Application object.

b. Parameters

Name	Required/Optional	Data Type	Description
fileDialogType	Required	MsoFileDialogType	The type of file dialog.

c. Remarks

MsoFileDialogType can be one of these MsoFileDialogType constants:

- msoFileDialogFilePicker. Allows user to select a file.
- msoFileDialogFolderPicker. Allows user to select a folder.
- msoFileDialogOpen. Allows user to open a file.
- msoFileDialogSaveAs. Allows user to save a file.

d. Example

In this example, Microsoft Excel opens the file dialog allowing the user to select one or more files. Once these files are selected, Excel displays the path for each file in a separate message.

```
Sub UseFileDialogOpen()

   Dim lngCount As Long

   ' Open the file dialog

   With Application.FileDialog(msoFileDialogOpen)

      .AllowMultiSelect = True

      .Show

      ' Display paths of each file selected

      For lngCount = 1 To .SelectedItems.Count
```

```
            MsgBox .SelectedItems(lngCount)

        Next lngCount

    End With

End Sub
```

7. COMMANDBARS.EXECUTEMSO METHOD

Executes the control identified by the idMso parameter.

a. Syntax

```
expression.ExecuteMso(idMso)
```

expression   An expression that returns a CommandBars object.

b. Parameters

Name	Required/Optional	Data Type	Description
idMso	Required	String	Identifier for the control.

c. Remarks

This method is useful in cases where there is no object model for a particular command. Works on controls that are built-in buttons, toggleButtons and splitButtons. On failure it returns E_InvalidArg for an invalid IdMso, and E_Fail for controls that are not enabled or not visible.

d. Example

The following sample executes the Copy button.

```
Application.CommandBars.ExecuteMso(“Copy”)
```

8. USERFORM BASICS

a. How to display a UserForm

The syntax that is used to display a UserForm programmatically is the following:

UserFormName.Show

To display a UserForm that is named UserForm1, use the following code:

```
UserForm1.Show
```

You can load a UserForm into memory without actually displaying it. It may take a complex UserForm several seconds to appear. Because you can preload a UserForm into memory, you can decide when to incur this overhead. To load UserForm1 into memory without displaying it, use the following code:

```
Load UserForm1
```

To display the UserForm, you must use the **Show** method that was previously shown.

b. How to temporarily hide a UserForm

If you want to temporarily hide a UserForm, use the **Hide** method. You may want to hide a UserForm if your application involves moving between UserForms. To hide a UserForm, use the following code:

```
UserForm1.Hide
```

c. How to remove a UserForm from memory

To remove a UserForm from memory, use the **Unload** statement. To unload a UserForm that is named UserForm1, use the following code:

```
Unload UserForm1
```

If you unload a UserForm in an event procedure that is associated with a UserForm or that is associated with a control on a UserForm (for example, you click a **CommandButton** control), you can use the "Me" keyword instead of the name of the UserForm. To use the "Me" keyword to unload a UserForm, use the following code:

```
Unload Me
```

d. How to use UserForm events

UserForms support many predefined events that you can attach VBA procedures to. When the event occurs, the procedure that you attached to the event runs. A single action that is performed by a user can initiate multiple events. Among the most frequently used events for a UserForm are the **Initialize** event, the **Click** event, and the **Terminate** event.

Note A Visual Basic module that contains an event procedure may be referred to as a module "behind" the UserForm. A module that contains event procedures is not visible in the **Modules** collection of the Microsoft Project Explorer window of the Visual Basic Editor. You must double-click the body of a UserForm to view the UserForm Code module.

e. How to trap UserForm events

To trap UserForm events, follow these steps:

1. Create a new workbook in Excel.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Double-click the **UserForm** to display the Code window for the UserForm.
5. In the module, type the following code:

```
Private Sub UserForm_Click()

    Me.Height = Int(Rnd * 500)
    Me.Width = Int(Rnd * 750)

End Sub

Private Sub UserForm_Initialize()

    Me.Caption = "Events Events Events!"
    Me.BackColor = RGB(10, 25, 100)

End Sub

Private Sub UserForm_Resize()

    msg = "Width: " & Me.Width & Chr(10) & "Height: " & Me.Height
    MsgBox prompt:=msg, Title:="Resize Event"

End Sub


Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)

    msg = "Now Unloading " & Me.Caption
    MsgBox prompt:=msg, Title:="QueryClose Event"

End Sub

Private Sub UserForm_Terminate()
```

```
    msg = "Now Unloading " & Me.Caption
    MsgBox prompt:=msg, Title:="Terminate Event"

End Sub
```

6. On the **Run** menu, click **Run Sub/UserForm**.

When the UserForm is first loaded, the macro uses the **Initialize** event to change the **Caption** property of the UserForm to "Events Events Events!" and the **BackColor** property to dark blue.

When you click the UserForm, you initiate the **Click** event. The **Click** event resizes the UserForm. Because you created a procedure for the **Resize** event, you receive two message boxes after you click the UserForm. The **Resize** event occurs two times because the code behind the **Click** event changes both the **Width** property and the **Height** property of the UserForm.

Closing the UserForm initiates the **QueryClose** event. The **QueryClose** event displays a message box that contains the caption that you gave the UserForm in the code for the **Initialize** event. You can use the **QueryClose** event when you want to perform a certain set of actions if the user closes the UserForm.

The **Terminate** event then generates a message box that states that the caption of the UserForm is UserForm1. The **Terminate** event occurs after the UserForm is removed from memory and the caption of the UserForm returns to its original state.

f. How to prevent a UserForm from being closed by using the Close button

When you run a UserForm, a **Close** button is added to the upper-right corner of the UserForm window. If you want to prevent the UserForm from being closed by using the **Close** button, you must trap the **QueryClose** event.

The **QueryClose** event occurs just before the UserForm is unloaded from memory. Use the ***CloseMode*** argument of the**QueryClose** event to determine how the UserForm is closed. The **vbFormControlMenu** value for the ***CloseMode*** argument indicates that the **Close** button was clicked. To keep the UserForm active, set the ***Cancel*** argument of the **QueryClose** event to**True**. To use the **QueryClose** event to prevent a UserForm from being closed by using the **Close** button, follow these steps:

1. Create a new workbook in Excel.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.

4. Add a **CommandButton** control to the UserForm.
5. Double-click the **UserForm** to display the Code window for the UserForm.
6. In the Code window, type the following code:

```
Private Sub CommandButton1_Click()

    Unload Me

End Sub

Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)

    IF CloseMode = vbFormControlMenu Then
        Cancel = True
        Me.Caption = "Click the CommandButton to close Me!"
    End If

End Sub
```

7. On the **Run** menu, click **Run Sub/UserForm**.

The UserForm is not closed when you click the **Close** button. You must click the **CommandButton** control to close the UserForm.

9. USER CONTROLS

Excel includes fifteen different controls that you can use on UserForms. This section contains various examples that use these controls programmatically.

Note The VBA code that is included in this article does not contain examples that affect all the properties and events for the controls. If you have to, you can use the Properties window to see a list of the properties that are available for a control. To see a list of properties, on the **View** menu, click **Properties Window**.

a. How to use design mode to edit controls

When you use the Visual Basic Editor to design a dialog box, you are using design mode. In design mode, you can edit controls and you can change the properties of a control on a UserForm in the Properties window. To display the Properties window, on the **View** menu, click **Properties Window**.

Note Controls do not respond to events while you are in design mode. When you run a dialog box to display it the way that users see it, the program is in run mode. Changes that you make to the properties of a control in run mode are not retained when the UserForm is unloaded from memory.

Note Controls do respond to events in run mode.

b. How to refer to controls on a UserForm

How you refer to controls programmatically depends on the type of Visual Basic module sheet where you run the code. If the code is running from a General module, the syntax is the following:

UserFormName.Controlname.Property = Value

For example, if you want to set the **Text** property of a **TextBox** control that is named **TextBox1** on a UserForm that is named UserForm1 to the value of **Bob**, use the following code:

```
UserForm1.TextBox1.Text = "Bob"
```

If the code is in a procedure that is initiated by an event of a control or by the UserForm, you do not have to refer to the name of the UserForm. Instead, use the following code:

```
TextBox1.Text = "Bob"
```

When you attach code to an object, the code is attached to one of the events of that object. In many of the examples in this article, you attach code to the **Click** event of the **CommandButton** object.

c. Label controls

Label controls are mainly used to describe other controls on a UserForm. A **Label** control cannot be edited by the user while the UserForm is running. Use the **Caption** property to set or to return the text in a **Label** control. Other frequently used properties for formatting a **Label** control include the **Font** property and the **ForeColor** property.

To use the **WITH** statement to change the properties of a **Label** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **Label** control to the UserForm.
5. Add a **CommandButton** control to the UserForm.
6. Double-click the **CommandButton** control to open the Code window for the UserForm.
7. In the Code window, type the following code for the **CommandButton1 Click** event:

```
Private Sub CommandButton1_Click()

    With Label1
        ' Set the text of the label.
        .Caption = "This is Label Example 1"
        ' Automatically size the label control.
        .AutoSize = True
        .WordWrap = False
        ' Set the font used by the Label control.
        .Font.Name = "Times New Roman"
        .Font.Size = 14
        .Font.Bold = True
        ' Set the font color to blue.
        .ForeColor = RGB(0, 0, 255)
    End With

End Sub
```

8. On the **Run** menu, click **Run Sub/UserForm**.
9. Click the **CommandButton**.

The text "This is Label Example 1" appears on the **Label** control in bold Times New Roman with a font size of 14.

d. TextBox controls

TextBox controls are frequently used to gather input from a user. The **Text** property contains the entry that is made in a **TextBox** control.

If you set the **PasswordChar** property of a **TextBox** control, it becomes a "masked-edit" control. Every character that is typed in the **TextBox** control is replaced visually by the character that you specify. To use a **TextBox** control to validate a password, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **TextBox** control to the UserForm.
5. On the **View** menu, click **Properties** to make the Properties window visible.
6. In the **PasswordChar** property of the **TextBox** control, type *.

   Note You are changing the value to an asterisk.
7. Add a **CommandButton** control to the UserForm.
8. Double-click the **CommandButton** control to open the Code window for the UserForm.
9. In the Code window, type the following code for the **CommandButton1 Click** event:

```
Private Sub CommandButton1_Click()

    If TextBox1.Text <> "userform" Then
        MsgBox "Password is Incorrect. Please reenter."
        TextBox1.Text = ""
        TextBox1.SetFocus
    Else
        MsgBox "Welcome!"
        Unload Me
    End If

End Sub
```

10. On the **Run** menu, click **Run Sub/UserForm**.
11. Type the password **userform** in the **TextBox** control.
12. Click the **CommandButton** control.

For this example, the password is "userform". If you type an incorrect password, you receive a message box that states that your password is incorrect, the **TextBox** control is cleared, and then you can retype the password. When you type a correct password, you receive a welcome message, and the UserForm is closed.

e. CommandButton controls

You can use a **CommandButton** control to start a VBA procedure. The VBA procedure is typically attached to the **Click** event of the **CommandButton** control. To use a **CommandButton** control that runs a procedure when the **Click** event occurs, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.

3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **CommandButton** control to the UserForm.
5. Double-click the **CommandButton** control to display the Code window for the UserForm.
6. In the Code window, type the following code:

```
Private Sub CommandButton1_Click()

   red = Int(Rnd * 255)
   green = Int(Rnd * 255)
   blue = Int(Rnd * 255)
   CommandButton1.BackColor = RGB(red, green, blue)

End Sub
```

7. On the **Run** menu, click **Run Sub/UserForm**.

The background color of the **CommandButton1** control changes every time that you click it.

f. ListBox controls

The purpose of the **ListBox** control is to present the user with a list of items to select from. You can store the item list for a**ListBox** control on an Excel worksheet. To populate a **ListBox** control with a range of cells on a worksheet, use the **RowSource**property. When you use the **MultiSelect** property, you can set up a **ListBox** control to accept multiple selections.

How to obtain the currently selected item from the ListBox control

Use the **Value** property of a **ListBox** control to return the currently selected item. To return the currently selected item in a single select **ListBox** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. In cells A1:A5 on Sheet1, type the values that you want to use to populate the **ListBox** control.
3. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
4. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
5. Add a **ListBox** control to the UserForm.
6. Double-click the **ListBox** control to display the Code window for the **ListBox** control.
7. In the Code window, type the following code for the **ListBox1 Click** event:

```
Private Sub ListBox1_Click()

   MsgBox ListBox1.Value

End Sub
```

8. On the **Run** menu, click **Run Sub/UserForm**.

When you click an item in the list, a message box appears with the currently selected item.

How to obtain the selected items in a multiple select ListBox control

To determine the items that are selected in a multiple select **ListBox** control, you must loop through all the items in the list, and then query the **Selected** property. To return the currently selected items in a multiple select **ListBox** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. In cells A1:A5 on Sheet1, type the values that you want to use to populate the **ListBox** control.
3. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
4. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
5. Add a **ListBox** control to the UserForm.
6. On the **View** menu, click **Properties** to see the Properties window.
7. Type the values that are indicated for the following **ListBox** control properties:

Property	Value
MultiSelect	1 – frmMultiSelectMulti
RowSource	Sheet1!A1:A8

8. Add a **CommandButton** control to the UserForm.
9. Double-click the **CommandButton** control to display the Code window for the UserForm.
10. In the Code window, type the following code for the **CommandButton1 Click** event:

```
Sub CommandButton1_Click ()

     ' Loop through the items in the ListBox.
     For x = 0 to ListBox1.ListCount - 1

        ' If the item is selected...
        If ListBox1.Selected(x) = True Then

           ' display the Selected item.
           MsgBox ListBox1.List(x)
        End If
     Next x
End Sub
```

11. On the **Run** menu, click **Run Sub/UserForm**.
12. Select one or more items in the list.
13. Click **CommandButton1**.

After you click **CommandButton1**, every item that you selected in the **ListBox** control appears in a separate message box. After all the selected items appear in a message box, the UserForm is automatically closed.

How to use the RowSource property to populate a ListBox control with cells on a worksheet

To use the **RowSource** property to populate a **ListBox** control from a range of cells on a worksheet, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. In cells A1:A5 on Sheet1, type the values that you want to use to populate the **ListBox** control.
3. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
4. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
5. Add a **ListBox** control to the UserForm.

6. Add a **CommandButton** control to the UserForm.
7. Double-click the **CommandButton** control to display the Code window for the UserForm.
8. In the Code window, type the following code for the **CommandButton1 Click** event:

```
Private Sub CommandButton1_Click()
   ListBox1.RowSource = "=Sheet1!A1:A5"
End Sub
```

9. On the **Run** menu, click **Run Sub/UserForm**.

   NoteListBox1 does not contain any values.
10. Click **CommandButton1**.

ListBox1 is populated with the values in cells A1:A5 on Sheet1.

How to populate a ListBox control with values in an array

This example shows you how to populate a **ListBox** control with an array variable. You must assign the values from the array to the **ListBox** control one item at a time. Typically, this process requires that you use a looping structure, such as a **For...Next**loop. To populate a **ListBox** control with an array variable, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **ListBox** control to the UserForm.
5. On the **Insert** menu, click **Module** to insert a module sheet.
6. In the Code window, type the following code:

```
Sub PopulateListBox()

   Dim MyArray As Variant
   Dim Ctr As Integer
   MyArray = Array("Apples", "Oranges", "Peaches", "Bananas", "Pineapples")

   For Ctr = LBound(MyArray) To UBound(MyArray)
      UserForm1.ListBox1.AddItem MyArray(Ctr)
   Next

   UserForm1.Show

End Sub
```

7. On the **Tools** menu, click **Macros**, click **PopulateListBox**, and then click **Run**.

The **PopulateListBox** procedure builds a simple array, and then adds the items in the array to the **ListBox** control by using the**AddItem** method. Then, the UserForm appears.

How to use a horizontal range of cells on a worksheet to populate a ListBox control

If you set the **RowSource** property of a **ListBox** control to a horizontal range of cells, only the first value appears in the **ListBox**control.

To populate a **ListBox** control from a horizontal range of cells by using the **AddItem** method, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. In cells A1:E1 on Sheet1, type the values that you want to use to populate the **ListBox** control.
3. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
4. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
5. Add a **ListBox** control to the UserForm.
6. On the **Insert** menu, click **Module** to insert a module sheet.
7. In the Code window, type the following code:

```
Sub PopulateListWithHorizontalRange()

   For Each x In Sheet1.Range("A1:E1")
      UserForm1.ListBox1.AddItem x.Value
   Next

   UserForm1.Show

End Sub
```

8. On the **Tools** menu, click **Macros**, click **PopulateListWithHorizontalRange**, and then click **Run**.

The macro procedure loops through cells A1:E5 on Sheet1, adding the values to **ListBox1** one at a time.

NoteListBox1 is not bound to cells A1:E5 on Sheet1.

How to return multiple values from a ListBox control that is bound to multiple columns of data

You can format **ListBox** controls to display more than one column of data. This means that the **ListBox** control displays more than one item on each line of the list. To return multiple values from the selected item in the list, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. Type the following data in the cells that are indicated on Sheet1:

A1: Year	B1: Region	C1: Sales
A2: 1996	B2: North	C2: 140
A3: 1996	B3: South	C3: 210
A4: 1997	B4: North	C4: 190
A5: 1997	B5: South	C5: 195

3. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
4. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
5. Add a **Label** control to the UserForm.

6. Add a **ListBox** control to the UserForm.
7. Right-click the **ListBox**, and then click **Properties**.
8. Type or select the values that are indicated for the following properties of the **ListBox** control as listed in the following table:

Property	Value
BoundColumn	1
ColumnCount	3
ColumnHeads	True
RowSource	Sheet1!A2:A5

9. Double-click the **ListBox** control to display the Code window for the **ListBox** control.
10. In the Code window, type the following code:

```
Private Sub ListBox1_Change()

    Dim SourceData As Range
    Dim Val1 As String, Val2 As String, Val3 As String

    Set SourceRange = Range(ListBox1.RowSource)

    Val1 = ListBox1.Value
    Val2 = SourceRange.Offset(ListBox1.ListIndex, 1).Resize(1, 1).Value
    Val3 = SourceRange.Offset(ListBox1.ListIndex, 2).Resize(1, 1).Value

    Label1.Caption = Val1 & " " & Val2 & " " & Val3

End Sub
```

11. On the **Run** menu, click **Run Sub/UserForm**.

When you click an entry in the **ListBox** control, the label changes to display all three of the items in that entry.

How to remove all the items from a ListBox control that is bound to a worksheet

To remove all the items from a **ListBox** control that is bound to a worksheet, clear the value that is stored in the **RowSource**property. To remove items from a **ListBox** control that is bound to a worksheet, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. In cells A1:A5 on Sheet1, type the values that you want to use to populate the **ListBox** control.
3. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
4. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
5. Add a **ListBox** control to the UserForm.
6. Right-click the **ListBox** control, and then click **Properties**.
7. In the **RowSource** property, type **Sheet1!A1:A5**.
8. Add a **CommandButton** control to the UserForm.
9. Double-click the **CommandButton** control to display the Code window for the **CommandButton** control.
10. In the Code window, type the following code for the **CommandButton1 Click** event:

```
Private Sub CommandButton1_Click()

    ListBox1.RowSource = ""

End Sub
```

11. On the **Run** menu, click **Run Sub/UserForm**.

The **ListBox** control that you added to the UserForm is populated with the values that you entered on Sheet1.

12. Click **CommandButton1**.

All the items are removed from **ListBox1**.

How to remove all the items from a ListBox control that is not bound to a worksheet

There is no single VBA command that removes all the items from a **ListBox** control if the list is not bound to a worksheet. To remove all the items from a **ListBox** control that is populated from a Visual Basic array, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **ListBox** control to the UserForm.
5. On the **Insert** menu, click **Module** to insert a module sheet.
6. In the Code window, type the following code:

```
Sub PopulateListBox()

   Dim MyArray As Variant
   Dim Ctr As Integer
   MyArray = Array("Apples", "Oranges", "Peaches", "Bananas", "Pineapples")

   For Ctr = LBound(MyArray) To UBound(MyArray)
      UserForm1.ListBox1.AddItem MyArray(Ctr)
   Next

   UserForm1.Show

End Sub
```

7. Add a **CommandButton** control to the UserForm.
8. Double-click the **CommandButton** control to display the Code window for the **CommandButton** control.
9. In the Code window, type the following code for the **CommandButton1 Click** event:

```
Private Sub CommandButton1_Click()

   For i = 1 To ListBox1.ListCount
      ListBox1.RemoveItem 0
   Next I
End Sub
```

10. On the **Tools** menu, click **Macros**, click **PopulateListBox**, and then click **Run**.

    The **ListBox** control is populated, and then the UserForm appears.
11. Click **CommandButton1**.

All the items are removed from **ListBox1**.

g. ComboBox controls

You can use the **ComboBox** control as a drop-down list box, or as a combo box where you can select a value in a list or type a new value. The **Style** property determines if the **ComboBox** control acts as a drop-down list box or a combo box.

Note All the examples in the previous section for the **ListBox** control can also be applied to the **ComboBox** control, except for the "How to obtain the selected items in a multiple select ListBox control" example.

How to add a new item to the list if the ComboBox control is not bound to a worksheet

When you type a value that is not already in the list in the **ComboBox** control, you may want to add the new value to the list. To add the new value that you typed in the **ComboBox** control if the **ComboBox** control is not bound to the worksheet, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **ComboBox** control to the UserForm.
5. On the **Insert** menu, click **Module** to insert a module sheet.
6. In the Code window, type the following code:

```
Sub PopulateComboBox()

    Dim MyArray As Variant
    Dim Ctr As Integer
    MyArray = Array("Apples", "Oranges", "Peaches", "Bananas", "Pineapples")

    For Ctr = LBound(MyArray) To Ubound(MyArray)
        UserForm1.ComboBox1.AddItem MyArray(Ctr)
    Next

    UserForm1.Show

End Sub
```

7. Add a **CommandButton** control to the UserForm.
8. Double-click the **CommandButton** control to display the Code window for the **CommandButton** control.
9. In the Code window, type the following code for the **CommandButton1 Click** event:

```
Private Sub CommandButton1_Click()

    Dim listvar As Variant

    listvar = ComboBox1.List

    On Error Resume Next
    ' If the item is not found in the list...
    If IsError(WorksheetFunction.Match(ComboBox1.Value, listvar, 0)) Then
        ' add the new value to the list.
        ComboBox1.AddItem ComboBox1.Value
    End If
End Sub
```

10. On the **Tools** menu, click **Macros**, click **PopulateListBox**, and then click **Run**.

    The **ComboBox** control is populated, and then the UserForm appears.

11. In the **ComboBox** control, type **Mangoes** (or any value that is not already in the list).
12. Click **CommandButton1**.

The new value that you typed now appears at the end of the list.

How to add a new item to the list if the ComboBox control is bound to a worksheet

When a user types a value that is not already in the list in the **ComboBox** control, you may want to add the new value to the list. To add the new value that you typed in the **ComboBox** control to the list, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. In cells A1:A5 on Sheet1, type the values that you want to use to populate the **ComboBox** control.
3. Select cells A1:A5 on Sheet1.
4. On the **Insert** menu, point to **Name**, and then click **Define**.

   In the **Names in workbook** box, type **ListRange** , and then click **OK**. This creates the defined name **ListRange**. You can use the defined name **ListRange** to bind the **RowSource** property of the **ComboBox** control to the worksheet.
5. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
6. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
7. Add a **ComboBox** control to the UserForm.
8. In the **Properties** for **ComboBox1**, type **Sheet1!ListRange** as the **RowSource** property.
9. Add a **CommandButton** control to the UserForm.
10. Double-click the **CommandButton** control to display the Code window for the **CommandButton** control.
11. In the Code window, type the following code for the **CommandButton1** **Click** event:

```
Private Sub CommandButton1_Click()

    Dim SourceData As Range
    Dim found As Object

    Set SourceData = Range("ListRange")
    Set found = Nothing
    ' Try to find the value on the worksheet.
    Set found = SourceData.Find(ComboBox1.Value)

    ' If the item is not found in the list...
    If found Is Nothing Then
        ' redefine ListRange.
        SourceData.Resize(SourceData.Rows.Count + 1, 1).Name = "ListRange"
        ' Add the new item to the end of the list on the worksheet.
        SourceData.Offset(SourceData.Rows.Count, 0).Resize(1, 1).Value _
         = ComboBox1.Value
        ' Reset the list displayed in the ComboBox.
        ComboBox1.RowSource = Range("listrange").Address(external:=True)
    End If
End Sub
```

12. On the **Run** menu, click **Run Sub/UserForm**.

    The UserForm appears on Sheet1.
13. In the **ComboBox** control, type a value that is not already in the list.

14. Click **CommandButton1**.

The new item that you typed in the **ComboBox** control is added to the list, and the list that the **ComboBox** control is bound to is expanded to include cells A1:A6.

How to display the list of a ComboBox control when the UserForm appears

Sometimes, it may be useful to display the list of a **ComboBox** control when a UserForm first appears. The following example uses the **Activate** event of the UserForm. To display the list of a **ComboBox** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. In cells A1:A5 on Sheet1, type the values that you want to use to populate the **ComboBox** control.
3. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
4. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
5. Add a **ComboBox** control to the UserForm.
6. In the **Properties** for **ComboBox1**, type **Sheet1!A1:A5** as the **RowSource** property.
7. Double-click the **UserForm** to display the Code window for the UserForm.
8. In the Code window, type the following code for the **CommandButton Click** event:

```
Private Sub UserForm_Activate()

        ComboBox1.DropDown
End Sub
```

9. On the **Run** menu, click **Run Sub/UserForm**.

The UserForm appears on Sheet1, and you can see the list for **ComboBox1**.

How to display the list of one ComboBox control when you make a selection in another ComboBox control

To automatically display the list of one **ComboBox** control when a choice is made in another **ComboBox** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. In cells A1:A10 on Sheet1, type the values that you want to use to populate the **ComboBox** control.
3. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
4. On the **Insert** menu, click **Module**.
5. In the Code window for the module, type the following code:

```
Sub DropDown_ComboBox()

   UserForm1.ComboBox2.DropDown

End Sub
```

6. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
7. Add a **ComboBox** control to the UserForm.
8. In the **Properties** for **ComboBox1**, type **Sheet1!A1:A5** as the **RowSource** property.

9. Double-click the **ComboBox** control to open the Code window for the **ComboBox** control.
10. In the Code window for the **ComboBox** control, type the following code for the **ComboBox Click** event:

```
Private Sub ComboBox1_Click()

   Application.OnTime Now, "DropDown_ComboBox"

End Sub
```

11. Add a second **ComboBox** control to the UserForm.
12. In the **Properties** for **ComboBox2**, type **Sheet1!A6:A10** as the **RowSource** property.
13. On the **Run** menu, click **Run Sub/UserForm**.

When you click an item in the **ComboBox1** list , the list for **ComboBox2** automatically appears.

h. Frame control

Use a **Frame** control to group logically related items in a UserForm. **Frame** controls are frequently used to group**OptionButton** controls.

How to loop through all the controls on a Frame control

To use a **For Each...Next** loop to access all the controls in a **Frame** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **Frame** control to the UserForm.
5. Add an **OptionButton** control to the **Frame** control.

   Repeat this step to add two more **OptionButton** controls in the **Frame** control.
6. Double-click the **Frame** control to open the Code window for the **Frame** control.
7. In the Code window, type the following code for the **Frame Click** event:

```
Private Sub Frame1_Click()

   Dim Ctrl As Control

   For Each Ctrl In Frame1.Controls
      Ctrl.Enabled = Not Ctrl.Enabled
   Next
End Sub
```

8. On the **Run** menu, click **Run Sub/UserForm**.
9. In the UserForm, click the **Frame** control.

The first time that you click the **Frame** control, all the controls in the **Frame** control are unavailable. If you click the **Frame**control again, the controls are available again.

i. OptionButton control

You can use groups of **OptionButton** controls to make one selection among a group of options. You can use either of the following techniques to group **OptionButton** controls:

- **Frame** control
- **GroupName** property

Note The **On** value, the **Yes** value, and the **True** value indicate that an **OptionButton** is selected. The **Off** value, the **No**value, and the **False** value indicate that an **OptionButton** is not selected.

How to determine the OptionButton control that is selected when the OptionButton controls are on a Frame control

When you group **OptionButtons** controls by using a **Frame** control, you can determine the **OptionButton** control that is selected by looping through all the controls in the **Frame** control and checking the **Value** property of each control. To determine the **OptionButton** control that is selected, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **Frame** control to the UserForm.
5. Add an **OptionButton** control to the **Frame** control.

   Repeat this step to add two more **OptionButton** controls in the **Frame** control.
6. Add a **CommandButton** control on the UserForm outside the **Frame** control.
7. Double-click the **CommandButton** control to display the Code window for the UserForm.
8. In the Code window, type the following code for the **CommandButton1 Click** event:

```
Private Sub CommandButton1_Click()

    For Each x In Frame1.Controls
        If x.Value = True Then
            MsgBox x.Caption
        End If
    Next
End Sub
```

9. On the **Run** menu, click **Run Sub/UserForm**.
10. In the **UserForm**, click one **OptionButton** control, and then click **CommandButton1**.

A message box appears that contains the caption of the currently selected **OptionButton** control.

How to determine the OptionButton control that is selected

The purpose of the following example is to determine the **OptionButton** control that is selected in Group1. To create a UserForm that has two groups of **OptionButton** controls, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **Frame** control to the UserForm.
5. Add an **OptionButton** control in the **Frame** control.

   Repeat this step to add two more **OptionButton** controls in the **Frame** control.
6. For each **OptionButton** control, type **Group1** in the **GroupName** property.
7. Repeat steps 4 and 5 to create a second **Frame** control that contains three **OptionButton** controls.
8. For each **OptionButton** control in the second **Frame** control, type **Group2** in the **GroupName** property.
9. Add a **CommandButton** control on the UserForm outside the **Frame** controls.
10. Double-click the **CommandButton** control to display the Code window for the UserForm.
11. In the Code window, type the following code for the **CommandButton1 Click** event:

```
Private Sub CommandButton1_Click()

    Dim x As Control

    ' Loop through ALL the controls on the UserForm.
    For Each x In Me.Controls
        ' Check to see if "Option" is in the Name of each control.
        If InStr(x.Name, "Option") Then
            ' Check Group name.
            If x.GroupName = "Group1" Then
                ' Check the status of the OptionButton.
                If x.Value = True Then
                    MsgBox x.Caption
                    Exit For
                End If
            End If
        End If
    Next
End Sub
```

12. On the **Run** menu, click **Run Sub/UserForm**.
13. In the UserForm, click one **OptionButton** control in Group1, and then click **CommandButton1**.

A message box appears that contains the caption of the **OptionButton** control that is currently selected.

j. CheckBox control

You can use a **CheckBox** control to indicate a true or false value. A **CheckBox** control that appears with a check mark in it indicates a value of **True**. A **CheckBox** that appears with no check mark indicates a value of **False**. If the value of the **TripleState** property is **True**, a **CheckBox** control can also have a value of **Null**. A **CheckBox** control that has a value of **Null** appears to be unavailable.

Note The **On** value, the **Yes** value, and the **True** value indicate that a **CheckBox** control is selected. The **Off** value, the **No** value, and the **False** value indicate that a **CheckBox** control is cleared.

How to check the value of a CheckBox control

To use the **Value** property to return the current value of a **CheckBox** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **CheckBox** control to the UserForm.
5. In the Properties list for **CheckBox1**, select **True** as the **TripleState** property.
6. Double-click the **CheckBox** control to display the Code window for the **CheckBox** control.
7. In the Code window, type the following code for the **CheckBox1 Change** event:

```
Private Sub CheckBox1_Change()

   Select Case CheckBox1.Value
      Case True
         CheckBox1.Caption = "True"
      Case False
         CheckBox1.Caption = "False"
      Case Else
         CheckBox1.Caption = "Null"
   End Select
End Sub
```

8. On the **Run** menu, click **Run Sub/UserForm**.

When you click the **CheckBox** control, the caption of the **CheckBox** control changes to reflect the current value.

k. ToggleButton control

A **ToggleButton** control has the same appearance as a **CommandButton** control until you click it. When you click a **ToggleButton** control, it appears to be pressed or pushed down. The **Value** property of a **ToggleButton** control is **True** when the button is selected and **False** when the button is not selected. If the value of the **TripleState** property is **True**, a **ToggleButton** control can also have a value of **Null**. A **ToggleButton** control that has a value of **Null** appears to be unavailable.

Note The **On** value, the **Yes** value, and the **True** value indicate that a **ToggleButton** control is selected. The **Off** value, the **No**value, and the **False** value indicate that a **ToggleButton** control is not selected.

How to obtain the value of a ToggleButton control

To obtain the value of a **ToggleButton** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.

4. Add a **ToggleButton** control on the UserForm.
5. Add a **Label** control to the UserForm.
6. Double-click the **ToggleButton** control to open the Code window for the **ToggleButton** control.
7. In the Code window, type the following code for the **ToggleButton1Click** event:

```
Private Sub ToggleButton1_Click()

    If ToggleButton1.Value = True Then
        ' Set UserForm background to Red.
        Me.BackColor = RGB(255, 0, 0)
    Else
        ' Set UserForm background to Blue.
        Me.BackColor = RGB(0, 0, 255)
    End If

End Sub
```

8. On the **Run** menu, click **Run Sub/UserForm**.

When you click the **ToggleButton** control, the background color of the UserForm changes.

How to create a group of mutually exclusive ToggleButton controls

This example uses the **MouseUp** event to set a variable and calls the **ExclusiveToggleButtons** procedure. The**ExclusiveToggleButtons** procedure determines the **ToggleButton** control that is selected, and then cancels the others. To create a group of mutually exclusive **ToggleButton** controls, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **Module**.
4. In the Code window for the module, type the following code:

```
' Variable that holds the name of the ToggleButton that was clicked.
Public clicked As String

Sub ExclusiveToggleButtons()

   Dim toggle As Control

   ' Loop through all the ToggleButtons on Frame1.
   For Each toggle In UserForm1.Frame1.Controls

       ' If Name of ToggleButton matches name of ToggleButton
       ' that was clicked...
       If toggle.Name = clicked Then
          '...select the button.
          toggle.Value = True
       Else
          '...otherwise clear the selection of the button.
          toggle.Value = False
       End If
    Next
End Sub
```

5. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.

6. Add a **Frame** control to the UserForm.
7. Add a **ToggleButton** control in the **Frame** control.

   Repeat this step to add two more **ToggleButton** controls in the **Frame** control.
8. Double-click the **Frame** control to display the Code window for the UserForm.
9. In the Code window for the module, type the following code for the **ToggleButton MouseUp** event:

```
Private Sub ToggleButton1_MouseUp(ByVal Button As Integer, _
        ByVal Shift As Integer, ByVal X As Single, ByVal Y As Single)

  clicked = ToggleButton1.Name
  Application.OnTime Now, "ExclusiveToggleButtons"

End Sub

Private Sub ToggleButton2_MouseUp(ByVal Button As Integer, _
        ByVal Shift As Integer, ByVal X As Single, ByVal Y As Single)

   clicked = ToggleButton2.Name
   Application.OnTime Now, "ExclusiveToggleButtons"

End Sub

Private Sub ToggleButton3_MouseUp(ByVal Button As Integer, _
        ByVal Shift As Integer, ByVal X As Single, ByVal Y As Single)

   clicked = ToggleButton3.Name
   Application.OnTime Now, "ExclusiveToggleButtons"
End Sub
```
10. On the **Run** menu, click **Run Sub/UserForm**.

When you click a **ToggleButton** control, the previously selected **ToggleButton** control is canceled.

I. TabStrip control

Use a **TabStrip** control to view different sets of information for a set of controls.

How to control a TabStrip control programmatically

To change the **BackColor** property of an **Image** control based on the tab that is selected, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **TabStrip** control to the UserForm.
5. Add an **Image** control that covers the base of the **TabStrip** control, but that does not cover the tabs.
6. In the Properties pane for Image1, type **&H000000FF&** in the **BackColor** property.
7. Double-click the **TabStrip** control to open the Code window for the **TabStrip** control.
8. In the Code window, type the following code for the **TabStrip1 Change** event:

```
Private Sub TabStrip1_Change()
```

```
    Dim i As Integer

    i = TabStrip1.SelectedItem.Index
    Select Case i
        Case 0
            ' If Tab1 is selected, change the color of Image control to Red.
            Image1.BackColor = RGB(255, 0, 0)
        Case 1
            ' If Tab2 is selected, change the color of Image control to Green.
            Image1.BackColor = RGB(0, 255, 0)
    End Select

End Sub
```

9. On the **Run** menu, click **Run Sub/UserForm**.

The color of the **Image** control changes depending on the page in the **TabStrip** control that is active.

m. MultiPage control

Use a **MultiPage** control to work with a lot of information that can be sorted into several categories. A **MultiPage** control is made up of one or more **Page** objects that each contain a different set of controls. You can set the active page programmatically by setting the **Value** property of the **MultiPage** control.

How to control a MultiPage control programmatically

To add a **MultiPage** control and control it by using a macro, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **MultiPage** control to the UserForm.
5. Add a **Label** control to Page1 on the **MultiPage** control.
6. Add a **TextBox** control to Page1 on the **MultiPage** control.
7. On the **MultiPage** control, click **Page2**, and then repeat steps 5 and 6 to add a **Label** control and a **TextBox** control.
8. Double-click the **MultiPage** control to open the Code window for the **MultiPage** control.
9. In the Code window, type the following code for the **MultiPage1 Change** event:

```
Private Sub MultiPage1_Change()

    Select Case MultiPage1.Value
        ' If activating Page1...
        Case 0
            Label1.Caption = TextBox2.Text
            TextBox1.Text = ""
        ' If activating Page2...
        Case 1
            Label2.Caption = TextBox1.Text
            TextBox2.Text = ""
    End Select

End Sub
```

10. In the Code window, type the following code for the **UserForm Initialize** event:

```
Private Sub UserForm_Initialize()

    ' Force Page1 to be active when UserForm is displayed.
    MultiPage1.Value = 0
    Label1.Caption = ""

End Sub
```

11. On the **Run** menu, click **Run Sub/UserForm**.

    In the **TextBox** control on Page1, type **Test**. When you click the **Page2** tab, **TextBox2** is cleared, and the caption of**Label2** changes to the entry that you made in **TextBox1** on Page1 ("Test").

How to create a wizard interface by using a MultiPage control

When a task requires several incremental steps, a wizard interface can be very effective. You can use the **MultiPage** control to create a wizard interface instead of using multiple UserForms. This example manipulates a **MultiPage** control that has three pages. A procedure that is attached to the **Initialize** event of the UserForm disables Page2 and Page3, and forces Page1 of the**MultiPage** control to be active.

Note When you index the pages of a **MultiPage** control by using the **Pages** collection, the first page in the collection is page zero. This procedure also sets the caption of the **CommandButton** controls and disables the **<Back** button.

Note The procedure that is assigned to the **Click** event of **CommandButton1** controls the functionality of the **<Back** button. The procedure that is assigned to the **Click** event of **CommandButton2** controls the functionality of the **Next>** button. To create a wizard interface by using a **MultiPage** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **MultiPage** control to the UserForm.
5. Right-click the **Page1** tab, and then click **New Page** to add Page3 to the **MultiPage** control.
6. Add a **CommandButton** control on the UserForm that is not on the **MultiPage** control.

   Repeat this step to add a second **CommandButton** control on the UserForm.
7. Double-click the **UserForm** to open the Code window for the UserForm.
8. In the Code window, type the following code for the **UserForm Initialize** event:

```
Private Sub UserForm_Initialize()

    With MultiPage1
        ' The next 2 lines disable Page2 & Page3.
        .Pages(1).Enabled = False
        .Pages(2).Enabled = False
        ' Make Page1 the active page.
        .Value = 0
    End With
    ' Set the caption on the CommandButtons.
    CommandButton1.Caption = "<Back"
    CommandButton1.Enabled = False
    CommandButton2.Caption = "Next>"

End Sub

' Procedure for the "<Back" button
Private Sub CommandButton1_Click()
    Select Case MultiPage1.Value
        Case 1                              ' If Page2 is active...
            With MultiPage1
                .Pages(0).Enabled = True        ' Enable Page1.
                .Value = MultiPage1.Value - 1   ' Move back 1 page.
                .Pages(1).Enabled = False       ' Disable Page2.
            End With
            CommandButton1.Enabled = False      ' Disable Back button.

        Case 2                              ' If Page3 is active...
            With MultiPage1
                .Pages(1).Enabled = True        ' Enable Page2.
                .Value = MultiPage1.Value - 1   ' Move back 1 page.
                .Pages(2).Enabled = False       ' Disable Page3.
   CommandButton2.Caption = "Next>"
            End With
    End Select

End Sub

' Procedure for the "Next>" button
Private Sub CommandButton2_Click()

    Select Case MultiPage1.Value
        Case 0                               ' If Page1 is active...
            With MultiPage1
                .Value = MultiPage1.Value + 1   ' Move forward 1 page.
                .Pages(1).Enabled = True        ' Enable Page2.
                .Pages(0).Enabled = False       ' Disable Page1.
            End With
            CommandButton1.Enabled = True       ' Enable Back button.

        Case 1                               ' If Page2 is active...
            With MultiPage1
                .Value = MultiPage1.Value + 1   ' Move forward 1 page.
                .Pages(2).Enabled = True        ' Enable Page3.
                .Pages(1).Enabled = False       ' Disable Page2.
            End With
            CommandButton2.Caption = "Finish"   ' Change Next button to Finish.

        Case 2                               ' If Page3 is active...
            MsgBox "Finished!"                   ' User is Finished.
            Unload Me                          ' Unload the UserForm.
    End Select

End Sub
```

9. On the **Run** menu, click **Run Sub/UserForm**.

When you click **Next>**, Page2 is activated and the **<Back** button becomes available. When you click **Next>** a second time, Page3 is activated and the caption for **CommandButton2** changes to "Finish".

n. ScrollBar control

You can use a **ScrollBar** control when you want to change the value that is displayed by another control, such as a **Label**control.

How to change a Label control that is based on the value of a ScrollBar control

To change the **Caption** property of a **Label** control to the current setting of the **Value** property of a **ScrollBar** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **ScrollBar** control to the UserForm.
5. Add a **Label** control to the UserForm.
6. Double-click the **ScrollBar** control to open the Code window for the **ScrollBar** control.
7. In the Code window, type the following code for the **ScrollBar1 Change** event:

```
Private Sub ScrollBar1_Change()

   Label1.Caption = ScrollBar1.Value

End Sub
```

8. On the **Run** menu, click **Run Sub/UserForm**.

When you scroll by using the **ScrollBar** control, **Label1** is updated with the current value of the **ScrollBar** control.

o. SpinButton control

A **SpinButton** control, like a **ScrollBar** control, is frequently used to increment or to decrement the value of another control, such as a **Label** control.
The **SmallChange** property determines how much the value of a **SpinButton** control changes when it is clicked.

How to add a SpinButton control that increments or decrements a date that is stored in a TextBox control

To add a **SpinButton** control that increments or decrements a date that is stored in a **TextBox** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **SpinButton** control to the UserForm.
5. Add a **TextBox** control to the UserForm.
6. Double-click the **SpinButton** control to open the Code window for the **SpinButton** control.
7. In the Code window, type the following code for the **SpinButton1 SpinUp** event:

```
Private Sub SpinButton1_SpinUp()
```

```
    TextBox1.Text = DateValue(TextBox1.Text) + 1

End Sub
```

8. In the Code window, type the following code for the **SpinButton1 SpinDown** event:

```
Private Sub SpinButton1_SpinDown()

    TextBox1.Text = DateValue(TextBox1.Text) - 1

End Sub
```

9. In the Code window, type the following code for the **UserForm Initialize** event:

```
Private Sub UserForm_Initialize()

    TextBox1.Text = Date

End Sub
```

10. On the **Run** menu, click **Run Sub/UserForm**.

When the UserForm appears, the current date appears in **TextBox1**. When you click the **SpinButton** control, the date is incremented or decremented by one day.

In this example, if you change the **SmallChange** property of **SpinButton1**, you do not affect the number of days the entry in**TextBox1** is changed by when you click **SpinButton1**. The number of days is determined only by the procedure that you attached to the **SpinUp** event and the **SpinDown** event of **SpinButton1**.

p. RefEdit control

The **RefEdit** control imitates the behavior of the reference boxes that are built into Excel. You can use the **Value** property to obtain the current cell addresses that are stored in a **RefEdit** control.

How to populate a range of cells based on the range that you select by using the RefEdit control

To use the **RefEdit** control to populate cells, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add a **RefEdit** control to the UserForm.
5. Add a **CommandButton** control to the UserForm.
6. Double-click the **CommandButton** control to open the Code window for the **CommandButton** control.
7. In the Code window, type the following code for the **CommandButton1 Click** event:

```
Private Sub CommandButton1_Click()

    Dim MyRange As String
    MyRange = RefEdit1.Value
    Range(MyRange).Value = "test"
    Unload Me
```

```
End Sub
```

8. On the **Run** menu, click **Run Sub/UserForm**.

   The UserForm appears.
9. Click the button in the **RefEdit** control.

   Notice that the UserForm collapses.
10. Select a range of cells such as A1:A5, and then click the button in the **RefEdit** control to expand the UserForm.
11. Click **CommandButton1**.

The UserForm closes and the cells that you selected now contain the word "test".

q. Image control

The purpose of the **Image** control is to display a picture on a UserForm. To assign a picture to an **Image** control at run time, use the **LoadPicture** function.

How to load a picture into an Image control

To insert an **Image** control that prompts you to select a picture to load when you click the **Image** control, follow these steps:

1. Start Excel, and then open a new blank workbook.
2. On the **Tools** menu, point to **Macro**, and then click **Visual Basic Editor**.
3. On the **Insert** menu, click **UserForm** to insert a UserForm in your workbook.
4. Add an **Image** control on the UserForm.
5. Double-click the **Image** control to open the Code window for the **Image** control.
6. In the Code window, type the following code for the **Image1 Click** event:

```
Private Sub Image1_Click()

   Dim fname As String

   ' Display the Open dialog box.
   fname = Application.GetOpenFilename(filefilter:= _
         "Bitmap Files(*.bmp),*.bmp", Title:="Select Image To Open")

   ' If you did not click Cancel...
   If fname <> "False" Then

      ' Load the bitmap into the Image control.
      Image1.Picture = LoadPicture(fname)

      ' Refresh the UserForm.
      Me.Repaint
   End If

End Sub
```

7. On the **Run** menu, click **Run Sub/UserForm**.

   The UserForm appears.

8. Click the **Image** control.

   When you click the **Image** control, the **Select Image To Open** dialog box appears, and then you can select a bitmap file to insert into the control.

1. CONTENTS

2. ADD-IN

a. What is an Add-In?

An Excel add-in is something you add to enhance Excel's functionality. Some add-ins provide new worksheet functions you can use in formulas; other add-ins provide new commands or utilities. If the add-in is designed properly, the new features blend in well with the original interface so they appear to be part of the program. Excel ships with several add-ins, including the Analysis ToolPak and Solver. You can also get Excel add-ins from third-party suppliers.

Any knowledgeable user can create add-ins, but VBA programming skills are required. An Excel add-in is basically a different form of an XLSM workbook file. More specifically, an add-in is a normal XLSM workbook with the following differences:

- The IsAddin property of the Workbook object is True.
- The workbook window is hidden and can't be unhidden by using the View➪Window➪Unhide command.
- The workbook is not a member of the Workbooks collection. Rather, it's in the AddIns collection.

You can convert any workbook file into an add-in, but not all workbooks are good candidates. Because add-ins are always hidden, you can't display worksheets or chart sheets contained in an add-in. However, you can access an add-in's VBA Sub and Function procedures and display dialog boxes contained on UserForms.

Excel add-ins usually have an XLAM file extension to distinguish them from XLSM worksheet files. Earlier versions of Excel (before 2007) created add-ins with an XLA extension.

b. Why Create Add-Ins?

You might decide to convert your Excel application into an add-in for any of the following reasons:

- **Make it more difficult to access your code:** When you distribute an application as an add-in (and you protect its VBA project), casual users can't view the sheets in the workbook. If you use proprietary techniques in your VBA code, you can make it more difficult for others to copy the code. Excel's protection features aren't perfect though, and passwordcracking utilities are available.

- **Avoid confusion:** If a user loads your application as an add-in, the file is invisible and therefore less likely to confuse novice users or get in the way. Unlike a hidden workbook, the contents of an add-in can't be revealed.

- **Simplify access to worksheet functions:** Custom worksheet functions that you store in an add-in don't require the workbook name qualifier. For example, if you store a custom function named MOVAVG in a workbook named NEWFUNC.XLSM, you must use syntax like the following to use this function in a different workbook:
  =NEWFUNC.XLSM!MOVAVG(A1:A50)
  But if this function is stored in an add-in file that's open, you can use much simpler syntax because you don't need to include the file reference:
  =MOVAVG(A1:A50)

- **Provide easier access for users:** After you identify the location of your add-in, it appears in the Add-Ins dialog box, with a friendly name and a description of what it does. The user can easily enable or disable your add-in.
- **Gain better control over loading:** Add-ins can be opened automatically when Excel starts, regardless of the directory in which they are stored.
- **Avoid displaying prompts when unloading:** When an add-in is closed, the user never sees the dialog box prompt asking if you want to save changes in the file.

c. Working with Add-Ins

You load and unload add-ins by using the Add-Ins dialog box. To display this dialog box, choose File➪Options➪Add-Ins. Then select Excel Add-Ins from the drop-down list at the bottom of this dialog screen and click Go.

If you're using Excel 2010, getting to this dialog box is a bit easier: Choose Developer➪Add-Ins➪Add-Ins.

The list box contains the names of all add-ins that Excel knows about. In this list, check marks identify any currently open add-ins. You can open and close addins from the Add-Ins dialog box by selecting or deselecting the check boxes.

You can also open most add-in files (as if they were workbook files) by choosing the File➪Open command. An add-in opened in this manner does not appear in the Add-Ins dialog box. In addition, if the add-in was opened by using the Open command, you can't close it by choosing File➪Close. You can remove the add-in only by exiting and restarting Excel or by writing a macro to close the add-in. When you open an add-in, you may or may not notice anything different. In many cases, however, the Ribbon changes in some way — Excel displayseither a new tab or one or more new groups on an existing tab. For example, opening the Analysis ToolPak add-in gives you a new item on the Data tab: Analysis➪Data Analysis. If the add-in contains only custom worksheet functions, the new functions appear in the Insert Function dialog box, and you'll see no change to Excel's user interface.

d. Configuring an Add-In

Although you can convert any workbook to an add-in, not all workbooks benefit from this conversion. A workbook with no macros makes a completely useless add-in. In fact, the only types of workbooks that benefit from being converted to an add-in are those with macros. For example, a workbook that consists of general-purpose macros (Sub and Function procedures) makes an ideal add-in.

Creating an add-in isn't difficult, but it does require a bit of extra work. Use the following steps to create an add-in from a normal workbook file:

1. Develop your application and make sure that everything works properly. Don't forget to include a method for executing the macro or macros. You might want to define a shortcut key or customize the user interface in some way (see Chapter 19). If the add-in consists only of functions, there's no need to include a method to execute them because they will appear in the Insert Function dialog box.

2. Test the application by executing it when a *different* workbook is active.

Doing so simulates the application's behavior when it's used as an add-in because an add-in is never the active workbook.

3. Activate the VBE and select the workbook in the Project window; choose Tools⇨*VBAProject* Properties and click the Protection tab; select the Lock Project for Viewing check box and enter a password (twice); click OK. This step is necessary only if you want to prevent others from viewing or modifying your macros or UserForms.

4. In Excel 2010, choose Developer⇨Document Panel. In Excel 2007,choose Office⇨Prepare⇨Properties. Excel displays its Document Properties pane below the Ribbon.

5. In the Document Properties pane, enter a brief descriptive title in the Title field and a longer description in the Comments field. Steps 4 and 5 are not required but make the add-in easier to use, because the descriptions you enter appear in the Add-Ins dialog box when your add-in is selected.

6. Choose File⇨Save As.

7. In the Save As dialog box, select Excel add-in (*.xlam) from the Save as Type drop-down list.

8. Specify the folder that will store the add-in. Excel proposes a folder named AddIns, but you can save the file in any folder you like.

9. Click Save.

A copy of your workbook is converted to an add-in and saved with an XLAM extension. Your original workbook remains open.

3. CHARTS

a. Referencing Charts and Chart Objects in VBA Code

If you go back far enough in Excel history, you find that all charts used to be created as their own chart sheets. Then, in the mid-1990s, Excel added the amazing capability to embed a chart right onto an existing worksheet. This allowed a report to be created with tables of numbers and charts all on the same page, something we take for granted today.

These two different ways of dealing with charts make it necessary for you to deal with two separate object models for charts. When a chart is on its own standalone chart sheet, you are dealing with a Chart object. When a chart is embedded in a worksheet, you are dealing with a ChartObject object.

New versions of Excel include a third evolutionary branch because objects on a worksheet are also members of the Shapes collection.

In legacy versions of Excel, to reference the color of the chart area for an embedded chart, you must refer to the chart in this manner:

```
Worksheets(“Jan”).ChartObjects(“Chart 1”).Chart.ChartArea.Interior.ColorIndex  = 4
```

In Excel 2010, you can use the Shapes collection:

```
Worksheets(“Jan”).Shapes(“Char 1t”).Chart.ChartArea.Interior.ColorIndex = 4
```

In any version of Excel, if a chart is on its own chart sheet, you don’t have to specify the container; you can simply refer to the Chart object:

```
Sheets(“Chart1”).ChartArea.Interior.ColorIndex = 4
```

b. Creating a Chart

In legacy versions of Excel, you used the Charts.Add command to add a new chart. Next, you specified the source data, type of chart, and whether the chart should be on a new sheet or embedded on an existing worksheet. The first three lines of the following code create a clustered column chart on a new chart sheet. The fourth line moves the chart back to be an embedded object in Sheet1:

```
Charts.Add

ActiveChart.SetSourceData Source:=Worksheets(“Sheet1”).Range(“A1:E4”)

ActiveChart.ChartType = xlColumnClustered

ActiveChart.Location Where:=xlLocationAsObject, Name:=”Sheet1”
```

If you plan to share your macros with people who still use Excel 2003, you should use the Charts.Add method. However, if your application will be running only Excel 2007 or Excel 2010, you can use the new AddChart method. The code for the AddChart method can be as simple as the following:

```
` Create chart on the current sheet

ActiveSheet.Shapes.AddChart.Select

ActiveChart.SetSourceData Source:=Range("A1:E4")

ActiveChart.ChartType = xlColumnClustered
```

Alternatively, you can specify the chart type, size, and location as part of the AddChart method, as described in the next section.

c. Specifying the Size and Location of a Chart

The AddChart method has additional parameters you can use to specify the type of chart, the chart's location on the worksheet, and the size of the chart.

The location and size of a chart are specified in points (72 points = 1 inch). For example, the Top parameter requires the number of points from the top of Row 1 to the top edge of the worksheet.

```
The following code creates a chart that roughly covers the Range C11:J30:

Sub SpecifyLocation()

Dim WS As Worksheet

Set WS = Worksheets("Sheet1")

WS.Shapes.AddChart(xlColumnClustered, _

Left:=100, Top:=150, _

Width:=400, Height:=300).Select

ActiveChart.SetSourceData Source:=WS.Range("A1:E4")

End Sub
```

It requires a lot of trial and error to randomly figure out the exact distance in points to cause a chart to line up with a certain cell. Fortunately, you can ask VBA to tell you the distance in points to a certain cell. If you ask for the Left property of any cell, you find the distance to the top-left corner of that cell. You can also ask for the width of a range or the height of a range. For example, the following code creates a chart in exactly C11:J30:

```
Sub SpecifyExactLocation()

Dim WS As Worksheet
```

```
Set WS = Worksheets("Sheet1")

WS.Shapes.AddChart(xlColumnClustered, _

Left:=WS.Range("C11").Left, _

Top:=WS.Range("C11").Top, _

Width:=WS.Range("C11:J11").Width, _

Height:=WS.Range("C11:C30").Height).Select

ActiveChart.SetSourceData Source:=WS.Range("A1:E4")

End Sub
```

In this case, you are not moving the location of the Chart object. Instead, you are moving the location of the container that contains the chart. In Excel 2010, it is either the ChartObject or the Shape object. If you try to change the actual location of the chart, you move it within the container. Because you can actually move the chart area a few points in either direction inside the container, the code will run, but you will not get the desired results.

To move a chart that has already been created, you can reference either the ChartObject or the Shape and change the Top, Left, Width, and Height properties, as shown in the following macro:

```
Sub MoveAfterTheFact()

Dim WS As Worksheet

Set WS = Worksheets("Sheet1")

With WS.ChartObjects("Chart 9")

.Left = WS.Range("C21").Left

.Top = WS.Range("C21").Top

.Width = WS.Range("C1:H1").Width

.Height = WS.Range("C21:C25").Height

End With

End Sub
```

d. Later Referring to a Specific Chart

When a new chart is created, it is given a sequential name, such as Chart 1. If you select a chart and then look in the name box, you see the name of the chart. If the name of the chart is Chart 14, this does not necessarily mean that there are 14 charts on the worksheet.

It may mean, many individual charts have been created and deleted.

This means that on any given day that your macro runs, the `Chart` object might have a different name. If you need to reference the chart later in the macro, perhaps after you have selected other cells and the chart is no longer active, you might ask VBA for the name of the chart and store it in a variable for later use, as shown here:

```
Sub RememberTheName()

Dim WS As Worksheet

Set WS = Worksheets("Sheet1")

WS.Shapes.AddChart(xlColumnClustered, _

Left:=WS.Range("C11").Left, _

Top:=WS.Range("C11").Top, _

Width:=WS.Range("C11:J11").Width, _

Height:=WS.Range("C11:C30").Height _

).Select

ActiveChart.SetSourceData Source:=WS.Range("A1:E4")

' Remember the name in a variable

ThisChartObjectName = ActiveChart.Parent.Name

' more lines of code...

' then later in the macro, you need to re-assign the chart

With WS.Shapes(ThisChartObjectName)

.Chart.SetSourceData Source:=WS.Range("A20:E24"), PlotBy:=xlColumns

.Top = WS.Range("C26").Top

End With

End Sub
```

In the preceding macro, the variable `ThisChartObjectName` contains the name of the `Chart` object. This method works great if your changes will happen later in the same macro. However, after the macro finishes running, the variable will be out of scope, and you won't be able to access the name later.

If you want to be able to remember a chart name, you could store the name in an out-ofthe-way cell on the worksheet. The first macro here stores the name in Cell Z1, and the second macro then later modifies the chart using the name stored in Cell Z1:

```
Sub StoreTheName()
```

```
Dim WS As Worksheet

Set WS = Worksheets("Sheet1")

WS.Shapes.AddChart(xlColumnClustered, _

Left:=WS.Range("C11").Left, _

Top:=WS.Range("C11").Top, _

Width:=WS.Range("C11:J11").Width, _

Height:=WS.Range("C11:C30").Height _

).Select

ActiveChart.SetSourceData Source:=WS.Range("A1:E4")

Range("Z1").Value = ActiveChart.Parent.Name

End Sub
```

After the previous macro stored the name in Cell Z1, the following macro will use the value in Z1 to figure out which chart to change:

```
Sub ChangeTheChartLater()

Dim WS As Worksheet

Set WS = Worksheets("Sheet1")

MyName = WS.Range("Z1").Value

With WS.Shapes(MyName)

.Chart.SetSourceData Source:=WS.Range("A20:E24"), PlotBy:=xlColumns

.Top = WS.Range("C26").Top

End With

End Sub
```

If you need to modify a preexisting chart—such as a chart that you did not create—and there is only one chart on the worksheet, you can use this line of code:

```
WS.ChartObjects(1).Chart.Interior.ColorIndex = 4
```

If there are many charts, and you need to find the one with the upper-left corner located in

Cell A4, you can loop through all the `Chart` objects until you find one in the correct location, like this:

```
For each Cht in ActiveSheet.ChartObjects
```

```
If Cht.TopLeftCell.Address = "$A$4" then

Cht.Interior.ColorIndex = 4

end if

Next Cht
```

e. Recording Commands from the Layout or Design Tabs

With charts in Excel 2010, there are three levels of chart changes. The global chart settings that indicate the chart type and style are on the Design tab. Selections from the built-in element settings appear on the Layout tab. You make micro changes by using the Format tab.

The macro recorder was not finished in Excel 2007, but it is working in Excel 2010. If you need to make certain changes, this enables you to quickly record a macro and then copy its code.

f. Specifying a Built-in Chart Type

Excel 2010 has 73 built-in chart types. To change a chart to one of the 73 types, you use the ChartType property. This property can be applied either to a chart or to a series within a chart. Here is an example that changes the type for the entire chart:

```
ActiveChart.ChartType = xlBubble
```

To change the second series on a chart to a line chart, you use this:

```
ActiveChart.Series(2).ChartType = xlLine
```

g. Specifying a Template Chart Type

Excel 2010 allows you to create a custom chart template with all your preferred settings such as colors and fonts. This technique is a great way to save time when you are creating a chart with a lot of custom formatting.

A VBA macro can make use of a custom chart template, provided you plan to distribute the custom chart template to each person who will run your macro.

In Excel 2010, you save custom chart types as .crtx files and store them in the %appdata%\Microsoft\Templates\Charts\ folder.

To apply a custom chart type, you use the following:

```
ActiveChart.ApplyChartTemplate "MyChart.crtx"
```

If the chart template does not exist, VBA returns an error. If you would like Excel to continue without displaying a debug error, you can instruct the error handler to resume with the next line. After applying the chart template, go back to the default state of the error handler so that you will see any errors. Here's how you do this:

```
On Error Resume Next

ActiveChart.ApplyChartTemplate ("MyChart.crtx")

On Error GoTo 0 ' that final character is a zero
```

h. Changing a Chart's Layout or Style

Two galleries—the Chart Layout gallery and the Styles gallery—make up the bulk of the Design tab.

The Chart Layout gallery offers from 4 to 12 combinations of chart elements. These combinations are different for various chart types. When you look at the gallery, the ToolTips for the layouts show that the layouts are named imaginatively as Layout 1 through Layout 11.

To apply one of the built-in layouts in a macro, you have to use the ApplyLayout method with a number from 1 through 12 to correspond to the built-in layouts. The following code applies Layout 1 to the active chart:

```
ActiveChart.ApplyLayout 1
```

To apply a style to a chart, you use the ChartStyle property, assigning it a value from 1 to 48:

```
ActiveChart.ChartStyle = 1
```

The ChartStyle property changes the colors in the chart. However, a number of formatting changes from the Format tab are not overwritten when you change the ChartStyle property. For example, suppose that you had applied glow or a clear glass bezel to a chart.

Running the preceding code will not clear that formatting.

To clear any previous formatting, you use the ClearToMatchStyle method:

```
ActiveChart.ChartStyle = 1

ActiveChart.ClearToMatchStyle
```

i. Using SetElement to Emulate Changes on the Layout Tab

The Layout tab contains a number of built-in settings. There are similar menus for each of the icons in the figure.

If you use a built-in menu item to change the titles, legend, labels, axes, gridlines, or background, it is probably handled in code that uses the SetElement method that is available in Excel 2010.

The macro recorder always works for the built-in settings on the Layout tab. If you do not feel like looking up the proper constant in this book, you can always quickly record a macro. The SetElement method is followed by a constant that specifies which menu item to select.

For example, if you want to choose Show Legend at Left, you can use this code:

```
ActiveChart.SetElement msoElementLegendLeft
```

j. Changing a Chart Title Using VBA

The Layout tab’s built-in menus enable you to add a title above a chart, but they do not enable you to change the characters in a chart title or axis title.

In the Excel interface, you can double-click the chart title text and type a new title tochange the title. To specify a chart title in VBA, use this code:

```
ActiveChart.ChartTitle.Caption = “My Chart”
```

Similarly, you can specify the axis titles by using the Caption property. The following code changes the axis title along the category axis:

```
ActiveChart.Axes(xlCategory, xlPrimary).AxisTitle.Caption = “Months”
```

k. Changing an Object’s Fill

The Shape Fill drop-down on the Format tab enables you to choose a single color, a gradient, a picture, or a texture for the fill.

To apply a specific color, you can use the RGB (red, green, blue) setting. To create a color, you specify a value from 0 to 255 for levels of red, green, and blue. The following code applies a simple blue fill:

```
Dim cht As Chart

Dim upb As UpBars

Set cht = ActiveChart

Set upb = cht.ChartGroups(1).UpBars

upb.Format.Fill.ForeColor.RGB = RGB(0, 0, 255)
```

If you would like an object to pick up the color from a specific theme accent color, you use the ObjectThemeColor property. The following code changes the bar color of the first series to accent color 6, which is an orange color in the Office theme. However, this might be another color if the workbook is using a different theme.

```
Sub ApplyThemeColor()

Dim cht As Chart

Dim ser As Series

Set cht = ActiveChart

Set ser = cht.SeriesCollection(1)

ser.Format.Fill.ForeColor.ObjectThemeColor = msoThemeColorAccent6

End Sub
```

To apply a built-in texture, you use the PresetTextured method. The following code applies a green marble texture to the second series. However, you can apply any of the 20 different textures:

```
Sub ApplyTexture()

Dim cht As Chart

Dim ser As Series

Set cht = ActiveChart

Set ser = cht.SeriesCollection(2)

ser.Format.Fill.PresetTextured msoTextureGreenMarble

End Sub
```

To fill the bars of a data series with a picture, you use the UserPicture method and specify the path and filename of an image on the computer, as in the following example:

```
Sub FormatWithPicture()

Dim cht As Chart

Dim ser As Series

Set cht = ActiveChart

Set ser = cht.SeriesCollection(1)

MyPic = “C:\PodCastTitle1.jpg”

ser.Format.Fill.UserPicture MyPic

End Sub
```

Microsoft removed patterns as fills from Excel 2007. However, this method was restored in Excel 2010 because of the outcry from customers who used patterns to differentiate columns printed on monochrome printers.

In Excel 2010, you can apply a pattern using the .Patterned method. Patterns have a type such as msoPatternPlain, as well as a foreground and background color. The following code creates dark red vertical lines on a white background:

```
Sub FormatWithPicture()

Dim cht As Chart

Dim ser As Series

Set cht = ActiveChart

Set ser = cht.SeriesCollection(1)

With ser.Format.Fill

.Patterned msoPatternDarkVertical

.BackColor.RGB = RGB(255,255,255)

.ForeColor.RGB = RGB(255,0,0)

End With

End Sub
```

Code that uses patterns will work in every version of Excel except Excel 2007. Therefore, do not use this code if you will be sharing the macro with coworkers who use Excel 2007.

I. Gradients

They are more difficult to specify than fills. Excel 2010 provides three methods that help you set up the common gradients. The OneColorGradient and TwoColorGradient methods require that you specify a gradient direction such as msoGradientFromCorner. You can then specify one of four styles, numbered 1 through 4, depending on whether you want the gradient to start at the top left, top right, bottom left, or bottom right. After using a gradient method, you need to specify the ForeColor and the BackColor settings for the object.

```
The following macro sets up a two-color gradient using two theme colors:

Sub TwoColorGradient()

Dim cht As Chart

Dim ser As Series
```

```
Set cht = ActiveChart

Set ser = cht.SeriesCollection(1)

ser.Format.Fill.TwoColorGradient msoGradientFromCorner, 3

ser.Format.Fill.ForeColor.ObjectThemeColor = msoThemeColorAccent6

ser.Format.Fill.BackColor.ObjectThemeColor = msoThemeColorAccent2

End Sub
```

When using the OneColorGradient method, you specify a direction, a style (1 through 4), and a darkness value between 0 and 1 (0 for darker gradients or 1 for lighter gradients).

When using the PresetGradient method, you specify a direction, a style (1 through 4), and the type of gradient such as msoGradientBrass, msoGradientLateSunset, or msoGradientRainbow.

Again, as you are typing this code in the VB Editor, the AutoComplete tool provides a complete list of the available preset gradient types.

m. Formatting Line Settings

The LineFormat object formats either a line or the border around an object. You can change numerous properties for a line, such as the color, arrows, dash style, and so on.

The following macro formats the trendline for the first series in a chart:

```
Sub FormatLineOrBorders()

Dim cht As Chart

Set cht = ActiveChart

With cht.SeriesCollection(1).Trendlines(1).Format.Line

.DashStyle = msoLineLongDashDotDot

.ForeColor.RGB = RGB(50, 0, 128)

.BeginArrowheadLength = msoArrowheadShort

.BeginArrowheadStyle = msoArrowheadOval

.BeginArrowheadWidth = msoArrowheadNarrow

.EndArrowheadLength = msoArrowheadLong

.EndArrowheadStyle = msoArrowheadTriangle

.EndArrowheadWidth = msoArrowheadWide
```

```
End With

End Sub
```

When you are formatting a border, the arrow settings are not relevant, so the code is shorter than the code for formatting a line. The following macro formats the border around a chart:

```
Sub FormatBorder()

Dim cht As Chart

Set cht = ActiveChart

With cht.ChartArea.Format.Line

.DashStyle = msoLineLongDashDotDot

.ForeColor.RGB = RGB(50, 0, 128)

End With

End Sub
```

4. CHART EVENTS

a. Chart Object Events

Chart events occur when the user activates or changes a chart. Events on chart sheets are enabled by default. To view the event procedures for a sheet, right-click the sheet tab and select View Code from the shortcut menu. Select the event name from the Procedure drop-down list box.

- Activate
- BeforeDoubleClick
- BeforeRightClick
- Calculate
- Deactivate
- MouseDown
- MouseMove
- MouseUp
- Resize
- Select
- SeriesChange

This example changes a point's border color when the user changes the point value.

```
Private Sub Chart_SeriesChange(ByVal SeriesIndex As Long, _

        ByVal PointIndex As Long)

    Set p = ActiveChart.SeriesCollection(SeriesIndex). _

        Points(PointIndex)

    p.Border.ColorIndex = 3

End Sub
```

b. Using Events with Embedded Charts

Events are enabled for chart sheets by default. Before you can use events with a Chart object that represents an embedded chart, you must create a new class module and declare an object of type Chart with events. For example, assume that a new class module is created and named EventClassModule. The new class module contains the following code.

```
Public WithEvents myChartClass As Chart
```

After the new object has been declared with events, it appears in the Object drop-down list box in the class module, and you can write event procedures for this object. (When you select the new object in the Object box, the valid events for that object are listed in the Procedure drop-down list box.)

Before your procedures will run, however, you must connect the declared object in the class module with the embedded chart. You can do this by using the following code from any module.

```
Dim myClassModule As New EventClassModule

Sub InitializeChart()

 Set myClassModule.myChartClass = _

 Charts(1).ChartObjects(1).Chart

End Sub
```

After you run the InitializeChart procedure, the myChartClass object in the class module points to embedded chart 1 on worksheet 1, and the event procedures in the class module will run when the events occur.

c. Chart.GetChartElement Method (Excel)

Returns information about the chart element at specified X and Y coordinates. This method is unusual in that you specify values for only the first two arguments. Microsoft Excel fills in the other arguments, and your code should examine those values when the method returns.

expression .**GetChartElement**(***x***, ***y***, ***ElementID***, ***Arg1***, ***Arg2***)

expression A variable that represents a **Chart** object.

Name	**Required/ Optional**	**Data Type**	
x	Required	**Long**	The X coordinate of the chart element.
y	Required	**Long**	The Y coordinate of the chart element.
ElementID	Required	**Long**	When the method returns, this argument contains the **XLChartItem** value of the chart element at the specified coordinates. For more information, see the ?Remarks? section.
Arg1	Required	**Long**	When the method returns, this argument contains information related to the chart element. For more

			information, see the ?Remarks? section.
Arg2	Required	**Long**	When the method returns, this argument contains information related to the chart element. For more information, see the ?Remarks? section.

The value of *ElementID* after the method returns determines whether *Arg1* and *Arg2* contain any information, as shown in the following table.

ElementID Constant	**Constant Value**	**Arg1**	**Arg2**
xlAxis	21	AxisIndex	AxisType
xlAxisTitle	17	AxisIndex	AxisType
xlDisplayUnitLabel	30	AxisIndex	AxisType
xlMajorGridlines	15	AxisIndex	AxisType
xlMinorGridlines	16	AxisIndex	AxisType
xlPivotChartDropZone	32	DropZoneType	None
xlPivotChartFieldButton	31	DropZoneType	PivotFieldIndex
xlDownBars	20	GroupIndex	None
xlDropLines	26	GroupIndex	None
xlHiLoLines	25	GroupIndex	None
xlRadarAxisLabels	27	GroupIndex	None
xlSeriesLines	22	GroupIndex	None
xlUpBars	18	GroupIndex	None
xlChartArea	2	None	None
xlChartTitle	4	None	None
xlCorners	6	None	None

xlDataTable	7	None	None
xlFloor	23	None	None
xlLeaderLines	29	None	None
xlLegend	24	None	None
xlNothing	28	None	None
xlPlotArea	19	None	None
xlWalls	5	None	None
xlDataLabel	7	SeriesIndex	PointIndex
xlErrorBars	9	SeriesIndex	None
xlLegendEntry	12	SeriesIndex	None
xlLegendKey	13	SeriesIndex	None
xlSeries	3	SeriesIndex	PointIndex
xlShape	14	ShapeIndex	None
xlTrendline	8	SeriesIndex	TrendLineIndex
xlXErrorBars	10	SeriesIndex	None
xlYErrorBars	11	SeriesIndex	None

The following table describes the meaning of *Arg1* and *Arg2* after the method returns.

Argument	**Description**
AxisIndex	Specifies whether the axis is primary or secondary. Can be one of the following **XlAxisGroup** constants: **xlPrimary** or **xlSecondary**.
AxisType	Specifies the axis type. Can be one of the following **XlAxisType** constants: **xlCategory**, **xlSeriesAxis**, or **xlValue**.

DropZoneType	Specifies the drop zone type: column, data, page, or row field. Can be one of the following **XlPivotFieldOrientation** constants:**xlColumnField**, **xlDataField**, **xlPageField**, or **xlRowField**. The column and row field constants specify the series and category fields, respectively.
GroupIndex	Specifies the offset within the **ChartGroups** collection for a specific chart group.
PivotFieldIndex	Specifies the offset within the **PivotFields** collection for a specific column (series), data, page, or row (category) field. -1 if the drop zone type is **xlDataField**.
PointIndex	Specifies the offset within the **Points** collection for a specific point within a series. A value of ? 1 indicates that all data points are selected.
SeriesIndex	Specifies the offset within the **Series** collection for a specific series.
ShapeIndex	Specifies the offset within the **Shapes** collection for a specific shape.
TrendlineIndex	Specifies the offset within the **Trendlines** collection for a specific trendline within a series.

Example

```
Private Sub Chart_MouseMove(ByVal Button As Long, _
 ByVal Shift As Long, ByVal X As Long, ByVal Y As Long)
 Dim IDNum As Long
 Dim a As Long
 Dim b As Long

 ActiveChart.GetChartElement X, Y, IDNum, a, b
 If IDNum = xlLegendEntry Then _
 MsgBox "WARNING: Move away from the legend"
End Sub
```

1. CONTENTS

2. STARTING AN APPLICATION FROM EXCEL

Launching another application from Excel is often useful. For example, you might want to execute another Microsoft Office application or even a DOS batch file from Excel. Or, as an application developer, you may want to make it easy for a user to access the Windows Control Panel to adjust system settings.

a. Using the VBA Shell function

The VBA Shell function makes launching other programs relatively easy. Following is an example of VBA code that launches the Windows Calculator application.

```
Sub StartCalc()
  Dim Program As String
  Dim TaskID As Double
  On Error Resume Next
  Program = “calc.exe”
  TaskID = Shell(Program, 1)
  If Err <> 0 Then
    MsgBox “Cannot start “ & Program, vbCritical, “Error”
  End If
End Sub
```

The Shell function returns a task identification number for the application specified in the first argument. You can use this number later to activate the task. The second argument for the Shell function determines how the application is displayed. (1 is the code for a normal-size window, with the focus.) Refer to the Help system for other values for this argument.

If the Shell function isn’t successful, it generates an error. Therefore, this procedure uses an On Error statement to display a message if the executable file can’t be found or if some other error occurs.

It’s important to understand that your VBA code doesn’t pause while the application that was started with the Shell function is running. In other words, the Shell function runs the application *asynchronously.* If the procedure has more instructions after the Shell function is executed, these instructions are executed concurrently with the newly loaded program. If any instruction requires user intervention (for example, displaying a message box), Excel’s title bar flashes while the other application is active.

In some cases, you may want to launch an application with the Shell function, but you need your VBA code to pause until the application is closed. For example, the launched application might generate a file that is used later in your code. Although you can't pause the execution of your code, you *can* create a loop that does nothing except monitor the application's status. The example that follows displays a message box when the application launched by the Shell function has ended:

```
Declare PtrSafe Function OpenProcess Lib “kernel32” (ByVal dwDesiredAccess As Long, _
ByVal bInheritHandle As Long,  ByVal dwProcessId As Long) As Long

Declare PtrSafe Function GetExitCodeProcess Lib “kernel32”_
(ByVal hProcess As Long, lpExitCode As Long) As Long

Sub StartCalc2()
  Dim TaskID As Long
  Dim hProc As Long
  Dim lExitCode As Long
  Dim ACCESS_TYPE As Integer, STILL_ACTIVE As Integer
  Dim Program As String
  ACCESS_TYPE = &H400
  STILL_ACTIVE = &H103
  Program = “Calc.exe”
  On Error Resume Next
  ‘ Shell the task
  TaskID = Shell(Program, 1)
  ‘ Get the process handle
  hProc = OpenProcess(ACCESS_TYPE, False, TaskID)
  If Err <> 0 Then
    MsgBox “Cannot start “ & Program, vbCritical, “Error”
    Exit Sub
  End If
  Do ‘Loop continuously
    ‘ Check on the process
    GetExitCodeProcess hProc, lExitCode
    ‘ Allow event processing
    DoEvents
  Loop While lExitCode = STILL_ACTIVE

  ‘ Task is finished, so show message
  MsgBox Program & “ was closed”
End Sub
```

While the launched program is running, this procedure continually calls the GetExitCode Process function from within a Do-Loop structure, testing for its returned value (lExitCode).

When the program is finished, lExitCode returns a different value, the loop ends, and the VBA code resumes executing.

b. Displaying a folder window

The Shell function is also handy if you need to display a particular directory using Windows Explorer. For example, the statement that follows displays the folder of the active workbook (but only if the workbook has been saved):

```
If ActiveWorkbook.Path <> "" Then _
Shell "explorer.exe " & ActiveWorkbook.Path, vbNormalFocus
```

c. Using the Windows ShellExecute API function

ShellExecute is a Windows Application Programming Interface (API) function that is useful for starting other applications. Importantly, this function can start an application only if an associated filename is known (assuming that the file type is registered with Windows). For example, you can use ShellExecute to display a Web document by starting the default Web browser.

Or you can use an e-mail address to start the default e-mail client.

The API declaration follows (this code works only with Excel 2010):

```
Private Declare PtrSafe Function ShellExecute Lib "shell32.dll" _
Alias "ShellExecuteA" (ByVal hWnd As Long, _
ByVal lpOperation As String, ByVal lpFile As String, _
ByVal lpParameters As String, ByVal lpDirectory As String, _
ByVal nShowCmd As Long) As Long
```

The following procedure demonstrates how to call the ShellExecute function. In this example, it opens a graphics file by using the graphics program that's set up to handle JPG files. If the result returned by the function is less than 32, then an error occurred.

```
Sub ShowGraphic()
  Dim FileName As String
  Dim Result As Long
  FileName = ThisWorkbook.Path & "\flower.jpg"
  Result = ShellExecute(0&, vbNullString, FileName, _
  vbNullString, vbNullString, vbNormalFocus)
  If Result < 32 Then MsgBox "Error"
End Sub
```

The next procedure opens a text file, using the default text file program:

```
Sub OpenTextFile()
  Dim FileName As String
  Dim Result As Long
  FileName = ThisWorkbook.Path & "\textfile.txt"
  Result = ShellExecute(0&, vbNullString, FileName, _
  vbNullString, vbNullString, vbNormalFocus)
  If Result < 32 Then MsgBox "Error"
End Sub
```

The following example is similar, but it opens a Web URL by using the default browser:

```
Sub OpenURL()
  Dim URL As String
  Dim Result As Long
  URL = "http://google.com"
  Result = ShellExecute(0&, vbNullString, URL, _
  vbNullString, vbNullString, vbNormalFocus)
  If Result < 32 Then MsgBox "Error"
End Sub
```

You can also use this technique with an e-mail address. The following example opens the default e-mail client (if one exists) and then addresses an e-mail to the recipient:

```
Sub StartEmail()
  Dim Addr As String
  Dim Result As Long
  Addr = "mailto:bgates@microsoft.com"
  Result = ShellExecute(0&, vbNullString, Addr, _
  vbNullString, vbNullString, vbNormalFocus)
  If Result < 32 Then MsgBox "Error"
End Sub
```

3. ACTIVATING AN APPLICATION WITH EXCEL

In the previous section, I discuss various ways to start an application. You may find that if an application is already running, using the Shell function may start another instance of it. In most cases, however, you want to *activate* the instance that's running — not start another instance of it.

a. Using AppActivate

The following StartCalculator procedure uses the AppActivate statement to activate an application if it's already running (in this case, the Windows Calculator). The argument for AppActivate is the caption of the application's title bar. If the AppActivate statement generates an error, it indicates that the Calculator is not running. Therefore, the routine starts the application.

```
Sub StartCalculator()
  Dim AppFile As String
  Dim CalcTaskID As Double
  AppFile = "Calc.exe"
  On Error Resume Next
  AppActivate "Calculator"
  If Err <> 0 Then
    Err = 0
    CalcTaskID = Shell(AppFile, 1)
    If Err <> 0 Then MsgBox "Can't start Calculator"
  End If
End Sub
```

b. Activating a Microsoft Office application

If the application that you want to start is one of several Microsoft applications, you can use the ActivateMicrosoftApp method of the Application object. For example, the following procedure starts Word:

```
Sub StartWord()
  Application.ActivateMicrosoftApp xlMicrosoftWord
End Sub
```

If Word is already running when the preceding procedure is executed, it is activated. The other constants available for this method are:

- xlMicrosoftPowerPoint
- xlMicrosoftMail (activates Outlook)
- xlMicrosoftAccess
- xlMicrosoftProject

c. Running Control Panel Dialog Boxes

Windows provides quite a few system dialog boxes and wizards, most of which are accessible from the Windows Control Panel. You might need to display one or more of these from your Excel application. For example, you might want to display the Windows Date and Time dialog box.

The key to running other system dialog boxes is to execute the rundll32.exe application by using the VBA Shell function.

The following procedure displays the Date and Time dialog box:

```
Sub ShowDateTimeDlg()
  Dim Arg As String
  Dim TaskID As Double
  Arg = "rundll32.exe shell32.dll,Control_RunDLL timedate.cpl"
  On Error Resume Next
  TaskID = Shell(Arg)
  If Err <> 0 Then
    MsgBox ("Cannot start the application.")
  End If
End Sub
```

Following is the general format for the rundll32.exe application:

```
rundll32.exe shell32.dll,Control_RunDLL filename.cpl, n,t
```

- filename.cpl: The name of one of the Control Panel *.CPL files.
- n: The zero-based number of the applet within the *.CPL file.
- t: The number of the tab (for multi-tabbed applets).

4. USING AUTOMATION IN EXCEL

You can write an Excel macro to control other applications, such as Microsoft Word. More accurately, the Excel macro will control Word's automation server. In such circumstances, Excel is the *client application,* and Word is the *server application.* Or you can write a VBA application in Word to control Excel. The process of one application's controlling another is sometimes known as *Object Linking and Embedding (OLE),* or simply *automation.*

The concept behind automation is quite appealing. A developer who needs to generate a chart, for example, can just reach into another application's grab bag of objects, fetch a Chart object, and then manipulate its properties and use its methods. Automation, in a sense, blurs the boundaries between applications. An end user may be working with an Access object and not even realize it.

In this section, I demonstrate how to use VBA to access and manipulate the objects exposed by other applications. The examples use Microsoft Word, but the concepts apply to any application that exposes its objects for automation — which accounts for an increasing number of applications.

a. Working with foreign objects using automation

As you may know, you can use Excel's Insert➜Text➜Object command to embed an object, such as a Word document, in a worksheet.

In addition, you can create an object and manipulate it with VBA. (This action is the heart of Automation.) When you do so, you usually have full access to the object. For developers, this technique is generally more beneficial than embedding the object in a worksheet. When an object is embedded, the user must know how to use the automation object's application. But when you use VBA to work with the object, you can program the object so that the user can manipulate it by an action as simple as a button click.

b. Early versus late binding

Before you can work with an external object, you must create an instance of the object. You can do so in either of two ways: early binding or late binding. *Binding* refers to matching the function calls written by the programmer to the actual code that implements the function.

Early binding

To use early binding, create a reference to the object library by choosing the Tools➜ References command in the Visual Basic Editor (VBE), which brings up the Reference dialog box.

Then put a check mark next to the object library you need to reference. After the reference to the object library is established, you can use the Object Browser, to view the object names, methods, and properties. To access the Object Browser, press F2 in the VBE.

When you use early binding, you must establish a reference to a version-specific object library. For example, you can specify Microsoft Word 10.0 Object Library (for Word 2002), Microsoft Word 11.0 Object Library (for Word 2003), Microsoft Word 12.0 Object Library (for Word 2007), or Microsoft Word 14.0 Object Library (for Word 2010). Then you use a statement like the following to create the object:

```
Dim WordApp As New Word.Application
```

Using early binding to create the object by setting a reference to the object library usually is more efficient and also often yields better performance. Early binding is an option, however, only if the object that you're controlling has a separate type library or object library file. You also need to ensure that the user of the application actually has a copy of the specific library installed.

Another advantage of early binding is that you can use constants that are defined in the object library. For example, Word (like Excel) contains many predefined constants that you can use in your VBA code. If you use early binding, you can use the constants in your code. If you use late binding, you'll need to use the actual value rather than the constant.

Still another benefit of using early binding is that you can take advantage of the VBE Object Browser and Auto List Members option to make it easier to access properties and methods; this feature doesn't work when you use late binding because the type of the object is known only at runtime.

Late binding

At runtime, you use either the CreateObject function to create the object or the GetObject function to obtain a saved instance of the object. Such an object is declared as a generic Object type, and its object reference is resolved at runtime.

You can use late binding even when you don't know which version of the application is installed on the user's system. For example, the following code, which works with Word 97 and later, creates a Word object:

```
Dim WordApp As Object
Set WordApp = CreateObject("Word.Application")
```

If multiple versions of Word are installed, you can create an object for a specific version. The following statement, for example, uses Word 2003:

```
Set WordApp = CreateObject("Word.Application.11")
```

The Registry key for Word's Automation object and the reference to the Application object in VBA just happen to be the same: Word.Application. They do not, however, refer to the same thing. When you declare an object As Word.Application or As New Word.Application, the term refers to the Application object in the Word library. But when you invoke the function CreateObject("Word.Application"), the term refers to the moniker by which the latest version of Word is known in the Windows System Registry. This isn't the case for all automation objects, although it is true for the main Office 2010 components. If the user replaces Word 2003 with Word 2010, CreateObject("Word.Application") will continue to work properly, referring to the new application. If Word 2010 is removed, however, CreateObject("Word.Application.14"), which uses the alternate version-specific name for Word 2010, will fail to work.

The CreateObject function used on an automation object such as Word.Application or Excel.Application always creates a new *instance* of that automation object. That is, it starts up a new and separate copy of the automation part of the program. Even if an instance of the automation object is already running, a new instance is started, and then an object of the specified type is created.

To use the current instance or to start the application and have it load a file, use the GetObject function.

If you need to automate an Office application, it is recommended that you use early binding and reference the earliest version of the product that you expect could be installed on your client's system. For example, if you need to be able to automate Word 2003, Word 2007, and Word 2010, you should use the type library for Word 2003 to maintain compatibility with all three versions. This approach, of course, will mean that you can't use features found only in the later version of Word.

c. GetObject versus CreateObject

VBA's GetObject and CreateObject functions both return a reference to an object, but they work in different ways.

The CreateObject function creates an interface to a new instance of an application. Use this function when the application isn't running. If an instance of the application is already running, a new instance is started. For example, the following statement starts Excel, and the object returned in XLApp is a reference to the Excel.Application object that it created.

```
Set XLApp = CreateObject("Excel.Application")
```

The GetObject function is either used with an application that's already running or to start an application with a file already loaded. The following statement, for example, starts Excel with the file Myfile.xls already loaded. The object returned in XLBook is a reference to the Workbook object (the Myfile.xlsx file):

```
Set XLBook = GetObject("C:\Myfile.xlsx")
```

d. A simple example of late binding

The following example demonstrates how to create a Word object by using late binding. This procedure creates the object, displays the version number, closes the Word application, and then destroys the object (thus freeing the memory that it used):

```
Sub GetWordVersion()
  Dim WordApp As Object
  Set WordApp = CreateObject("Word.Application")
  MsgBox WordApp.Version
  WordApp.Quit
  Set WordApp = Nothing
End Sub
```

This example can also be programmed using early binding. Before doing so, choose Tools ➜References to set a reference to the Word object library. Then you can use the following code:

```
Sub GetWordVersion()
  Dim WordApp As New Word.Application
  MsgBox WordApp.Version
  WordApp.Quit
  Set WordApp = Nothing
End Sub
```

The Word object that's created in this procedure is invisible. If you'd like to see the object's window while it's being manipulated, set its Visible property to True, as follows:

```
WordApp.Visible = True
```

5. USING SENDKEYS

Not all applications support Automation. In some cases, you can still control some aspects of the application even if it doesn’t support Automation. You can use Excel’s SendKeys method to send keystrokes to an application, simulating actions that a live human might perform.

Although using the SendKeys method may seem like a good solution, you’ll find that it can be very tricky and not completely reliable. In fact, it may not work at all. A potential problem is that it relies on a specific user interface. If a later version of the program that you’re sending keystrokes to has a different user interface, your application might no longer work. Consequently, you should use SendKeys only as a last resort.

Following is a very simple example. This procedure runs the Windows Calculator program and displays its Scientific mode: That is, it executes the View➜Scientific command.

```
Sub TestKeys()
  Shell “calc.Exe”, vbNormalFocus
  Application.SendKeys “%vs”
End Sub
```

In this example, the code sends out Alt+V (the percent sign represents the Alt key) followed by S.

SendKeys is documented in the Help system, which describes how to send nonstandard keystrokes, such as Alt and Ctrl key combinations.

The TestKeys procedure fails on a system with Windows 7 installed. Although the Windows 7 calculator uses the same menu accelerator key, Windows 7 supports SendKeys only if User Account Control (a security feature) is turned off.

That’s a good example of why you should use SendKeys only as a last resort.