Microsoft Office Training Series

Excel VBA

Introduction

- Courses never Cancelled
- 12+ Months Schedule
- 24 Months Online Support
- UK Wide Delivery

STL MicrosoftTraining.net

Learning & Performance Institute
Accredited Learning Provider

ilm Recognised Provider

Microsoft Partner
Certified Silver Partner

Welcome to your Excel VBA Introduction training course

- Record macros
- The visual basic Editor
- Understand objects (Object oriented programming)
- Control structure using decision code (If Then Else & Select Case)
- Understand and use loops (Do, For Next, For Each)
- Test code and debugging tools

Contents

Unit 1 Recording Macros

In this unit you will learn how to:

- Record a macro (absolute & relative macros)
- Run a macro
- Execute macros from button & from the quick access toolbar

Recording and Running Macros

A macro is a series of commands in Visual Basic, also known as a Sub Procedure. Macros allow you to automate tedious or complicated tasks, particularly those that are prone to error.

You can record a sequence of commands and replay the actions by running the macro. Examining the code of a recorded macro can give you insight into how Visual Basic works.

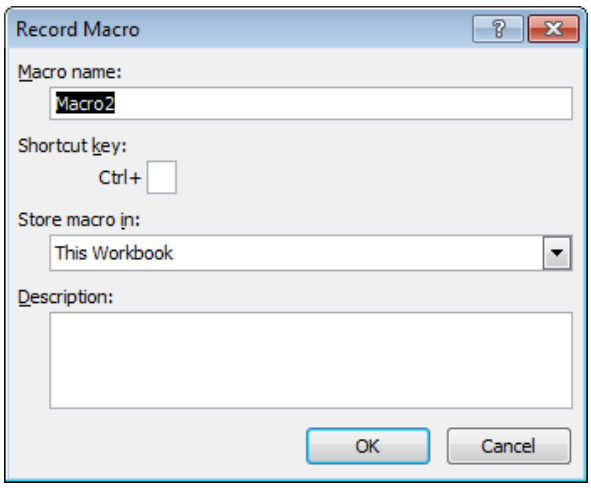
Macros can be stored on the current worksheet or made available globally by saving them in the Personal.xlsm workbook. This is a hidden workbook that automatically opens when you open Excel.

Recording a Macro

View Ribbon > Macro Section > Macro > Record Macro

Note: Excel will require the Developer Ribbon to be available for most VBA related tasks.
2007: Office Button > Options > Display > Tick Show Developer
2010/2013/2016: File Ribbon > Options > Customise Ribbon > Tick Show Developer

The **Record Macro** dialog box appears.



- Type the macro's name in the Macro name box (cannot contain spaces)
- Select where the macro is to be stored
- Add a shortcut key, if desired
- Type a description, if desired (this will appear in the VB editor as commented code)
- Click OK.

Perform the actions to be recorded.

To end recording

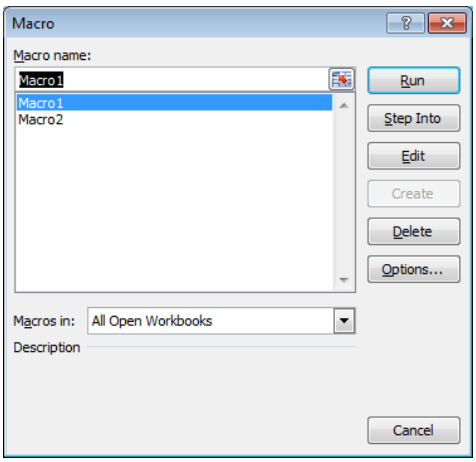
- Click Stop button in bottom left of Status bar
- Or from the Developer ribbon

Running a Macro

A macro can be run by using a keystroke combination, a menu, a toolbar or the Macro dialog box. This provides a list of all available macros in the open workbooks. To open this:

- Developer Ribbon > Code Section > Macros Button

The Macro dialog box appears.



- Select the desired macro from the Macro Name list
- Click Run.

Macros without a workbook name in front indicate that they belong to the active workbook.

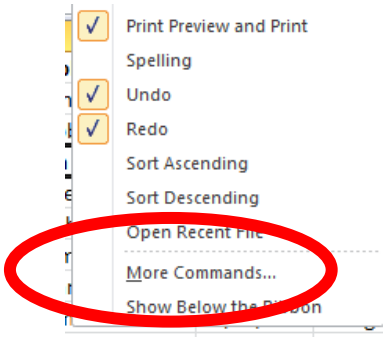
Click the Step Into button in the Macro dialog box to run the macro one line at a time. Once the VB editor displays, press F8.

Keep pressing F8 to step through the code. Display both the Excel and VB Editor windows in order to see the results of the code execution.

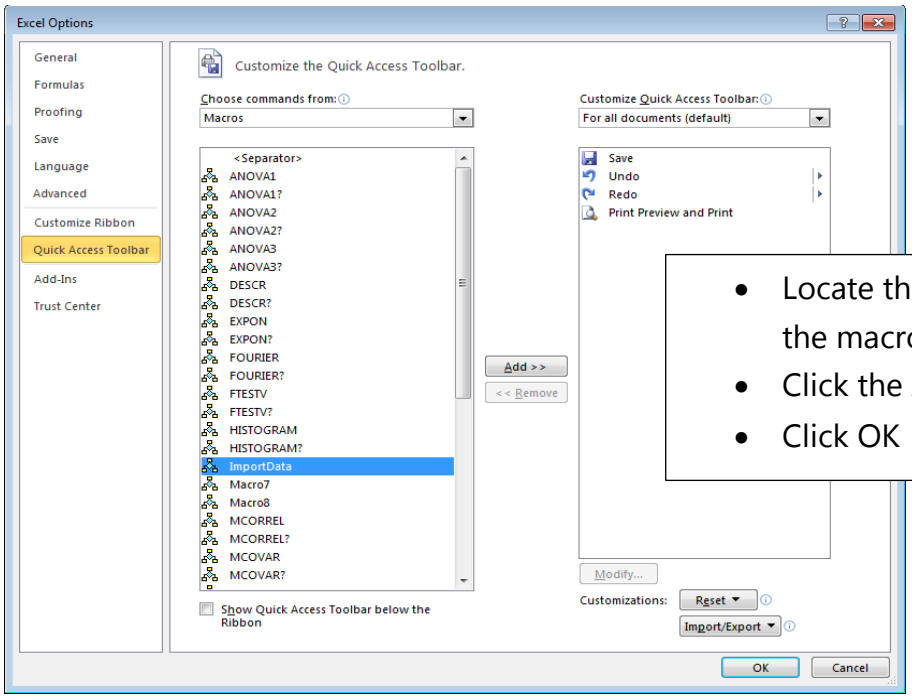
Adding a Macro/Procedure to the Quick Access Toolbar

To add the recorded Macro as a button on the Quick Access Toolbar, top left of the Excel window follow these steps:

Quick Access Toolbar



- Select the small drop menu button (shown above)
- From the menu select “More Commands”
- This will display the Excel Options dialog
- Click menu called “Choose Commands From”
- Select Macros



- Locate the name of the macro and select
- Click the Add button
- Click OK

The Personal Macro Workbook

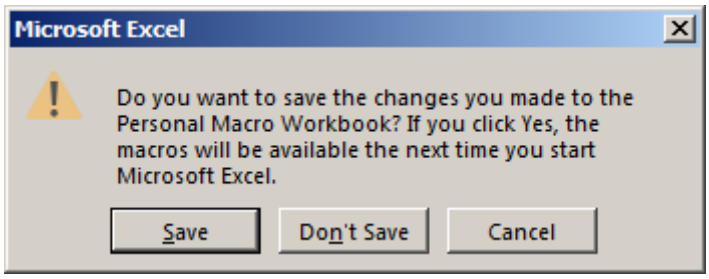
The personal macro workbook is automatically created by Excel the first time you record a macro into it. It is then stored in a trusted location as part of your personal profile. It is loaded when Excel is running, therefore macros stored here are always available.

When to create Personal Macros
How do you decide whether to store macros in the Personal Macros workbook or a particular workbook?

Macros which could be used in any workbook would be better saved into the Personal Workbook. For example a macro to create a page setup with a particular header and footer could then be run for any workbook. Another example might be a macro to change the selected text to upper or proper case.

Saving the Personal Macros

When exiting from Excel you will be given an opportunity to save the Personal Macro Workbook.



This will makes it available every time you use Excel. The workbook file is named Personal.xlsb.

Editing a Personal Macro

Personal macros are stored within a hidden workbook which makes them a little harder to edit. The easiest way without unhiding the workbook is to switch to the Visual Basic Editor (VBE) by pressing **Alt+F11**.

Assigning Personal Macros to the Toolbar

Once created it would be convenient to assign Personal macros to the Quick Access Toolbar.

Unit 2 Working with the Visual Basic Editor

In this unit you will learn how to:

- Understand the visual basic editor
- Navigate the tools
- Protect your code
- Understand the personal macro workbook

Introducing Visual Basic for Applications

Visual Basic for Applications or VBA is a development environment built into the Microsoft Office© Suite of products.

VBA is an Object Oriented Programming (OOP) language. It works by manipulating objects. In Microsoft™ Office® the programs are objects. In Excel worksheets, charts and dialog boxes are also objects.

In VBA the object is written first

I'm fixing the Yellow House = .House.Yellow.Fix

	House	**Yellow**	**Fix**
English	.noun	.adjective	.verb
VBA	.object	.property	.method

When working in VBA tell Excel exactly what to do. Don't assume anything.

Some General tips

Do not hesitate to use the macro recorder to avoid typos in your code. It will also allow you to get access to useful code without having to memorise it.

Write your code in lower case letters. If the spelling is RIGHT, the Visual Basic Editor will capitalize the necessary letters. If it doesn't.... check your spelling.

All VBA sentences must be on a single line. When you need to write long sentences of code and you want to force a line break to make it easier to read you must add a space and an underscore at the end of each line and then press Return. Here is an example of a single sentence broken into 3 lines:

```
Range("A1:E9").Sort Key:=Range("C2"), Order1:=xlAscending,  _
MatchCase:=False, Orientation:=xlTopToBottom, _
DataOption1:=xlSortTextAsNumbers
```

Flickering Screen

Running a macro or VBA code may cause the screen to flicker as the monitor is the slowest part of the program and cannot keep up with the very fast changes taking place. To switch off the screen until the program is run enter the following code line:

Application.ScreenUpdating = False

Screen comes on automatically on completion of the program.

CutCopyMode

After each Paste operation, you should turn off copying:

ActiveSheet.Paste

Application.CutCopyMode = False

DisplayAlerts

If you don't want Excel to ask you things like "Do you want to delete this file..." you can use the following line of code at the beginning of the relevant VBA procedure.

Application.DisplayAlerts = False

Then at the end make sure you use the following code to reactivate Display Alerts.

Application.DisplayAlerts = True

Compare Text

If you try to compare two strings in VBA the system compares the Binary information of the strings so that

"My Name" Is Not Equal To "my name".

To make the computer compare the words in the string, rather than the Binary you need to enter the code:

Option Compare Text

In the Declarations area of the module.

Quit

The following line of code closes Excel altogether.

Application.Quit

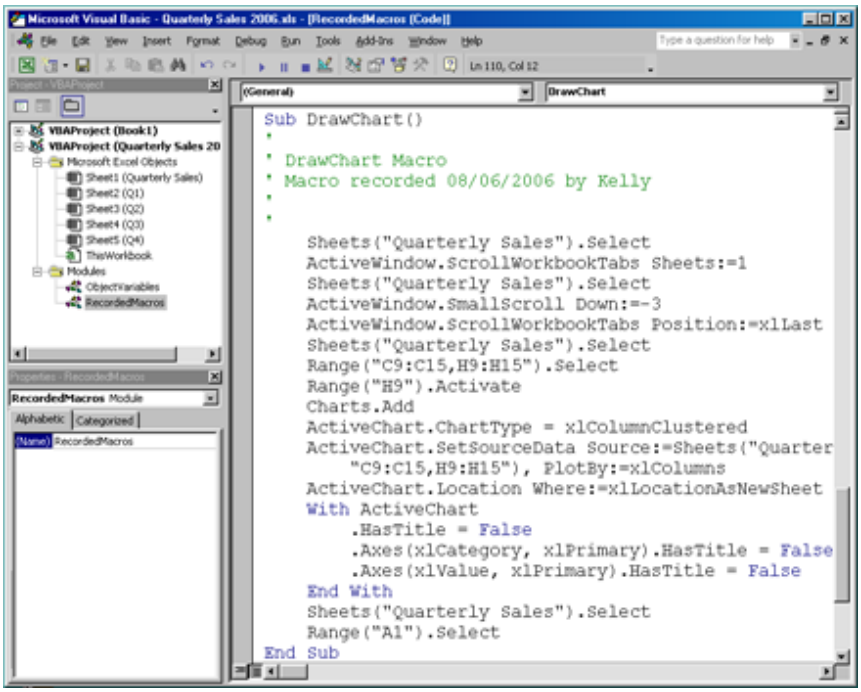
Editing Macros in Visual Basic Editor

When you record a macro, the recorded instructions are inserted into a Procedure whose beginning and end are denoted with the key words **Sub** and **End Sub**. This is stored within a Module. A module can contain many procedures.

Code generated when a macro is recorded can be modified to provide a more customised function. To do this:

- Developer Ribbon > Code Section > Macros
- Select the desired macro from the **Macro Name** list
- Click Edit

The **Visual Basic Editor** appears.

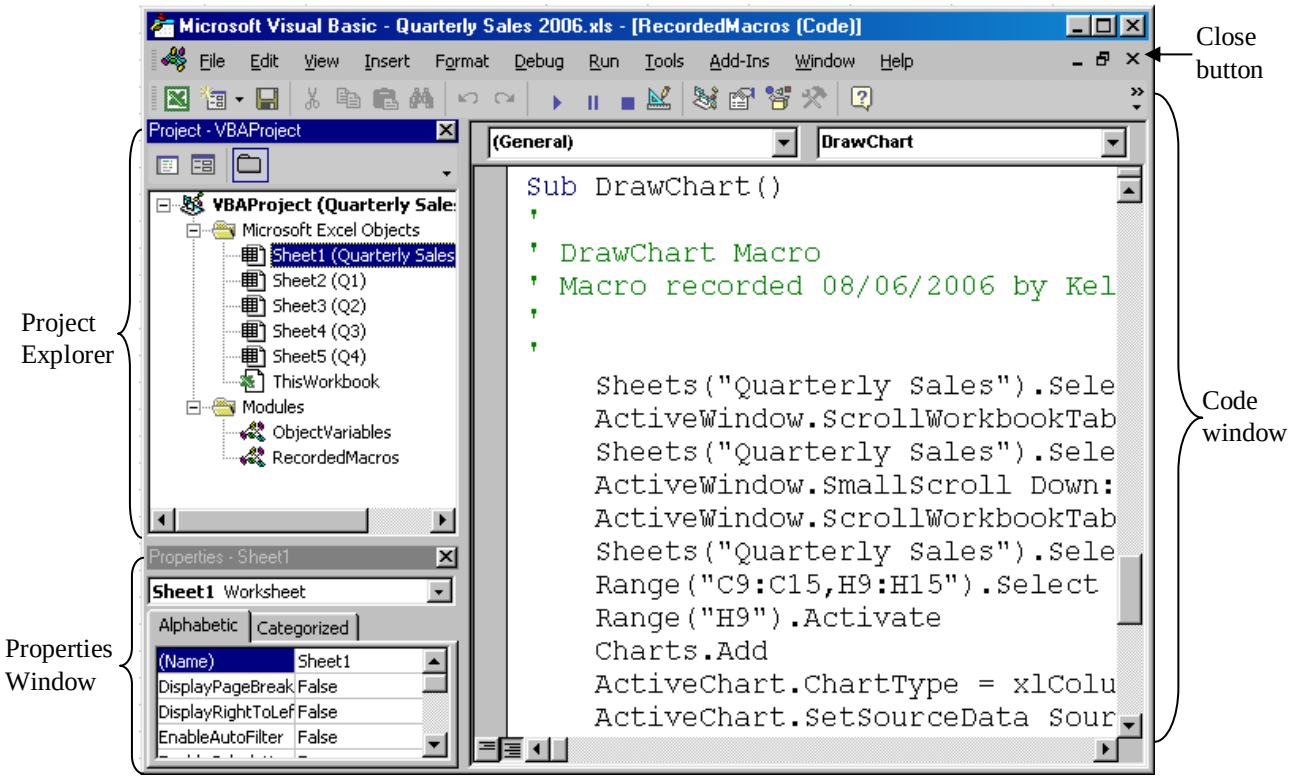


- Make the desired changes
- Save the macro
- Close the **Visual Basic Editor** window.

Important Note

You can usually figure out how to code any action in Excel by recording it in a macro and viewing the resulting macro code.

Understanding the Development Environment

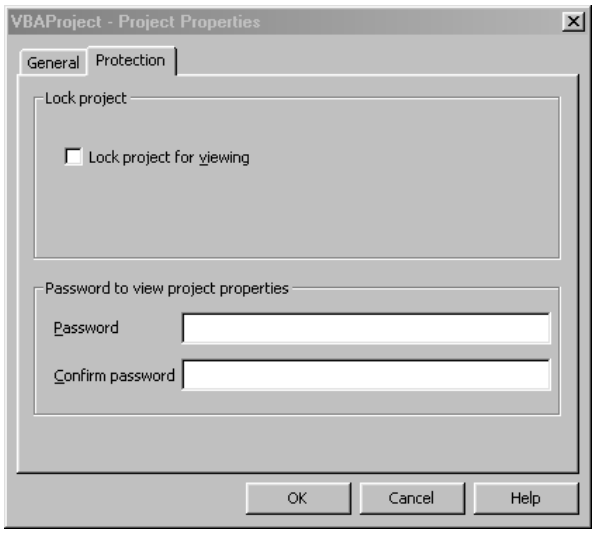


Title bar, Menu bar and Standard toolbar	The centre of the Visual basic environment. The menu bar and toolbar can be hidden of customized. Closing this window closes the program.
Project Explorer	Provides an organized view of the files and components belonging to the project. If hidden the Project Explorer can be displayed by pressing **Ctrl + R**
Properties Window	Provides a way to change attributes of forms and controls (e.g. name, colour, etc). If hidden press **F4** to display.
Code Window	Used to edit the Visual basic code. Press **F7** and it will open an object selected in Project Explorer. Close the window with the **Close** button that appears on the menu bar.

Protect/Lock Excel VBA Code

When we write VBA code it is often desirable to have the VBA Macro code not visible to end-users. This is to protect your intellectual property and/or stop users messing about with your code.

To protect your code, from within the Visual Basic Editor



- Open the **Tools** Menu
- Select **VBA Project Properties**

The Project **Properties** dialog box appears.

- Click the **Protection page tab**
- Check "**Lock project for viewing**"
- Enter your password and again to confirm it.
- Click **OK**

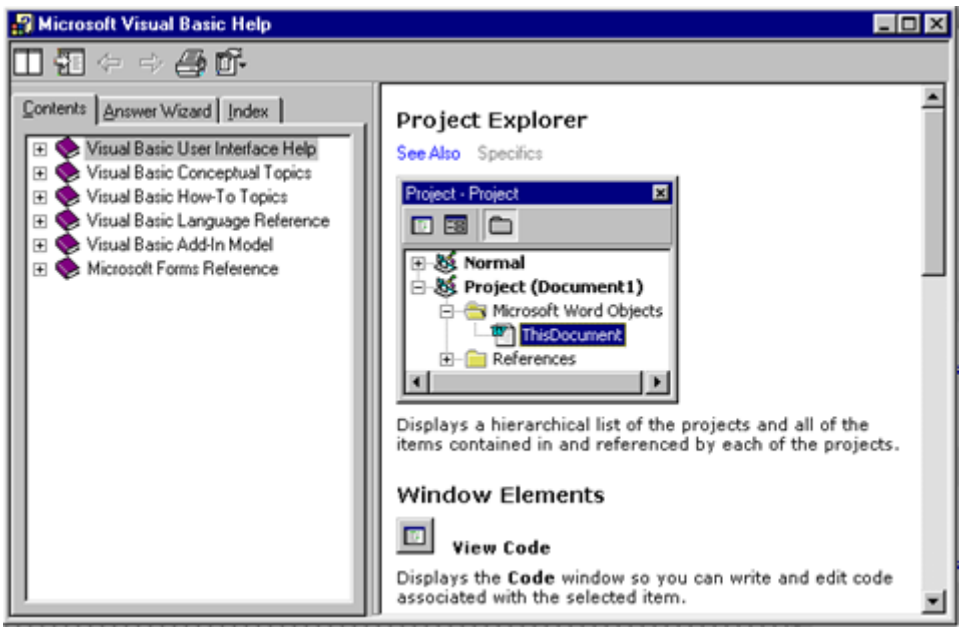
After doing this you must **Save and Close** the Workbook for the protection to take effect.

The safest password to use is one that uses a combination of upper, lower case text and numbers. Be sure not to forget it.

Notes

Using Help

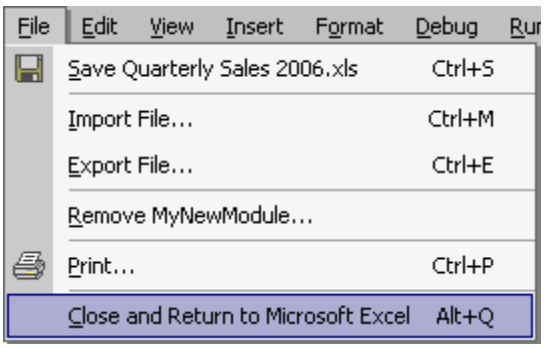
If the **Visual Basic Help** files are installed, by pressing **F1**, a help screen displays explaining the feature that is currently active:



Alternatively use the **Ask a Question** box on the menu bar to as a quick way to find help on a topic.

Closing the Visual Basic Editor

To close the **Visual Basic Editor** use one of the following:



- Open the **File** menu; select **Close and Return to Microsoft Excel**

OR

- Press **Alt + Q**

OR

- Click **Close** in the title bar.

Unit 3 Developing with Procedures

In this unit you will learn how to:

- Understand and create modules
- Navigate the tools
- Protect your code
- Understand the personal macro workbook

Procedure is a term that refers to a unit of code created to perform a specific task. In Excel, procedures are stored in objects called **Modules**.

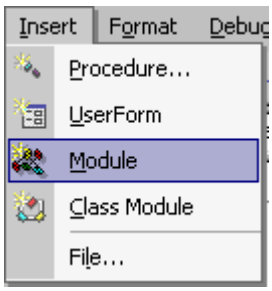
In this unit we will look at both Modules and Procedures.

Understanding and Creating Modules

Standard modules can be used to store procedures that are available to all forms, worksheets and other modules. These procedures are usually generic and can be called by another procedure while the workbook is open.

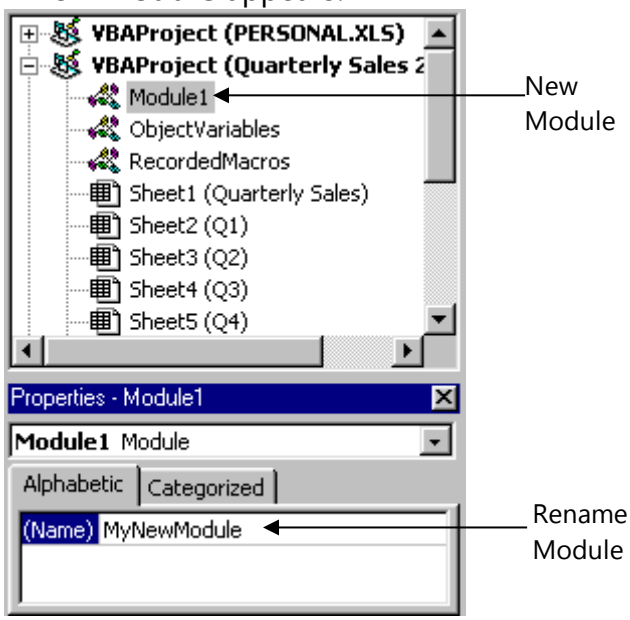
Within a project you can create as many standard modules as required. You should store related procedures together within the same module.

Standard modules are also used to declare global variables and constants. To create a standard module in the VB Editor:



- Open the **Insert** menu
- Select **Module**.

A new **Module** appears:



- Display the **Properties** window if necessary
- In the **Properties** window change the name of the module

Defining Procedures

A procedure is a named set of instructions that does something within the application.

To execute the code in a procedure you refer to it by name from within another procedure. This is known as Calling a procedure. When a procedure has finished executing it returns control to the procedure from which it was called.

There are two general types of procedures:

Sub procedures	perform a task and return control to the calling procedure
Function procedures	perform a task and return a value, as well as control, to the calling procedure

If you require 10 stages to solve a problem write 10 sub procedures. It is easier to find errors in smaller procedures than in a large one.

The procedures can then be called, in order, from another procedure.

Naming Procedures

There are rules and conventions that must be followed when naming procedures in Visual Basic.

While rules must be followed or an error will result, conventions are there as a guideline to make your code easier to follow and understand.

The following **rules** must be adhered to when naming procedures:

- Maximum length of the name is 255 characters
- The first character must be a letter
- Must be unique within a given module
- Cannot contain spaces or any of these characters: **. , @ & $ # ( ) !**

You should consider these naming **conventions** when naming procedures:

- As procedures carry out actions, begin names with a verb
- Use the proper case for the word within the procedure name
- If procedures are related try and place the words that vary at the end of the name

Following these conventions, here is an example of procedure names:

PrintClientList

GetDateStart

GetDateFinish

Creating a Sub-Procedure

Most Excel tasks can be automated by creating procedures. This can be done by either recording a macro or entering the code directly into the VB Editor's Code window.

Sub procedures have the following syntax:

```
[Public/Private] Sub ProcedureName ([argument list])

Statement block

End Sub
```

Public indicates procedure can be called from within other modules. It is the default setting

Private indicates the procedure is only available to other procedures in the same module.

The **Sub…End Sub** structure can be typed directly into the code window or inserted using the **Add Procedure** dialog box.

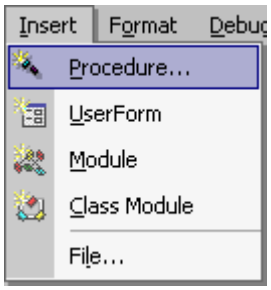
To create a sub procedure:

- Create or display the module to contain the new sub procedure
- Click in the **Code** window
- Type in the Sub procedure using the relevant syntax
  Type in the word Sub, followed by a space and the Procedure name
  Press **Enter** and VB inserts the parenthesis after the name and the End Sub line.

OR

- Use **Add Procedure**.

To display the **Add Procedure** dialog box:



- Open the **Insert** menu
- Select **Procedure**.

The **Add Procedure** dialog box appears:

- Type the name of the procedure in the **Name** text box
- Select **Sub** under **Type**, if necessary
- Make the desired selection under **Scope**
- Click **OK**.

Below is an example of a basic sub procedure:

```
Sub Welcome()
    MsgBox "Hello User, How are you"
End Sub
```

Notes

Auto Quick Info is a feature of the Visual Basic that displays a syntax box when you type a procedure or function name.

The example below shows the tip for the Message Box function:

Arguments in square brackets are optional.

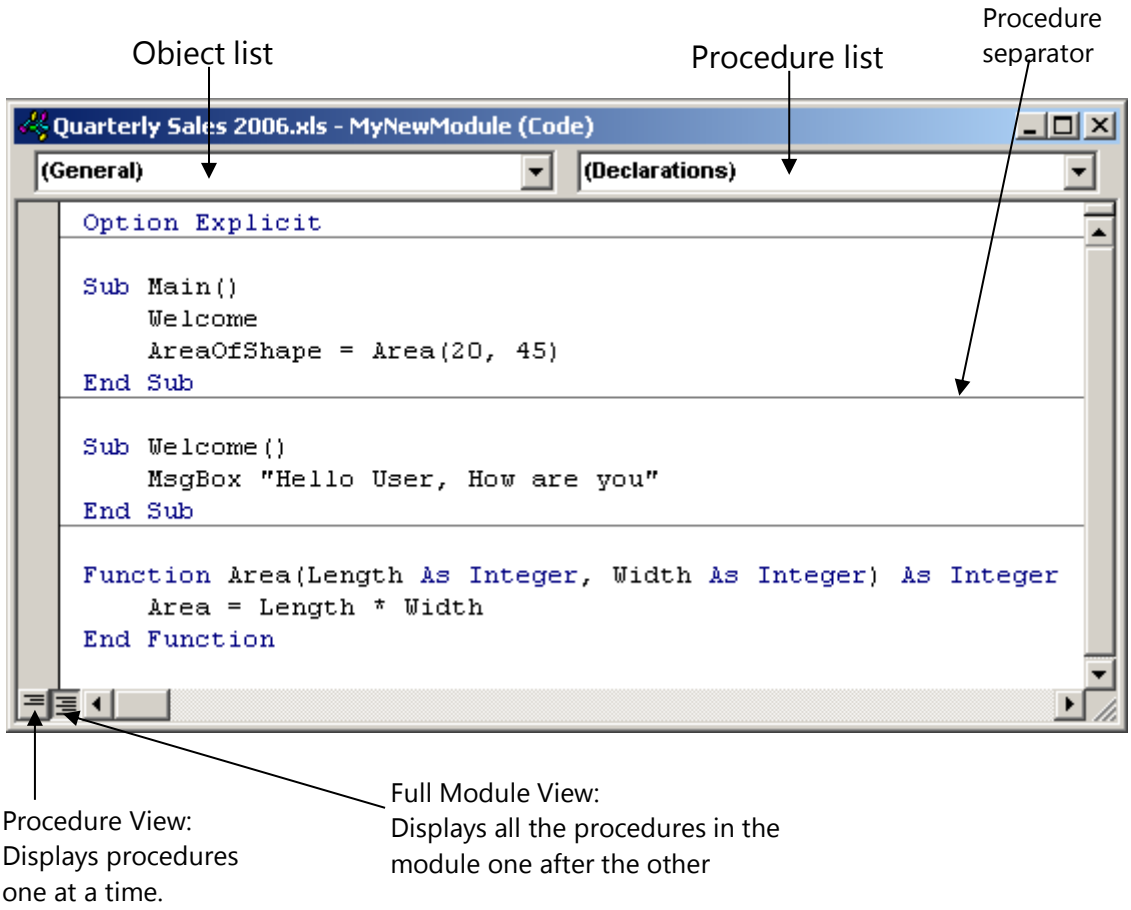
Values passed to procedures are sometimes referred to as parameters.

Notes

Working Using the Code Editor

The Code editor window is used to edit Visual Basic code. The two drop down lists can be used to display different procedures within a standard module or objects' event procedures within a class module.

Below is an illustration of the code window:

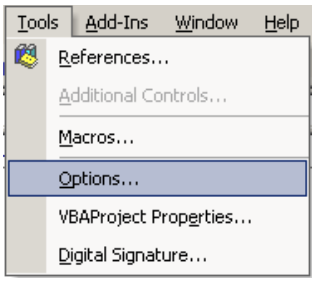


Object List	Displays a list of objects contained in the current module.
Procedure List	Displays a list of general procedures in the current module when General is selected in the Object list. When an object is selected in the Object list it displays a list of events associated with the object.

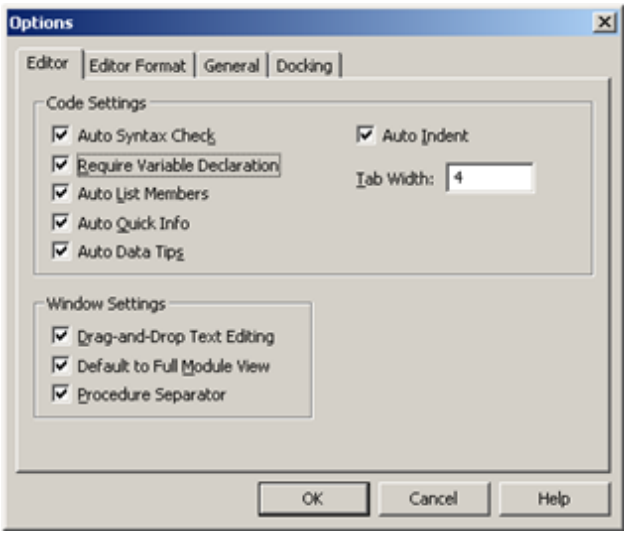
Notes

Setting Code Editor Options

The settings for the **Code Editor** can be changed. To do this:



- Open the **Tools** menu in the **VB Editor**
- Select **Options**.



The **Options** dialog box appears:

The following are explanations of the **Code Setting** selections:

Auto Syntax Check	Automatically displays a **Help** message when a syntax error is detected. Message appears when you move off the code line containing the error
Require Variable Declaration	Adds the line ***Option Explicit*** to all newly created modules, requiring all variables to be explicitly declared before they are used in a statement.

Auto List Members	Displays a list box under your insertion point after you type an identifiable object. The list shows all members of the object class. An item selected from the list can be inserted into your code by pressing the **Tab** key
Auto Quick Info	Displays a syntax box showing a list of arguments when a method, procedure or function name is typed
Auto Data Tips	Displays the value of a variable when you point to it with a mouse during break mode. Useful for debugging.
Auto Indent	Indent the specified amount when **Tab** is pressed and indents all subsequent lines at the same level.

The **Windows Settings** selections are explained below:

Drag-and-Drop Text Editing	Allows you to drag and drop code around the Code window and into other windows like the Immediate window.
Default to Full Module View	Displays all module procedures in one list with optional separator lines between each procedure. The alternative is to show one procedure at a time, as selected through the Procedure list.
Procedure Separator	Displays a grey separator line between procedures if Module view is selected

Editing Guidelines

Below are some useful guidelines to follow when editing code:

- If a statement is too long carry it over to the next line by typing a space and underscore (_) character at the end of the line. This also works for comments.

  Strings that are continued require a closing quote, an ampersand (&), and a space before the underscore. This is called ***Command Line Continuation***.

- Indent text within control structures for readability. To do this:
  - Select one or more lines
  - Press the **Tab** key **OR**
  - Press **Shift + Tab** to remove the indent.

- Complete statements by pressing **Enter** or by moving focus off the code line by clicking somewhere else with the mouse or pressing an arrow key.

  When focus is moved off the code line, the code formatter automatically places key words in the proper case, adjusts spacing, adds punctuation and standardizes variable capitalization.

It is also a good idea to comment your code to document what is happening in your project. Good practice is to comment what is not obvious.

Start the line with an apostrophe (') or by typing the key word **Rem** (for remark). When using an apostrophe to create a comment, you can place the comment at the end of a line containing a code statement without causing a syntax error.

Notes

Unit 4 Managing Program Execution

In this unit you will learn how to:

- Understand how Excel execute the code
- Use bolean expressions
- Use decision code (If & Select Case)
- Loop through objects

Defining Control-Of-Flow structures

When a procedure runs, the code executes from top to bottom in the order that it appears. Only the simplest of programs execute in this manner. Most programs incorporate logic to control which lines of code to execute.

The **Control-Of-Flow** structures described below provide this logic:

Sequential	Each line of code is executed in order from top to bottom.
Unconditional Branching	A statement that directs the flow of program execution to another location in the program without condition. Calling a **Function**, a **Sub** or using the **GoTo** statement are examples of unconditional branching
Conditional Branching	The code to be executed is based on the outcome of a **Boolean** expression. Decision structures like **If** and **Select Case** are used to implement conditional branching.
Looping	A block of code executed repeatedly as long as a certain condition exists. The **For…Next** and the **Do..Loop** are examples of looping structures
Halt Statements	Commands used to stop code execution. The **Stop** command stops execution but retains variables in memory. The **End** command terminates the application.

Using Boolean Expressions

A **Boolean** expression returns a True or False value. Many **Boolean** expressions take the form of two expressions either side of a comparison operator. If the result is true the condition is met and control is passed to the code to be executed.

Here are some examples of **Boolean** expressions:

```
Firstname = "Alan"

UnitPrice > 1.60

OrderAmount < 500
```

The following comparison operators are used in **Boolean** expressions:

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
=	Equal to
<>	Not equal to
Is	Compares object variables
Like	Compares string expressions

When testing for more than one condition **Boolean** expressions can be joined with a **Logical Operator**.

The following is a list of **Logical Operators**:

And	Each expression must be True for the condition to be true.
Or	One of the expressions must be True for the condition to be true.
Not	The expression must be False for the condition to be true.

The following are examples of multiple conditions joined by logical operator:

```
UnitPrice > 1.60 AND OrderAmount > 1000

DateJoined <= 2004 OR DeptName = "Sales"
```

A null expression will be treated as a false expression.

Notes

Using the If…End If Decision Structures

If…End If is used to execute one or more statements depending upon a text condition. There are four forms of the **If** construct.

The first contains the condition and statement to be executed in the same line:

```
If <condition> Then <statement>

If OrderAmount >1000 Then Discount = "Yes"
```

The block form is used when several statements are to be executed based on result of the test condition:

```
If <condition> Then
<statement block>
End If

If Country = "England" Then

Account = "Domestic"

TransportCost = 10.00

End If
```

Like the **If…Then** structure the **If…Then…Else** structure passes control to the statement block that follows the **Then** keyword when the condition is **True** and passes control to the statement block that follows the **Else** keyword when the condition is **False**.

```
If <condition> Then
```

```
<statement block>
Else
<statement block>
End If


If Country = "England" Then

Account = "Domestic"

TransportCost = 10.00

Else

Account = "Foreign"

TransportCost = 40.00



End If
```

By modifying the basic structure and inserting **ElseIf** statements, an **If…Then…Else** block that tests multiple conditions is created. The conditions are tested in the order of appearance until a condition is true.

If a true condition is found, the statement block following the condition is performed; execution then continues with the first line of code following the **End If** statement. If no condition is true, execution will continue with the **End If** statement. An optional **Else** clause at the end of the block will catch the cases that do not meet any of the conditions.

```
If <condition_1> Then
<statementBlock1>
[ElseIf <condition_2> Then
[<StatementBlock2>]]
[ElseIf <condition_3> Then
[<StatementBlock3>]]
[ElseIf <condition_N> Then
[<StatementBlockN>]]
End If
```

```
If Country = "England" Then

Account = "Domestic"

TransportCost = 10.00

ElseIf Country = "Wales" Then

Account = "Domestic"

TransportCost = 20.00

ElseIf Country = "Scotland" Then

Account = "Domestic"

TransportCost = 25.00

ElseIf Country = "Northern Ireland" Then

Account = "Domestic"
```

```
TransportCost = 30.00

Else

Account = "Foreign"

TransportCost = 40.00


End If
```

Using the Select Case…End Select Structure

The **Select Case** statement is often used in place of the complex **If** statement. The advantage of using this style is that your code will be more readable and efficient. The downside is that it is only useful if compared against just one value.

The **Select Case** structure contains the test expression in the first line of the block. Each **Case** statement in the structure then compares against the test expression.

The syntax of the **Select Case** structure, followed by two examples is shown below:

Notes

```
Select Case <TestExpression>
Case <Expression_1>
<StatementBlock1>
Case <Expression_2>
<StatementBlock2>
Case <Expression_3>
<StatementBlock3>
Case <Expression_N>
<StatementBlockN>
End Select
```

```
Select Case Country

Case "England"

Account = "Domestic"

TransportCost = 10.00

Case "Wales"

Account = "Domestic"

TransportCost = 20.00

Case "Scotland"

Account = "Domestic"

TransportCost = 25.00

Case "Northern Ireland"

Account = "Domestic"

TransportCost = 30.00

Case Else

Account = "Foreign"
```

```
TransportCost = 40.00

End Select


Select Case TestScore

Case 0 To 50

Result = "Below Average"

Case 51 To 70

Result = "Good"

Case Is > 70

Result = "Excellent"

Case Else

Result = "Irregular Test Score"

End Select
```

Using the Do…Loop Structure

The **Do…Loop** structure controls the repetitive execution of the code based upon a test of a condition. There are two variations of the structure: **Do While** and **Do Until**.

The **Do While** structure executes the code as long as the condition is true.
The **Do Until** structure executes the code up to the point where the condition becomes true or as long as the condition is false. The condition is any expression that can be evaluated to true or false.

The **Exit Do** is optional and can be used to quit the **Do** statement and resume execution with the statement following the Loop. Multiple **Exit Do** statements can be placed anywhere within the Loop construct.

The following syntax is used to perform the statement block zero or more times:

```
Do While <condition>
<statement block>
[Exit Do]
Loop


Do Until <condition>
<statement block>
[Exit Do]
Loop


Do While ActiveCell.Value <> ""

ActiveCell.Value = ActiveCell.Value *1.25

ActiveCell.Offset(1).Select

Loop
```

To perform the statement block at least once, use one of the following:

```
Do
```

<statement block>
[Exit Do]
Loop While <*condition*>

Do
<statement block>
[Exit Do]
Loop Until <*condition*>

Do

Count = Count +1

Loop Until Count = NoStudents

Using The For…Next Structure

The **For…Next** structure executes a block of statements a specific number of times using a counter that increases or decreases values. Beginning with the start value, the counter is increased or decreased by the increment. The default increment is 1. Specify an increment of -1 to count backwards.

The **Exit For** statement is optional and can be used to quit the **For** construct and resume execution with the statement following the **Next**.

Below is the syntax of the **For…Next** statement:

```
For <counter> = <start> To <end> [Step <increment>
<statement block>
[Exit For]
Next [<counter>]

Dim MyIndex as Integer
For MyIndex = 1 To NoRows
Cells (MyIndex,4).Select
Total = Total + Cells (NoRows,4).Value
Next MyIndex
```

Using the For Each…Next Structure

The **For Each…Next** structure is used primarily to loop through a collection of objects. With each loop it stores a reference to a given object within the collection to a variable. The variable can be used by the code to access the object's properties. By default it will loop through ALL the objects in a collection.

The **Exit For** statement is optional and can be used to quit the **For Each** construct and resume execution with the statement following the **Next**.

Below is the syntax of the **For Each…Next** statement:

```
For Each <element> in <CollectionReference>
<statement block>
[Exit For]
Next [<element>]
```

```
Dim BookVar As Workbook

For Each BookVar In Application.Workbooks
BookVar.Save
Next BookVar
```

Using the While Wend loop

The While Wend loop is an old loop which are not used much anymore but you can still find it, when you Google Excel VBA.

Below is the syntax of the **While…Wend** statement:

```
While <condition>

<statement>

Wend
```

An Example could look like this:

```
While ActiveCell>100

ActiveCell.Offset(0,2).Value=”Value is greater than 100”

Wend
```

As you can see in the example the loop is working very similar as the Do loop.

Guidelines for Use Of Control-Of-Flow Structures

Use the following as a guide in choosing the appropriate **Decision** structure:

Use	To
If…Then Or If…Then…End If	Execute one statement based on the result of one condition
If…Then…End If	Execute a block of statements based on the result of one condition
If…Then…Else…End If	Execute 1 of 2 statement blocks based on the result of one condition
Select Case…End Select	Execute 1 of 2 or more statement blocks based on 2 or more conditions, with all conditions evaluated against 1 expression.
If…Then…ElseIf…End If	Evaluate 1 of 2 or more statement blocks based on 2 or more conditions, with conditions evaluated against 2 or more expressions.

Use the following as a guide in choosing the appropriate **Looping** structure:

Use	To
For…Next	Repeat a statement block a specific number of times. The number is known or calculated at the beginning of the loop and doesn't change.
For…Each	Repeat a statement block for each element in a collection or array.
For…Next	Repeat a statement block while working through a list when the number of list items is known or is calculated beforehand.

Do…Loop	Repeat a statement block while working through a list when the number of list items is not known or are likely to change.
Do…Loop	Repeat a statement block while a condition is met.

Using loops to easily make changes across multiple worksheets

Loops are one of the key tools in Excel VBA when we need to perform tasks through a number of objects (cells, worksheets, charts, workbooks etc.) . Here we will look at how to loop through Excel worksheets and workbooks.

Loop Through Excel Worksheets

Below you will find three examples using different loops but all three will perform exactly the same task.

The For Each loop

1. An object variable (sh) is used and declared as Worksheet to tell Excel that we want store worksheets (the address) in the memory of our computer (Dim sh As Worksheet).
2. The For Each loop will loop through each worksheet in the active workbook (For Each sh In ActiveWorkbook.Sheets).
3. The code will add 500 in A1 in all sheets in the active workbook.

```
Sub LoopSheets()
Dim sh As Worksheet
For Each sh In ActiveWorkbook.Sheets
sh.Range("A1").Value = 500
Next sh
End Sub
```

The For Next loop

1. A data variable is used to store a whole number (integer) in the computer's memory (Dim iCounter As Integer).
2. The For Next loop is used to loop through all sheets in the active workbook but the loop needs to know how many worksheets there is in the active workbook (ActiveWorkbook.Sheets.Count).
3. The iCounter variable is used to move through the worksheets and the value 500 is entered in A1 in all worksheets in the active workbook (Sheets(iCounter).Range("A1").Value = 500).

```
Sub LoopSheetst2()
Dim iCounter As Integer
For iCounter = 1 To ActiveWorkbook.Sheets.Count
Sheets(iCounter).Range("A1").Value = 500
Next iCounter
End Sub
```

The Do loop

1. A data variable is used to store a whole number (integer) in the computer's memory (Dim iCounter As Integer).
2. 1 is stored in the iCounter variable (iCounter = 1).
3. A Do Until loop is used to run until criteria is met in this example until the value in the variable iCounter is total number of worksheets in the active workbook plus one (Do Until iCounter = ActiveWorkbook.Worksheets.Count + 1).

```
Sub LoopSheets3()
Dim iCounter As Integer
iCounter = 1
Do Until iCounter = ActiveWorkbook.Worksheets.Count + 1
Sheets(iCounter).Range("A!").Value = 500
iCounter = iCounter + 1
Loop
End Sub
```

Loop Workbooks

Below you will find three examples using different loops but all three will perform exactly the same task but this time the loops will loop through workbooks.

The For Each loop

1. An object variable (wBook) is used and declared as Workbook to tell Excel that we want store workbooks (the address) in the memory of our computer (Dim WBook As Workbook).
2. The **For Each** loop will loop through each open workbook (For Each wBook In Workbooks).
3. The code will add 2 in A2 in sheet 1 in all open workbooks.

```
Sub LoopWorkBooks()
Dim WBook As Workbook
For Each WBook In Workbooks
WBook.Sheets(1).Range("A2").Value = 2
Next WBook
End Sub
```

The For Next loop

1. A data variable is used to store a whole number (integer) in the computer's memory (Dim iWB As Integer).

2. The **For Next** loop is used to loop through all open workbooks but the loop needs to know how many open workbooks we have (Workbooks.Count).
3. The iWB variable is used to move through the open workbooks and the value 2 is entered in A2 in sheet 1 in all open workbooks (Workbooks(iWB).Sheets(1).Range("A2").Value = 2).

```
Sub LoopWorkBooks2()
Dim iWB As Integer
For iWB = 1 To Workbooks.Count
Workbooks(iWB).Sheets(1).Range("A2").Value = 2
Next iWB
End Sub
```

The Do loop

1. A data variable is used to store a whole number (integer) in the computer's memory (Dim iCounter As Integer).
2. 1 is stored in the iCounter variable (iCounter = 1).
3. A **Do Until** loop is used to run until the criteria is met in this example until the value in the variable iCounter is total number of open workbooks plus one (Do Until iCounter = Workbooks.Count + 1).

```
Sub LoopWorkBooks3()
Dim iCounter As Integer
iCounter = 1
Do Until iCounter = Workbooks.Count + 1
Workbooks(iCounter).Sheets(1).Range("A2").Value = 2
iCounter = iCounter + 1
Loop
End Sub
```

Loop workbooks & worksheets

In the examples below nested loops are looping through workbooks and worksheets and again the **For Each**, **For Next** and the **Do** loop are used to do the job.

The For Each loop

Exactly as in the examples above in this post variables are used to store the address of the workbooks and worksheets in the computers memory (Dim WBook As Workbook & Dim sh As Worksheet). A **For Each** loop is used to run through the workbooks and one to run through the worksheets.

```
Sub LoopWorkBookSheets()
Dim WBook As Workbook
Dim sh As Worksheet
For Each WBook In Workbooks
For Each sh In WBook.Worksheets
sh.Range("a1") = 2
Next sh
```

```
Next WBook
End Sub
```

The For Next loop

Two **For Next** loops are needed to run through all worksheets in all open workbooks. Two variables are used (counter variables) to loop one workbook at the time and one worksheet.

```
Sub LoopWorkBookSheets2()
Dim iWB As Integer
Dim iCounter As Integer
For iWB = 1 To Workbooks.Count
For iCounter = 1 To Workbooks(iWB).Sheets.Count
Workbooks(iWB).Sheets(iCounter).Range("b1").Value = 450
Next iCounter
Next iWB
End Sub
```

The Do loop

It takes more coding to run through all worksheets in all open workbooks by using the **Do** loop. Again two loops are needed one for the workbooks and one for the worksheets.

```
Sub LoopWorkBookSheets3()
Dim iWorkBookCounter As Integer
Dim iSheetCounter As Integer
iWorkBookCounter = 1
iSheetCounter = 1
Do Until iWorkBookCounter = Workbooks.Count + 1
Do Until iSheetCounter =
Workbooks(iWorkBookCounter).Sheets.Count + 1
Workbooks(iWorkBookCounter).Sheets(iSheetCounter).Rang
e("c1").Value = 5
iSheetCounter = iSheetCounter + 1
Loop
iWorkBookCounter = iWorkBookCounter + 1
iSheetCounter = 1
Loop
End Sub
```

Some people prefer to use the **For Each** loop for a couple of reasons. The **For Each** loop is a faster loop and normally you need less coding.

Unit 5 Debugging the Code

In this unit you will learn how to:

- Understand errors
- Navigate the tools
- Protect your code
- Understand the personal macro workbook

Understanding Errors

When developing code, problems will always occur. Wrong use of functions, overflow and division by zero are some of the things that will cause an error and not produce the intended results.

Errors are called ***Bugs***. The process of removing bugs is known as ***Debugging***. VBA provides tools to help see how the code is running.

There are three general types of errors:

Syntax Errors

Syntax errors occur when code is entered incorrectly and is typically discovered by the line editor or the compiler.

- **Discovered by Line Editor**: When you move off a line of code in the Code window, the syntax of the line is checked. If an error is detected the whole line turns red by default indicating the line needs to be changed.
- **Discovered by Compiler**: While the line editor checks one line at a time, the compiler checks all the lines in each procedure and all declarations within the project. If **Option Explicit** is set, the compiler also checks that all variables are declared and that all objects have references to the correct methods, properties and events. The compiler also checks that all required

statements are present, for example that each **If** has an **End If**. When the compiler finds an error it displays a message box describing the error.

Run-Time Errors

When a program is running and it encounters a line of code that it cannot be executed, a run-time error is generated. These errors occur when a certain condition exists. A condition could run fine 10 times but cause an error on the 11th. When a run-time error occurs, execution is halted a message box appears defining the error.

Logic Errors

Logic errors create unexpected outcomes when a procedure is executed. Unlike syntax or run-time errors the application is not halted and you are not shown the offending line of code. These errors are more difficult to locate and correct.

Minimizing Errors

Here are a few suggestions to help you minimize or make it easier to find errors in your code:

- Add comments to code explaining what a line of code or procedure is meant to do. This is important if other people are going to look at the code.
- Create meaningful variable names. Use prefixes to identify data or object type.
- Any time you use division that contains a variable in the denominator, test the denominator to ensure that it doesn't equal zero
- Force variable declarations with the use of **Option Explicit**. A simple misspelling of a variable name will lead to a logic error, not a run-time error.
- Give procedures names that clearly describe what they do.
- Keep procedures as short as possible, giving it one or two specific tasks to carry out.
- Test procedures with large data sets representing all possible permutations of reasonable or unreasonable data. Make your procedure fail before someone else does.

Notes

Using Debugging Tools

VBA's debugging tools are useful for checking and understanding the cause of logic and run-time errors in the code.

The toolbar buttons as they appear left to right are explained below:

Design Mode	Turns design mode off and on.
Run / Continue	Runs code or resumes after a code break
Break	Stops the execution of a program while it's running and switches to Break Mode.
Reset	Clears the execution stack and module level variables and resets the project.
Toggle Breakpoint	Sets or removes a Break Point at the current line.
Step Into	Executes code one statement at a time.
Step Over	Allows selected ode to be stepped over during execution.
Step Out	Executes the remaining lines of a procedure after a break
Locals Window	Displays the value of variables and properties during code execution
Immediate Window	Displays a window where individual lines of code can be executed and variables evaluated.
Watch Window	Displays the value of each expression that is added to a window.
Quick Watch	Displays the current value of the selected expression.

Call Stack	Displays all the currently loaded procedures

Debugging is done when the application is suspended (in **Break Mode**). Everything loaded into memory remains in memory and can be evaluated. A program enters Break mode in one of the following ways

- A code statement generates a run-time error
- A breakpoint is intentionally set on a line of code
- A **Stop** statement is entered within the program code.

Identifying the Value of Expressions

While debugging it is useful to find out the value of variables and expressions while your code is executing.

VBA has the **Locals Window**, **Immediate Window**, **Watch Window** and **Quick Watch**, described in **Using Debugging Tools** on the previous page, which can be used to find the values of expressions

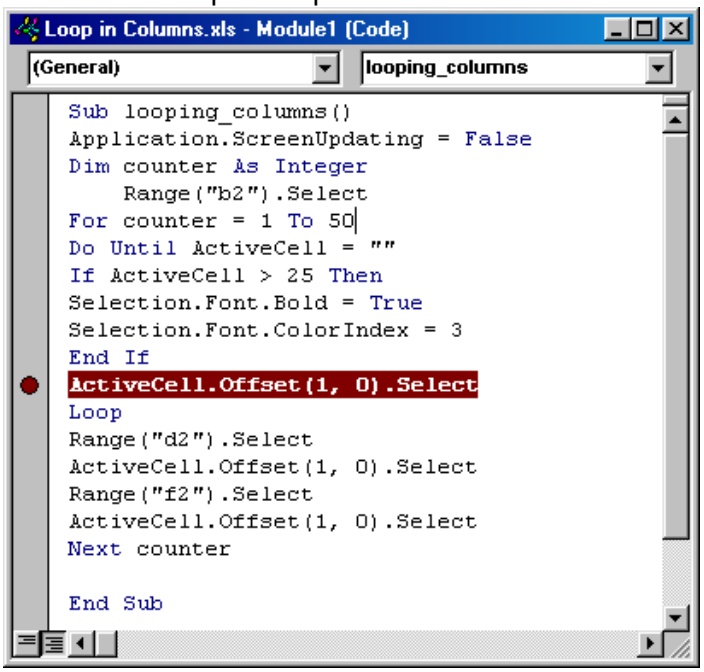
Another quick way of finding out the value of variables and expressions is the **Auto Data Tip** which displays the value of the expression where the mouse is pointing.

Setting Breakpoints

Setting breakpoints allows you to identify the location where you want your program to enter into break mode. The program runs to the line of code and stops. The code window displays and the line of code where the break point is set is highlighted.

When the code is halted, the value of a variable or expression can be checked by holding the mouse pointer over the expression or in the immediate window.

To set a breakpoint open the code window and select the desired procedure:



- Position the insert point on the desired line of code
- Set the breakpoint by clicking **Toggle Breakpoint** on the **Debug toolbar**

OR

- Open the **Debug menu** and select **Toggle Breakpoint**

OR

- Click in the grey area to the left of the line of code

How to Step Through Code

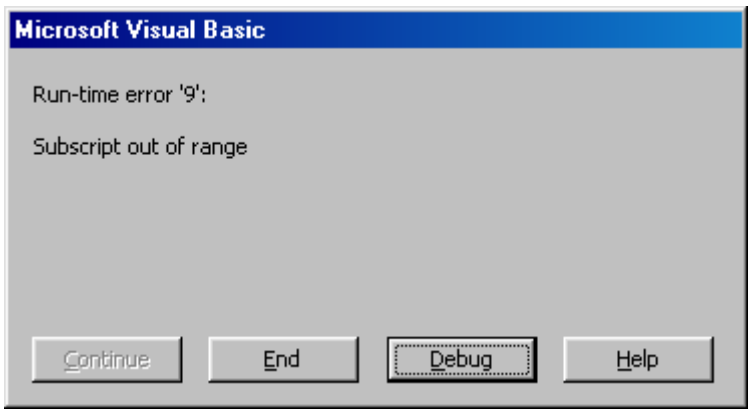
The step tools allow you to step one line at a time through the code to see exactly which statements in your procedure are being executed.

Step Into	F8	Executes code one statement at a time. If the statement calls another procedure execution steps into the called procedure and continues to execute one step at a time.
Step Over	Shift + F8	Executes code one statement at a time. If the statement calls another procedure the procedure is executed without pausing.
Step Out	Ctrl + Shift + F8	Executes the remaining lines of a procedure without pausing.
Run To Cursor	Ctrl + F8	Runs from the current statement to the location of the cursor in the Code window if you are stepping through code.
Set next Statement	Ctrl + F9	Runs the statement of your choice rather than the next statement.
Call Stack	Ctrl + L	Displays all the currently active procedures in the application that have started but are not completed.

Notes

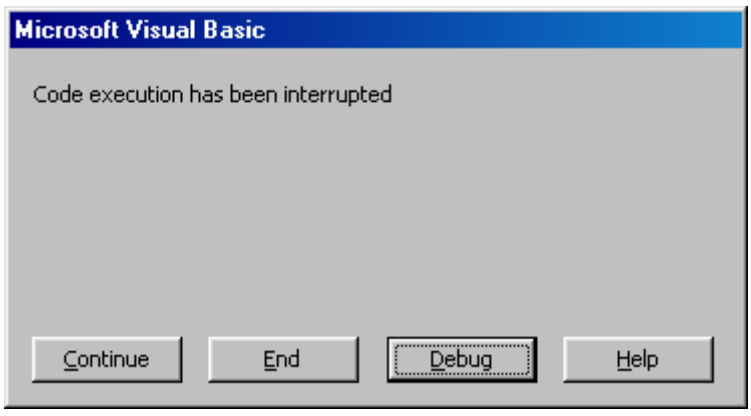
Working with Break Mode during Run Mode

During code execution the program can enter into Break Mode either intentionally or because of a run-time error. When a run-time error occurs a message appears that describes the error.



Click the **Debug** button to display the code window with the offending line highlighted.

If during the program execution you need to intervene, for example it's stuck in an endless loop, you can do so by pressing **Ctrl + Break** or the **Break button** in the **Visual Basic Editor**. It is also possible to break pressing **Esc** twice quickly.



That action will suspend the program execution and produce the following message:

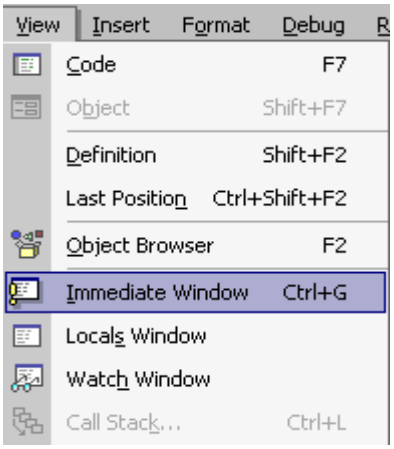
Notes

Using the Immediate Window

The **Immediate window** is a debugging feature of Visual Basic. It can be used to enter commands and evaluate expressions.

Code stored in a sub or function procedure can be executed by calling the procedure from the **Immediate window**.

To open the Immediate window:



- Open the **View** menu
- Select **Immediate window**

OR

- Press **Ctrl+G**.

The Immediate window appears.

To execute a sub procedure:

- Type **SubProcedureName ([Argument list])**
- Press **Enter**.

To execute a function and print the return value in the window:

- Type **? FunctionName ([Argument list])**
- Press **Enter**.

To evaluate an expression:

- Type **? Expression**
- Press **Enter**.

Within the code, especially in loops, use the **Debug.Print** statement to display values in the Immediate window while the code is executing. The Immediate window must be open for this.

Notes

Unit 6 Understanding Objects

In this unit you will learn how to:

- Understand the visual basic editor
- Navigate the tools
- Protect your code
- Understand the personal macro workbook

An object is an element of an application that can be accessed and manipulated using Visual Basic. Examples of objects in Excel are worksheets, charts and ranges.

Defining Objects

Objects are defined by lists of **Properties**, and **Methods**. Many also allow for custom sub-procedures to be executed in response to **Events**.

The term **Class** refers to the general structure of an object. The class is a template that defines the elements that all objects within that class share.

Properties

Properties are the characteristics of an object. The data values assigned to properties describe a specific instance of an object.

A new workbook in Excel is an instance of a Workbook object, created by you, based on the Workbook class. Properties that define an instance of a Workbook object would include its name, path, password, etc.

Methods

Methods represent procedures that perform actions.

Printing a worksheet, saving a workbook selecting a range are all examples of actions that can be executed using a method.

Events

Many objects can recognize and respond to events. For each event the object recognizes you can write a sub procedure that will execute when the specific event occurs.

A workbook recognizes the Open event. Code inserted into the Open event procedure of the workbook will run whenever the workbook is opened.

Events may be initiated by users, other objects, or code statements. Many objects are designed to respond to multiple events.

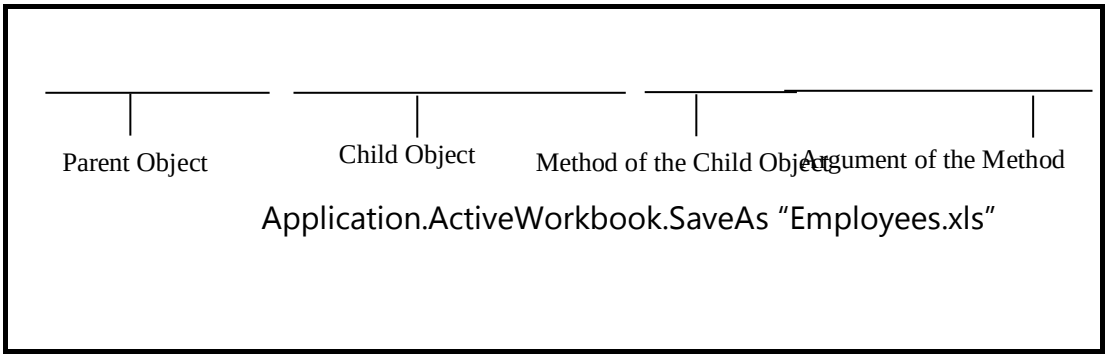
Notes

Examining the Excel Object Hierarchy

The Excel Object Module is a set of objects that Excel exposes to the development environment. Many objects are contained within other objects. This indicates a hierarchy or parent-child relationship between the objects.

The Application object represents the application itself. All other objects are below it and accessible through it. It is by referencing these objects, in code, that we are able to control Excel.

Objects, their properties and methods are referred to in code using the "dot" operator as illustrated below:



Some objects in Excel are considered global. This means they are on top of the hierarchy and can be referenced directly. The Workbook object is a child object of the Excel Application object. But since the Workbook object is global you don't need to specify the Application object when referring to it.

Therefore the following statements are equal:

```
Application.ActiveWorkbook.SaveAs "Employees.xls
```

```
ActiveWorkbook.SaveAs "Employees.xls"
```

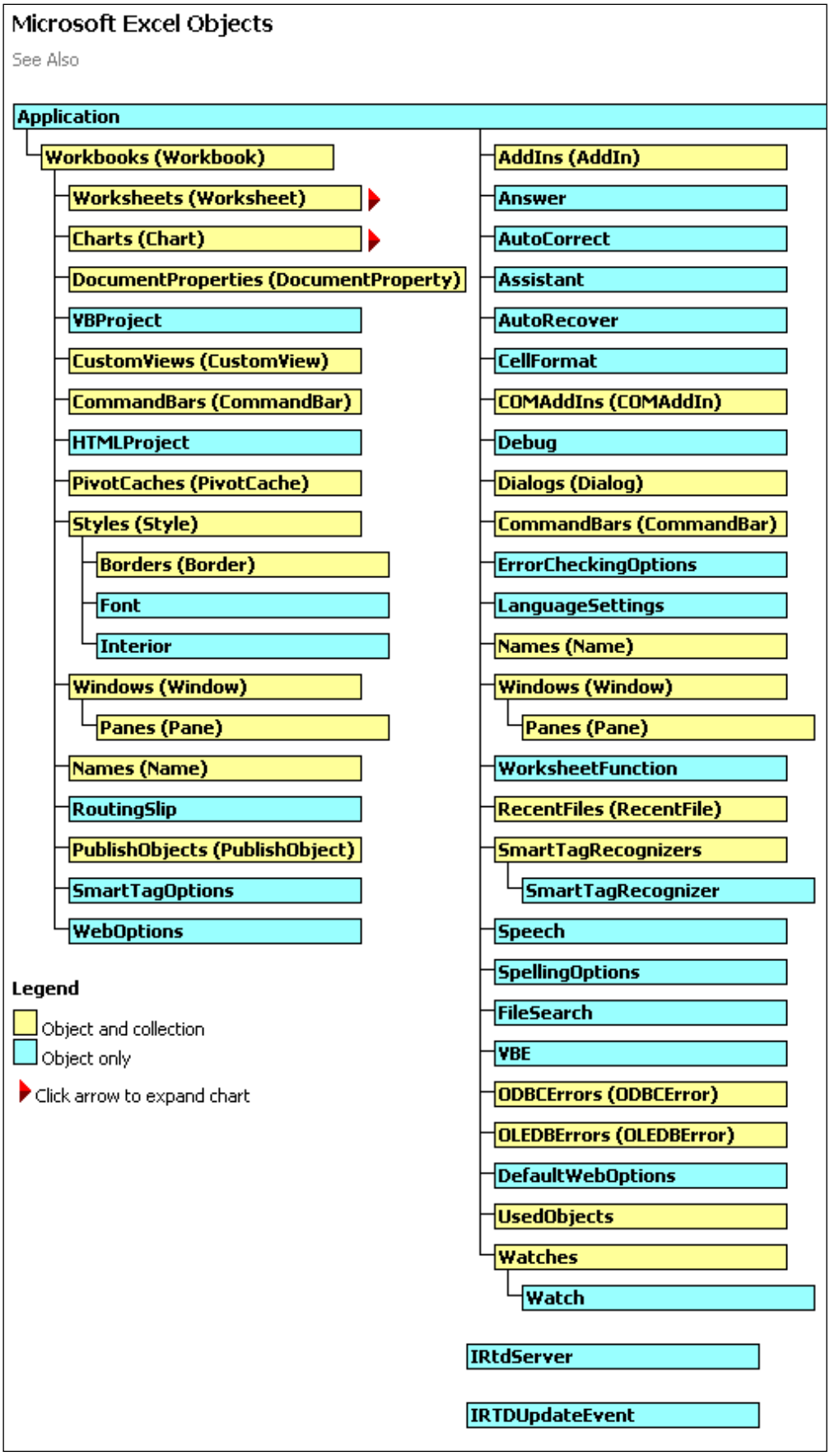
Some objects in the Excel Object model represent a **Collection** of objects. A collection is a set of objects of the same type.

The Workbooks collection in Excel represents a set of all open workbooks. An item in the collection can be referenced using an index number or its name.

To view the entire Excel Object model:

- Open the **Help** window
- Select the **Contents** tab
- Expand **Programming Information**
- Expand **Microsoft Excel Visual basic Reference**
- Select **Microsoft Excel Object Model**.

The following illustration shows a portion of the Excel object hierarchy. Most projects will only use a fraction of the available objects.


Microsoft Excel Objects
See Also
Application
Workbooks (Workbook)
Worksheets (Worksheet)
Charts (Chart)
DocumentProperties (DocumentProperty)
VBProject
CustomViews (CustomView)
CommandBars (CommandBar)
HTMLProject
PivotCaches (PivotCache)
Styles (Style)
Borders (Border)
Font
Interior
Windows (Window)
Panes (Pane)
Names (Name)
RoutingSlip
PublishObjects (PublishObject)
SmartTagOptions
WebOptions
AddIns (AddIn)
Answer
AutoCorrect
Assistant
AutoRecover
CellFormat
COMAddIns (COMAddIn)
Debug
Dialogs (Dialog)
CommandBars (CommandBar)
ErrorCheckingOptions
LanguageSettings
Names (Name)
Windows (Window)
Panes (Pane)
WorksheetFunction
RecentFiles (RecentFile)
SmartTagRecognizers
SmartTagRecognizer
Speech
SpellingOptions
FileSearch
VBE
ODBCErrors (ODBCError)
OLEDBErrors (OLEDBError)
DefaultWebOptions
UsedObjects
Watches
Watch
IRtdServer
IRTDUpdateEvent
Legend
Object and collection
Object only
Click arrow to expand chart

Defining Collections

A collection is a set of similar objects such as all open workbooks, all worksheets in a workbook or all charts in a workbook.

Many Excel collections have the following properties:

Application	Refers to the application that contains the collection
Count	An integer value representing the number of items in the collection.
Item	Refers to a specific member of the collection identified by name or position. Item is a method rather than a property
Parent	Refers to the object containing the collection

Some collections provide methods similar to the following:

Add	Allows you to add items to a collection
Delete	Allows you to remove an item from the collection by identifying it by name or position.

Referencing Objects in a Collection

A large part of programming is referencing the desired object, and then manipulating the object by changing its properties or using its methods. To reference an object you need to identify the collection in which it's contained.

The following syntax references an object in a collection by using its position. Since the **Item** property is the default property of a collection there is no need to include it in the syntax.

CollectionName(Object Index Number)

Workbooks.Item(1)

Workbooks(1)

Charts(IntCount)

Notes

The following syntax refers to an object by using the object name. Again the **Item** property is not necessary:

```
CollectionName(ObjectName)

Workbooks("Employees")

Worksheets("Purchases By Month")

Sheets("Total Sales")

Charts("Profits 2006")
```

Using the Object Browser

The Object Browser is used to examine the hierarchy and contents of the various classes and modules.

The Object Browser is often the best tool to use when you are searching for information about an object such as:

- Does an object have a certain property, method or event
- What arguments are required by a given method
- Where does an object fit in the hierarchy

To access the **Object Browser**:

In the **Visual Basic Editor**, do one of the following:

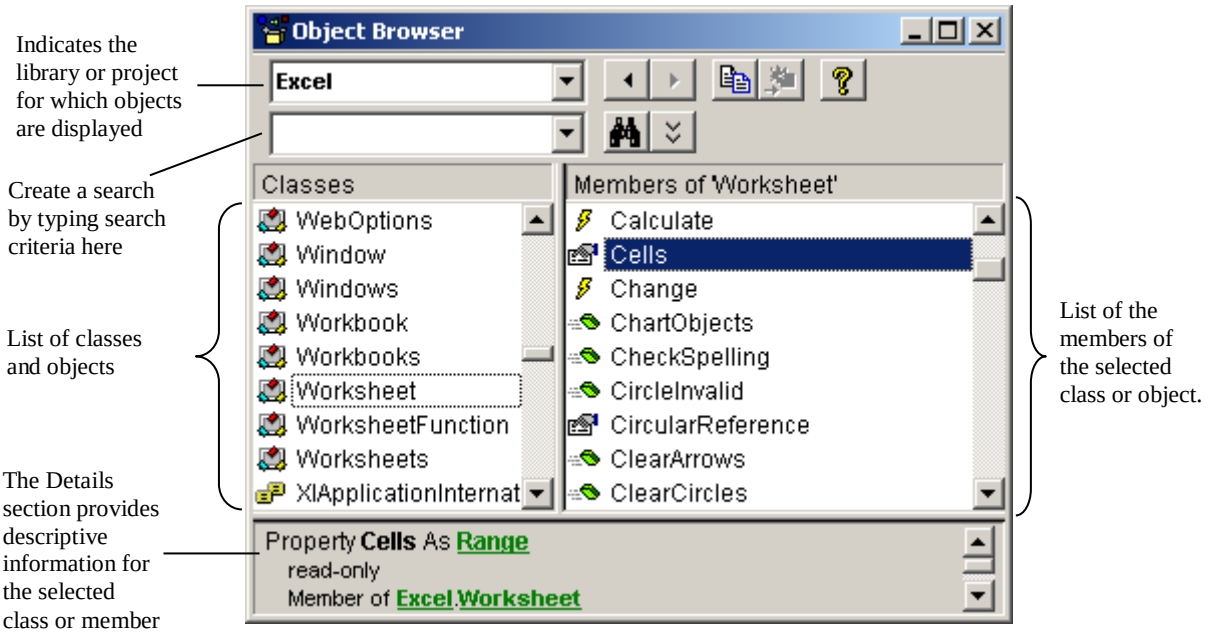
- Open the **View** menu
- Select **Object Browser** **OR**

- Press **F2** **OR**

- Click  the **Object Browser** icon.

 Notes

The **Object Browser** dialog box appears.



The following icons and terms are used in the **Object Browser**:

	Class	Indicates a Class (Eg Workbook, Worksheet, Range, Cells)
	Property	Is a value representing an attribute of a class (Eg. Name, Value)
	Method	Is a procedure that perform actions (Eg. Copy, Print Out, Delete)
	Event	Indicates an event which the class generates (Eg Click, Activate)
	Constant	Is a variable with a permanent value assigned to it (Eg vbYes)
	Enum	Is a set of constants

 Module Is a standard module

To search for an object in the **Object Bowser**:

- Type in the search criteria in the Search Text box
- Click

To close the Search pane:

- Click

Working with Properties

Most objects in Excel have an associated set of properties. During execution, code can read property values and in some cases, change them as well.

The syntax to read an object's property is as follows:

```
ObjectReference.PropertyName

ActiveWorkbook.Name
```

The syntax to change an object's property is as follows:

```
ObjectReference.PropertyName = expression

ActiveWorkbook.Name = "Quarterly Sales 2006"
```

The With Statement

The **With statement** can be used to work with several properties or methods belonging to a single object without having to type the object reference on each line.

The **With statement** helps optimize the code because too many "dots" in the code slows down execution.

The syntax for the **With statement** is as follows:

```
With ObjectName
<Statement>
End With


With ActiveWorkbook
        .PrintOut
        .Save
        .Close
End With
```

You can nest **With statements** if needed.

Make sure that the code does not jump out of the **With** block before the **End With** statement executes. This can lead to unexpected results.

Working With Methods

Many Excel objects provide public **Sub** and **Function** procedures that are callable from outside the object using references in your VB code. These procedures are called **methods**, a term that describes actions an object can perform.

Some methods require arguments that must be supplied when using the method.

The syntax to invoke an object method is as follows:

```
ObjectReference.method [argument]

Workbooks.Open "Sales 2006"

Range("A1:B20").Select

Selection.Clear
```

When calling procedures or methods that have arguments you have two choices of how to list the argument values to be sent.

Values can be passed by listing them in the same order as the argument list. This is known as a **Positional Argument**.

Alternatively you can pass values by naming each argument together with the value to pass. This is known as a **Named Argument**. When using this method it

is not necessary to match the argument order or insert commas as placeholders in the list of optional arguments

The syntax for using named arguments is as follows:

```
Argumentname:= value
```

The example shows the **PrintOut** method and its syntax:

```
Sub
PrintOut([From],[To],[Copies],[Preview],[ActivePrinter],[PrintToFile],[Collate],
[PrToFilename])
```

The statements below show both ways of passing values when calling the PrintOut method. The first passes by **Position**, the second by **Naming**:

```
Workbooks("Quarterly Sales 2006").PrintOut (1,2,2, , , ,True)

Workbooks("Quarterly Sales 2006").PrintOut From:=1, To:=2, Copies:=2,
Collate:=True
```

Event Procedures

An event procedure is a sub procedure created to run in response to an event associated with an object. For example run a procedure when a workbook opens.

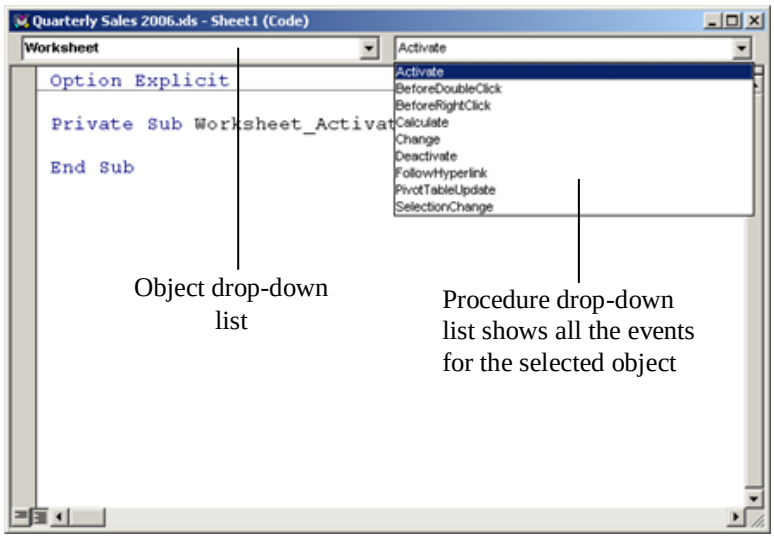
Event procedure names are created automatically. They consist of the object, followed by an underscore and the event name. These names cannot be changed. Event procedures are stored in the class module associated with the object for which they are written.

The syntax of the **Activate Event procedure** is as follows:

```
Private Sub Worksheet_Activate()
```

Creating An Event Procedure

To create an Event Procedure:



- Display the code window for the appropriate class module
- Select the Object from the Object drop-down list
- Select the event from the Procedure drop-down list
- Enter the desired code in the Event Procedure

Notes

Subject		Examples / Notes
Building Blocks	VBA Terminology	Objects (eg Worksheet) Property (eg Name) Method (eg Close) Procedure Container Objects (eg Workbook) Collection Objects (eg Worksheets) Type "Microsoft Excel Objects" in VBE Help to get the Excel object Hierarchy
	Visual Basic Editor (VBE)	The Projects window The Properties window The Code window **Alt-F11** – back and forth between VBE and Excel
	Changing object properties	Using the Properties window OR Using code: Object.property = newvalue Eg: ActiveSheet.Name = "New Sheet"
	Using methods	Syntax: object.method Eg: ActiveCell.Select ActiveSheet.Protect
	Coding to react to events	In the code window, select the object from the top left drop down menu and the Event from the top right drop down menu Eg: Private Sub Worksheet_Activate() End Sub
	Msgbox	Msgbox("This is my message") **vbCrLf** (Carriage return and Linefeed) Allows text displayed on a MsgBox to appear on multiple lines

	Adding Buttons	To toolbar (right click on toolbar and choose Customise) To worksheet (display Forms or Visual Basic toolbars)
	Object Browser	In VBE, select View / Object Browser to explore the 'library' of VBA code

Subject		**Examples / Notes**
Dealing with Data	Data Types	Byte, Boolean, Integer, Long, Single, Double, String, Date, Currency. .Also Variant and Object Type "Data Type Summary" in VBE Help to get the sizes and ranges for all data types
	Variables	Declaring variables: Implicitly by just using them Explicitly (Dim variable as type) Initialising (i.e. giving a variable a value): UserName = "My Name" Deptnumber = 234
	Scope	Procedure Level scope: ***Private Sub Worksheet_Activate()*** ***Dim MyVariable As String*** ***MyVariable = "Jonathan"*** ***End Sub*** Module Level scope: ***Option Explicit*** ***Dim MyVariable As String*** ***Private Sub Worksheet_Activate()*** ***MyVariable*** = "Jonathan" ***End Sub*** Public scope: ***Option Explicit*** ***Public MyVariable*** As String

		Private Sub Worksheet_Activate() ***MyVariable*** = "Jonathan" End Sub
	Modules	**Insert** menu to insert new module
	Procedures	**Add** menu to add new procedure, or type it: *Sub MyProceture* End Sub
	Calling Procedures	Call MyProcedure

Subject		**Examples / Notes**

Controlling Program Flow	Decision Structures	***If X = Y Then***
		
		Elseif X = Z Then
		
		Else
		
		End If

		Select Case username ***Case "Liz"*** ***Case "Jonathan"*** ***End Select***
	Loop Structures	**Fixed Iterations** ***For ThisCount = 1 to 10*** ***Next ThisCount***
		Variable Iterations For Each SheetVar In Worksheets (for Collections) Next Do While / Until X = Y Loop

Subject		Examples / Notes
More User Interaction	Creating a Custom User Form	In VBE, select **Insert** and **UserForm**
	Adding Controls	Use the control toolbox
	Naming Discipline	With Forms and Buttons and other controls… Change the name (use the Properties window) – eg: frmMainCommands txtUserName cmdCloseButton
	Adding code to forms/controls	Double-click on the object Refer to objects in your code, eg: txtUserName.Value = "Some Text"

	Responding to Events	In Code Window for forms, use top left drop down menu to select a control, and top right drop down menu shows events
		
		Eg:
		
		Private Sub cmdEnterName_Click()
		
		Range("E1").Value = txtUserName
		
		End Sub
		
		Or
		
		Private Sub txtUserName_AfterUpdate()
		
		If txtName.Value>11 And txtName.Value<15 Then
		
		Exit Sub
		
		Else
		MsgBox ("Not a valid Dept number")
		txtUserName.Value = ""
		
		End If
		End Sub

Subject		**Examples / Notes**
Debugging and Handling Errors	Types of Error	Compile Time Run Time Logical Type "Trappable Errors" in VBE Help to get the list of all trappable errors and their descriptions

	Debugging Tools	On the **Debug** menu: Breakpoint On the **View** menu: Locals Window (all variables) Watch Window (your choice of variables) Immediate Window
	On Error	On Error Goto Label Label: (must be left justified & with colon) On Error Resume Next

Subject		Examples / Notes
Extras	Line continuation	Workbooks.Open Filename:= _ "c:\MyDocuments\Excel VBA\Courses2005.xls"

	MsgBox buttons	Resp = MsgBox("Do you want to continue?", _
		vbYesNoCancel)
		
		If Resp = 6 then
		Msgbox("You hit 'Yes' didn't you?")
		Elseif Resp = 7 then
		Msgbox("You hit 'No' didn't you?")
		Elseif Resp = 2 then
		Msgbox("You hit 'Cancel' didn't you?")
		End If
		
		Type "VB Constants" in VBE Help to view the selection of VB Constants available

	Breaking Out	Press **Ctrl-Break keys** to interrupt code manually (or break out of an unending loop)
	Stop	Alternative to Breakpoint Sub Import() Stop End Sub
	Other useful code	Application.Dialogs(xlDialogOpen).Show ActiveWindow.ActivateNext Stop Screen Flickering Running VBA code may cause the screen to flicker. To switch off the screen until the program is run enter the following code line: Application.ScreenUpdating = False Screen comes on automatically on completion of the program. To Save a Workbook and close an Application ActiveWorkbook.Save ActiveWorkbook.SaveAs "Employees.xls" (Save Workbook with different name) Application.Quit (Quit the application. Code can be used in all Office applications