Microsoft Application Series

Excel VBA

Introduction-Intermediate

Best STL

- Courses never cancelled: guaranteed
- Last minute rescheduling
- 12 months access to Microsoft trainers
- 12+ months schedule
- UK wide delivery

www.microsofttraining.net

Your Best STL Learning Tools

Welcome to your Best STL training course.

As part of your training, we provide you with the following tools and resources to support and enhance your learning experience.

Thank you for choosing Best STL.

1 In-course handbook

To guide you through your training while you are on the course.

Contains unit objectives, exercises and space to write notes.

2 Reference material

Available online through your delegate account.

Comprehensive reference material with 100+ pages, containing step-by-step instructions.

3 12 months access to Microsoft trainers

Available through online support forum.

Need help? Our team of Microsoft qualified trainers are on hand to offer advice and support.

4 Delegate account

Your delegate account gives you access to:

- Reference material
- Course exercise files
- Advice & support forum
- Rewards programme
- Promotions & Newsletters

5 Trainer hints and tips

Hints and tips available online from our Microsoft qualified trainers for:

- All MS Office applications
- VBA
- MS Project
- MS Visio
  + more

6 Save with Promotions

Save on further training courses you book with Promotions.

- 30% off list price (time limited)
- £50 off list price (blue card discount)

Contents

Unit 1 The VBA Environment

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.

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 Copy/Paste operation, you should empty the clipboard with the following line of code to make sure that the computer memory doesn't overload:
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

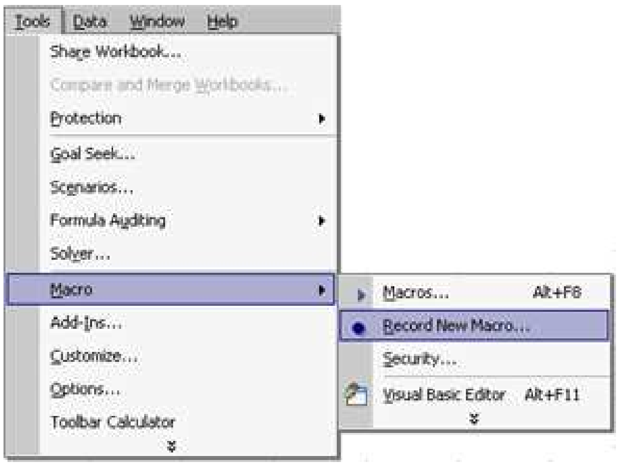
Recording and Running Macros

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.

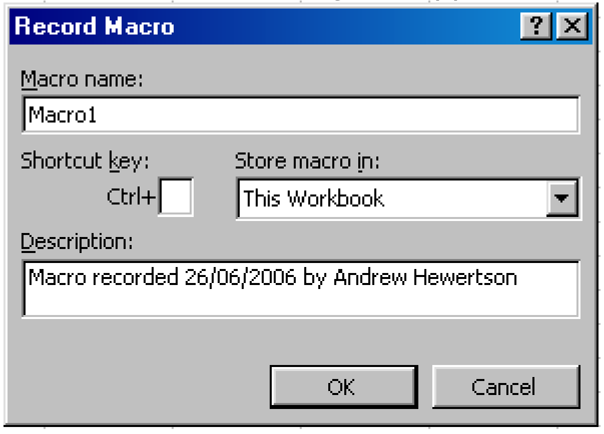
Macros can be stored on the current worksheet or made available globally by saving them in the Personal.xls workbook. This is a hidden workbook that automatically opens when you open Excel.

Recording a Macro



- Open the **Tools** menu
- Select **Macro**
- Choose **Record New Macro**.

The **Record Macro** dialog box appears.



Macro name: Macro1

Shortcut key: Ctrl+

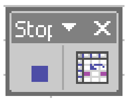
Store macro in: This Workbook

Description: Macro recorded 26/06/2006 by Andrew Hewertson

- 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 the recording



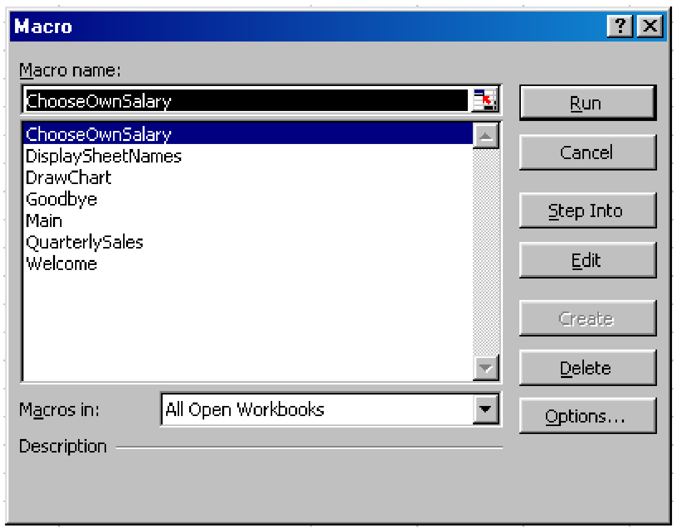
- Click the **Stop Recording** button. ■

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:

- Open the **Tools** menu
- Select **Macro**
- Choose **Macros**.

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 a Custom Toolbar

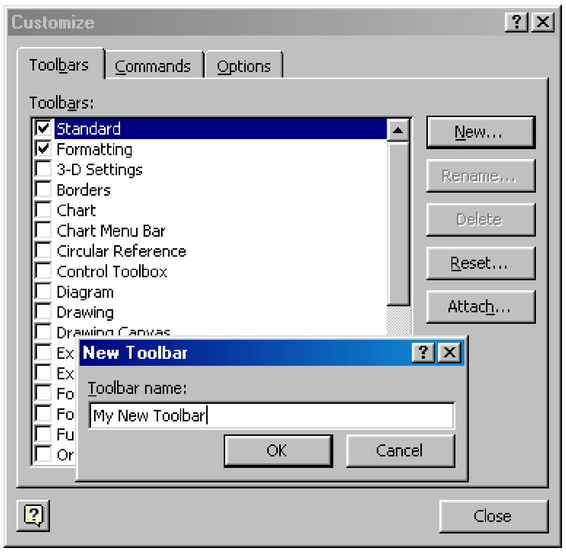
Macros and Sub Procedures can be executed from the Macro dialog box and from within other procedures. You can also execute procedures from toolbars and menus.

To assign a procedure to a custom toolbar:

- Open **Tools** menu **OR**
- Right–click in the toolbars area

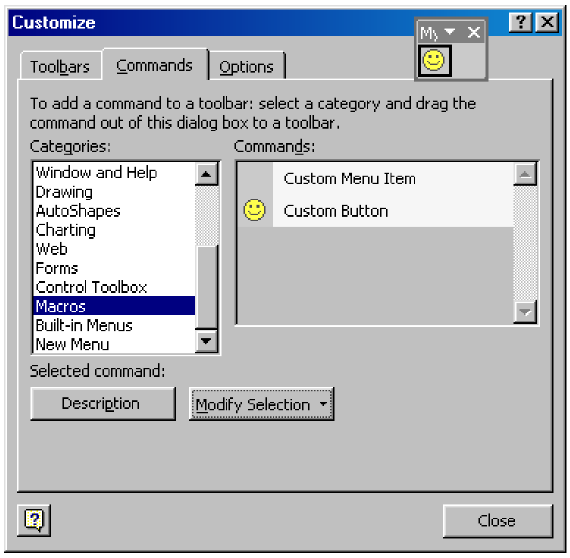
- Select **Customize**.

The **Toolbars** dialog box appears.



- Click the **Toolbars** tab
- Click **New**
- Name the new toolbar
- Click **OK**.

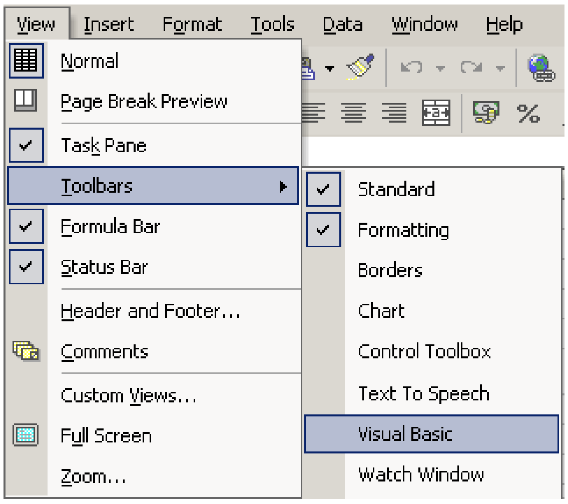
A new toolbar appears ready for buttons to be added. To do this:


To add a command to a toolbar: select a category and drag the command out of this dialog box to a toolbar.

- Click the **Commands** tab
- Select **Macros** from the **Categories** list.
- Drag the custom Button icon onto the new toolbar
- Click **Modify Selection**
- Click **Assign Macro**
- Select the required macro and click **OK**
- Click **Close**.

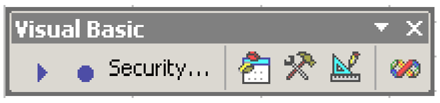
Using the Visual Basic Toolbar

As an alternative to this you can use the Visual Basic Toolbar to record and manage macros. To do this:



- Open the **View** Menu
- Select **Toolbars**
- Choose **Visual Basic**.

The **Visual basic** toolbar appears.



The most used buttons are described below:

▶	Run a Macro. A list of available macros appears
●	Record a Macro. The Record Macro toolbar appears
Security...	Opens the Security dialog box allowing the user to set security levels.
	Open the Visual Basic Editor.
	Open the Control Toolbox to access a variety of Form Controls
	Switch design mode On and Off

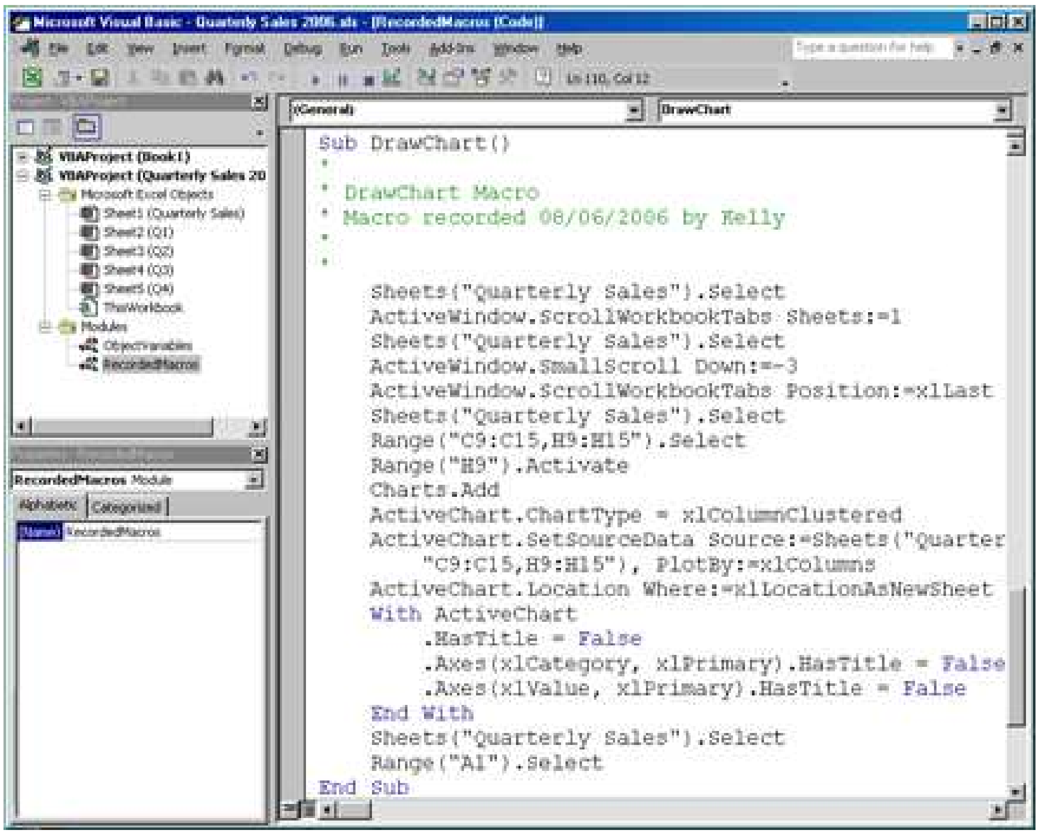
Editing Macros in Visual Basic Editor

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:

- Open the **Tools** menu
- Select **Macro**
- Choose **Macros**
- Select the desired macro from the **Macro Name** list
- Click **Run**.

The **Visual Basic Editor** appears.

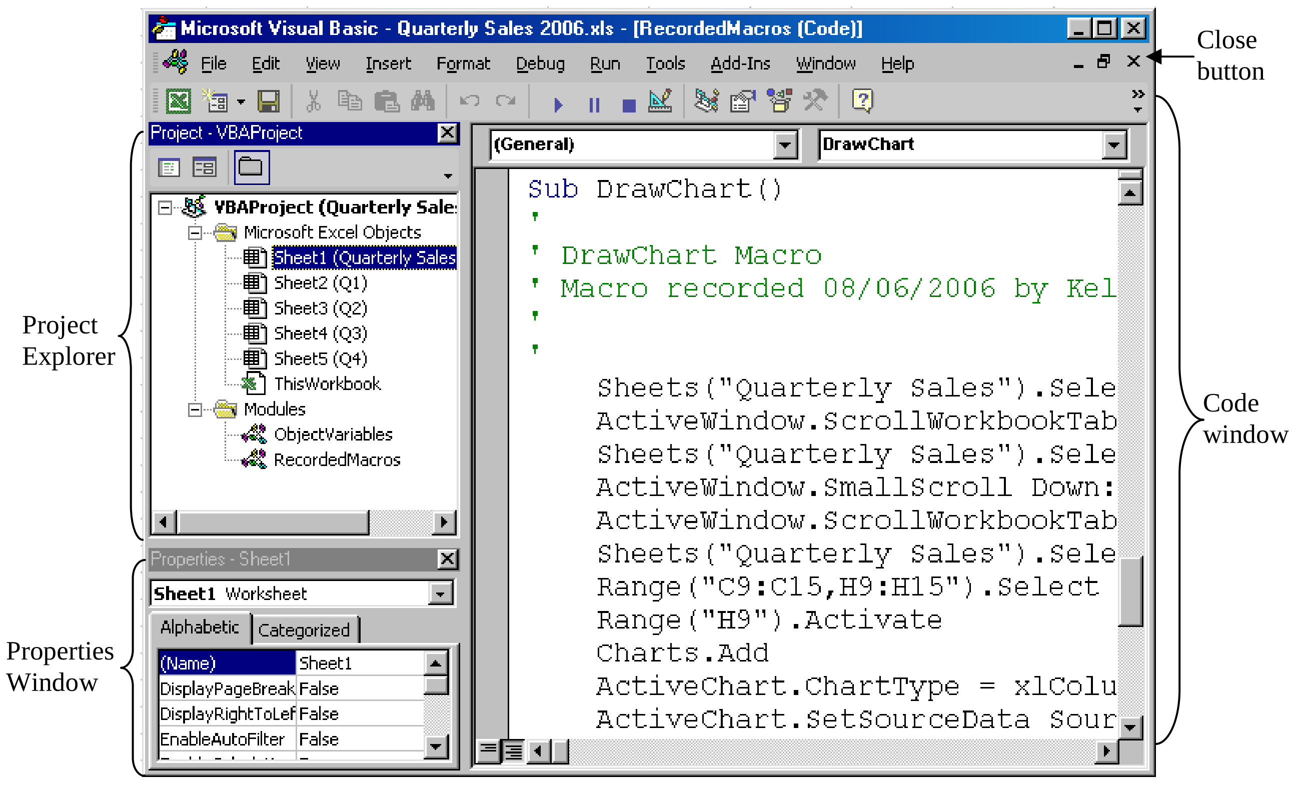


```
Sub DrawChart()
'
' DrawChart Macro
' Macro recorded 08/06/2006 by Kelly
'
'
    Sheets("Quarterly Sales").Select
    ActiveWindow.ScrollWorkbookTabs Sheets:=1
    Sheets("Quarterly Sales").Select
    ActiveWindow.SmallScroll Down:=-3
    ActiveWindow.ScrollWorkbookTabs Position:=xlLast
    Sheets("Quarterly Sales").Select
    Range("C9:C15,H9:H15").Select
    Range("H9").Activate
    Charts.Add
    ActiveChart.ChartType = xlColumnClustered
    ActiveChart.SetSourceData Source:=Sheets("Quarter
        "C9:C15,H9:H15"), PlotBy:=xlColumns
    ActiveChart.Location Where:=xlLocationAsNewSheet
    With ActiveChart
        .HasTitle = False
        .Axes(xlCategory, xlPrimary).HasTitle = False
        .Axes(xlValue, xlPrimary).HasTitle = False
    End With
    Sheets("Quarterly Sales").Select
    Range("A1").Select
End Sub
```

- 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



```
Sub DrawChart()
'
' DrawChart Macro
' Macro recorded 08/06/2006 by Kel
'
'
    Sheets("Quarterly Sales").Sele
    ActiveWindow.ScrollWorkbookTab
    Sheets("Quarterly Sales").Sele
    ActiveWindow.SmallScroll Down:
    ActiveWindow.ScrollWorkbookTab
    Sheets("Quarterly Sales").Sele
    Range("C9:C15,H9:H15").Select
    Range("H9").Activate
    Charts.Add
    ActiveChart.ChartType = xlColu
    ActiveChart.SetSourceData Sour
```

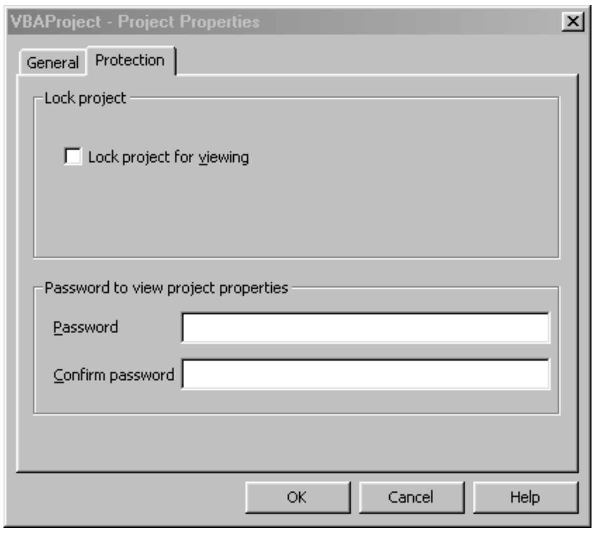
(Name)	Sheet1
DisplayPageBreak	False
DisplayRightToLef	False
EnableAutoFilter	False

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**

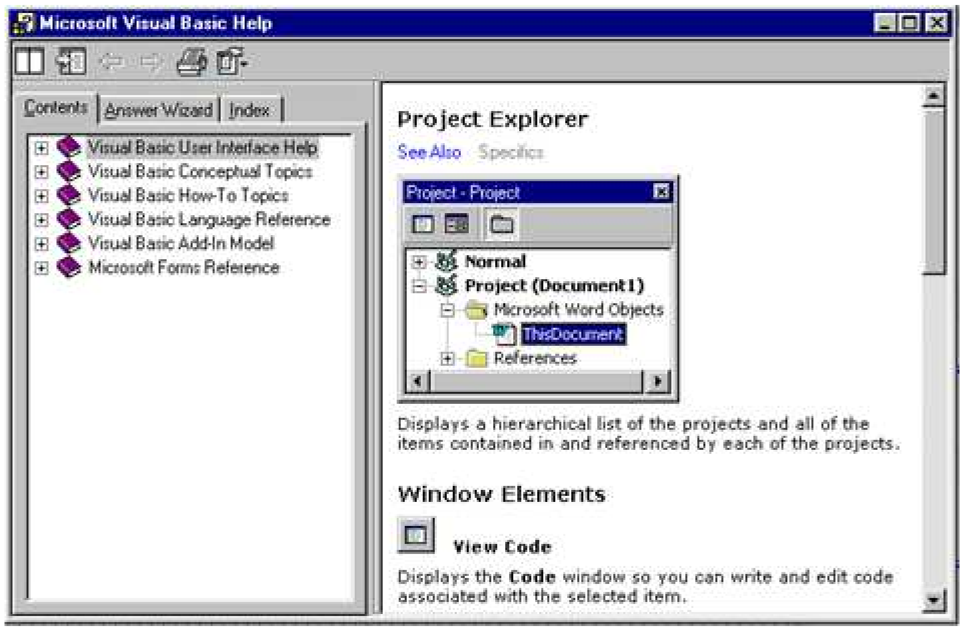
After doing this you must **Save and Close** the Workbook for the protection to take effect.

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:


Microsoft Visual Basic Help
Contents
Answer Wizard
Index
Visual Basic User Interface Help
Visual Basic Conceptual Topics
Visual Basic How-To Topics
Visual Basic Language Reference
Visual Basic Add-In Model
Microsoft Forms Reference
Project Explorer
See Also
Specifics
Project - Project
Normal
Project (Document1)
Microsoft Word Objects
ThisDocument
References
Window Elements
View Code

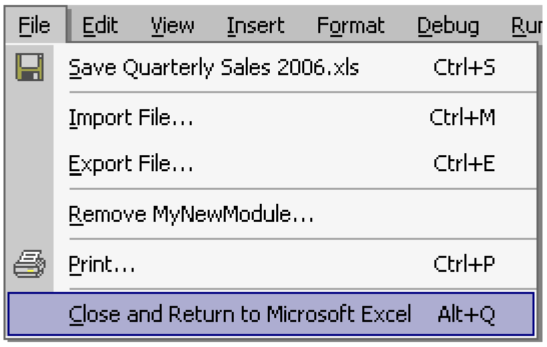
Displays a hierarchical list of the projects and all of the items contained in and referenced by each of the projects.

Displays the **Code** window so you can write and edit code associated with the selected item.

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 2 Developing with Procedures and Functions

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**.

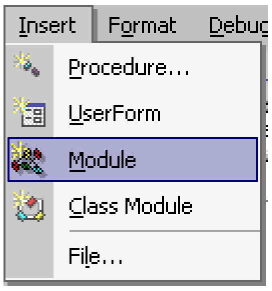
In this unit we will look at both Modules and Procedures.

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.

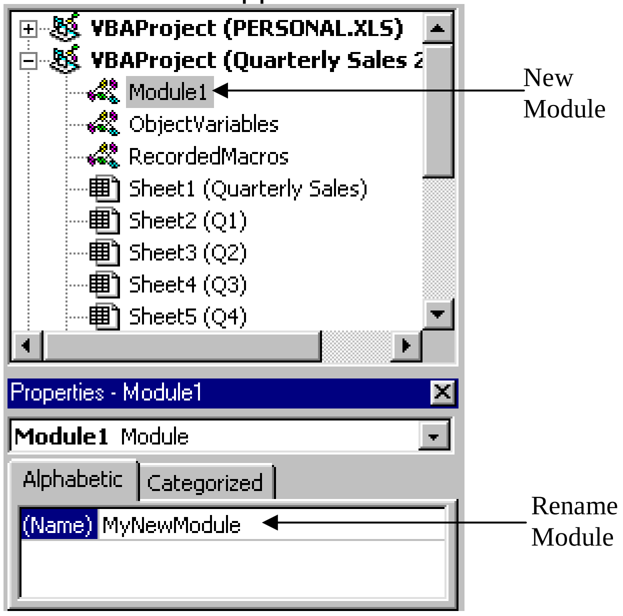
Within a project you can create as many standard modules as required. You should store related procedures together within the same module.

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:



(Name) MyNewModule

- 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 the following 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.

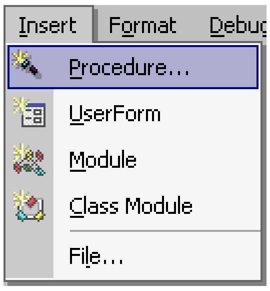
The **Sub…End Sub** structure can be typed directly into the code window or inserted using the **Add Procedure** dialog box.

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

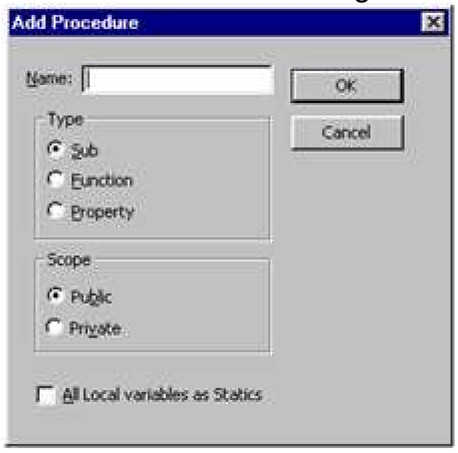
- Use **Add Procedure**.

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

Creating a Function Procedure

Function procedures are similar to built-in functions such as Sum(). They are sometimes called **user-defined** function.

A function returns a value to the procedure that calls it. The value the function generates is assigned to the name of the function.

Function procedures have the following syntax:

```
[Public/Private] Function FunctionName ([argument list]) [As <Type>]

[Statement block]

[FunctionName = <expression>]

End Function
```

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 **As** clause sets the data type of the function's return value.

To create a function procedure:

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

Display the **Add Procedure** dialog box (as in **Creating a Sub Procedure**):

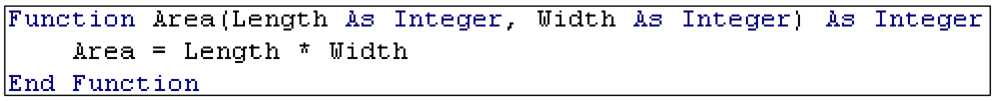
- Open the **Insert** menu
- Select **Procedure**.

Notes

The **Add Procedure** dialog box appears (as seen in **Creating a Sub Procedure**):

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

Below is an example of a basic function procedure:

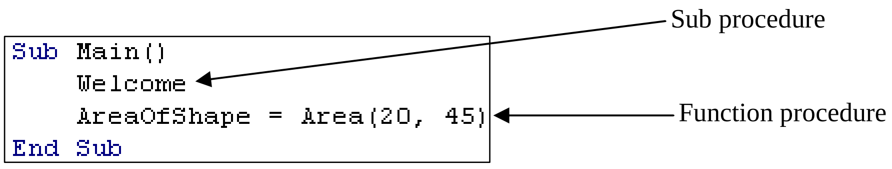


```
Function Area(Length As Integer, Width As Integer) As Integer
    Area = Length * Width
End Function
```

Calling Procedures

A sub procedure or function is called from the point in another procedure where you want the code to execute. The procedure being called must be accessible to the calling procedure. This means it must be in the same module or be declared public.

Below is an example of calls to Sub and Function procedures:

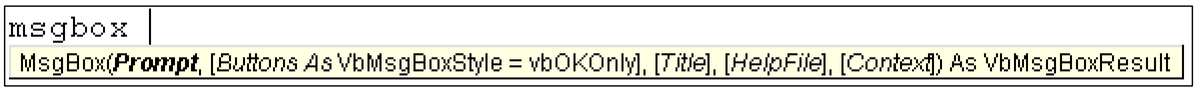


```
Sub Main()
    Welcome
    AreaOfShape = Area(20, 45)
End Sub
```

When passing multiple arguments (as in the function procedure above) always separate them with commas and pass them in the same order as they are listed in the syntax.

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:



```
msgbox
MsgBox(Prompt, [Buttons As VbMsgBoxStyle = vbOKOnly], [Title], [HelpFile], [Context]) As VbMsgBoxResult
```

Arguments in square brackets are optional.

Values passed to procedures are sometimes referred to as parameters.

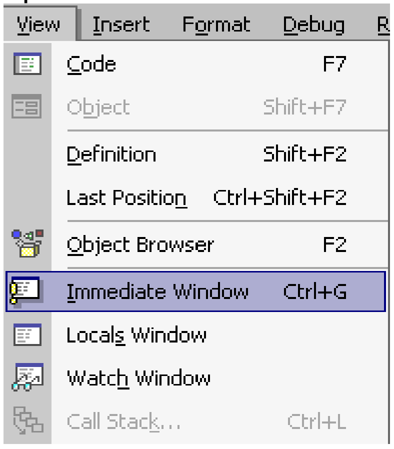
Notes

Using the Immediate Window to Call Procedures

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**.

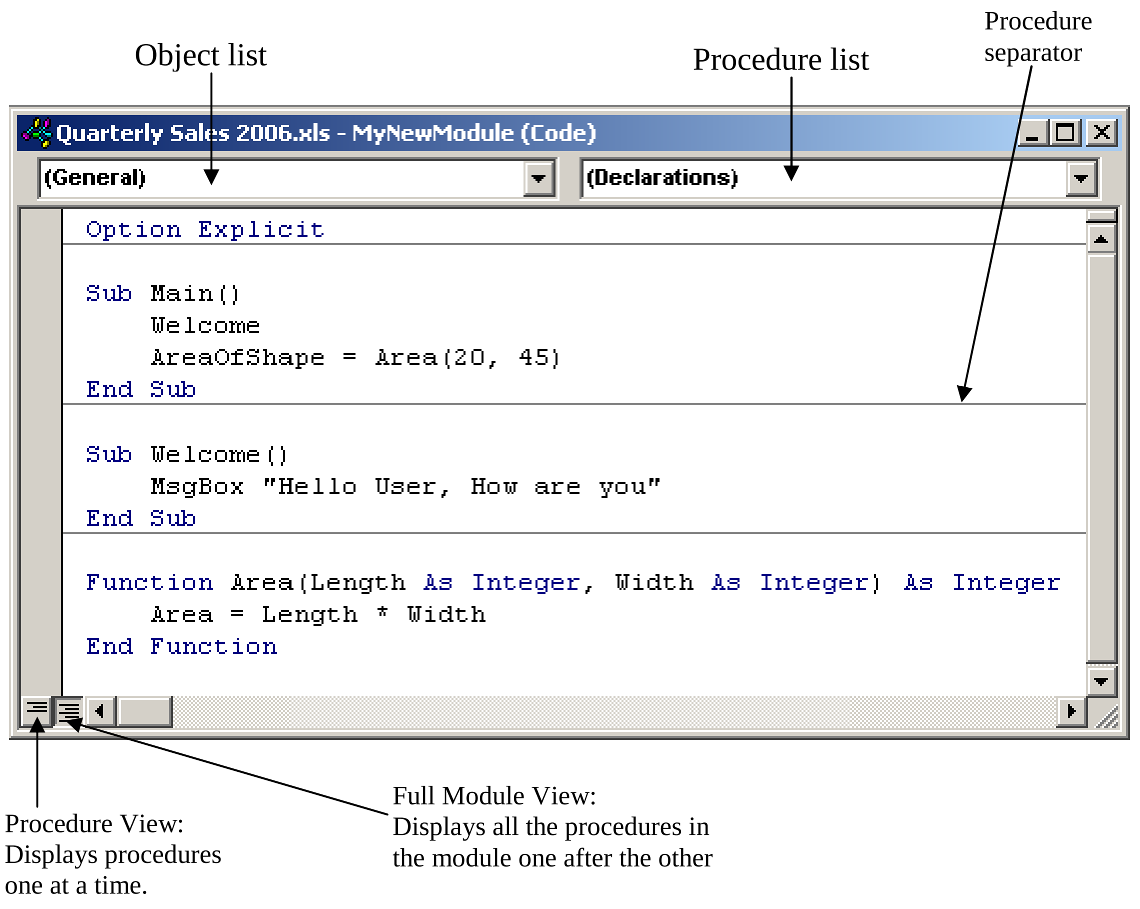
Within the code, especially in loops, use the **Debug.Print** statement to display values in the Immediate window while the code is executing. The Immediate window must be open for this.

Notes

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
Procedure list
Procedure separator
Quarterly Sales 2006.xls - MyNewModule (Code)
(General)
(Declarations)
Procedure View: Displays procedures one at a time.
Full Module View: Displays all the procedures in the module one after the other

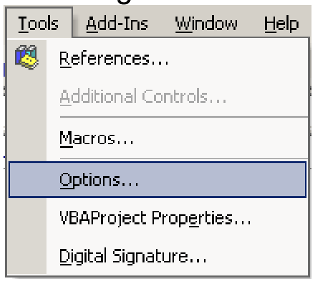
```
Option Explicit

Sub Main()
    Welcome
    AreaOfShape = Area(20, 45)
End Sub

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

Function Area(Length As Integer, Width As Integer) As Integer
    Area = Length * Width
End Function
```

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

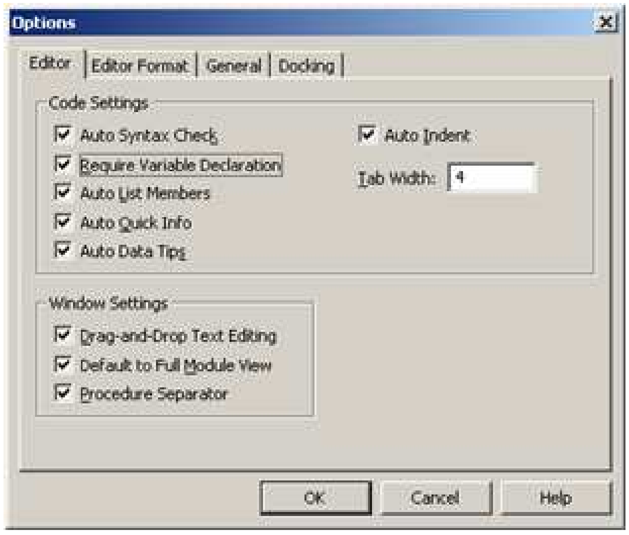
Setting Code Editor Options

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:



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

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 3 Understanding Objects

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.

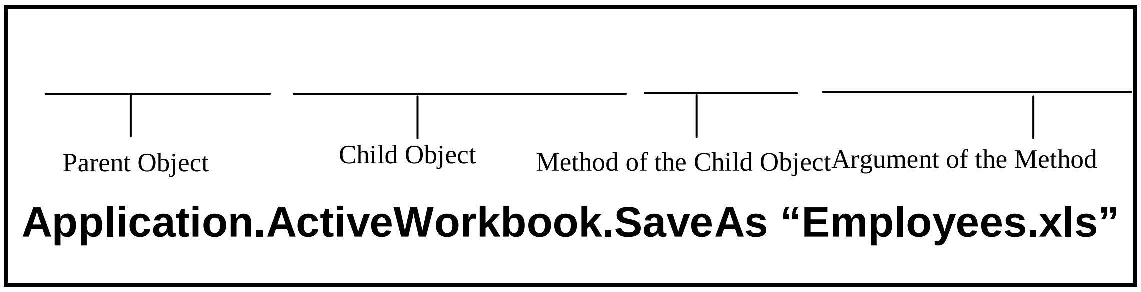
Notes

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:



Application.ActiveWorkbook.SaveAs “Employees.xls”

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”
```

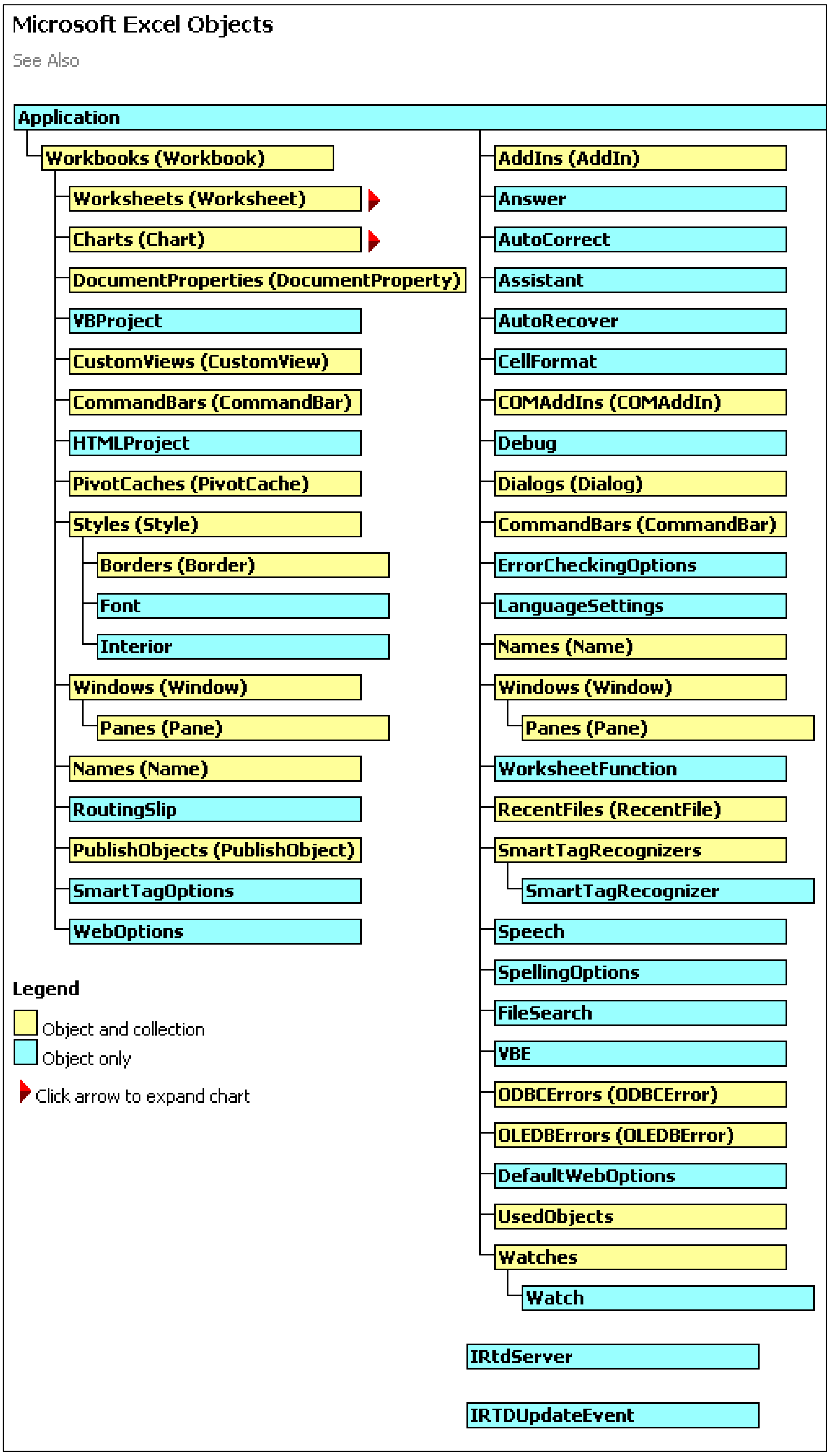
Some objects in the Excel Object model represent a **Collection** of objects. A collection is a set of objects of the same type.

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.



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:

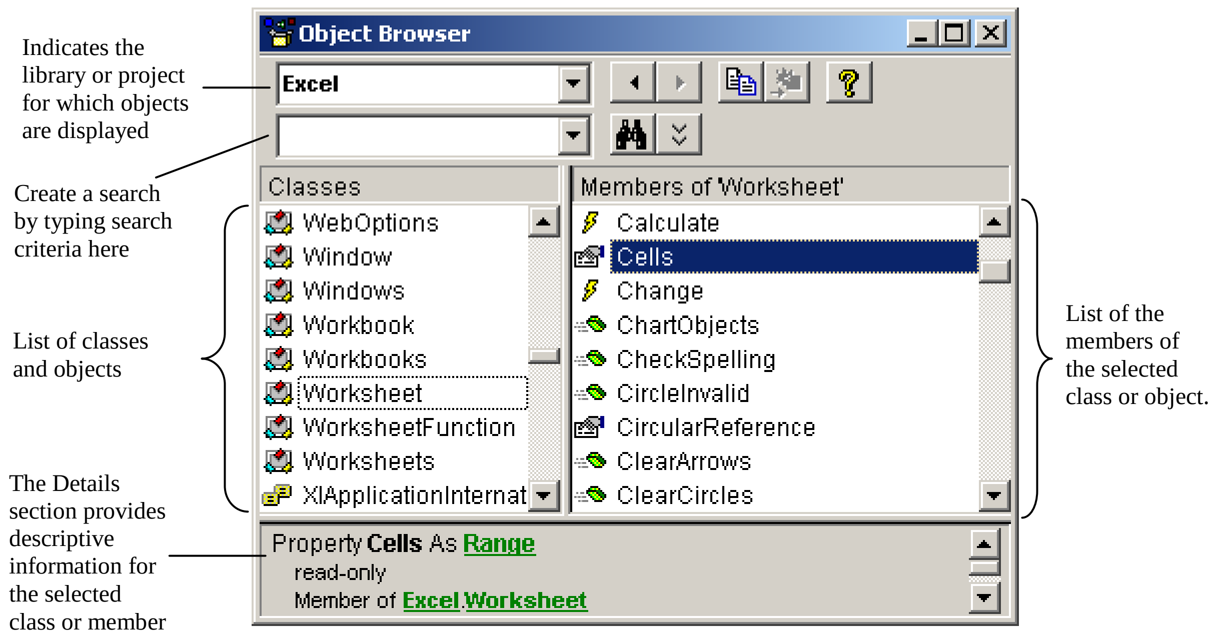
- Open the **View** menu
- Select **Object Browser** **OR**

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

- Click the **Object Browser** icon.

Notes

The **Object Browser** dialog box appears.


Indicates the library or project for which objects are displayed
Create a search by typing search criteria here
List of classes and objects
The Details section provides descriptive information for the selected class or member
List of the members of the selected class or object.
Object Browser
Excel

Classes
- WebOptions
- Window
- Windows
- Workbook
- Workbooks
- Worksheet
- WorksheetFunction
- Worksheets
- XlApplicationInternat

Members of 'Worksheet'
- Calculate
- Cells
- Change
- ChartObjects
- CheckSpelling
- CircleInvalid
- CircularReference
- ClearArrows
- ClearCircles

Property **Cells** As Range
read-only
Member of Excel.Worksheet

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.

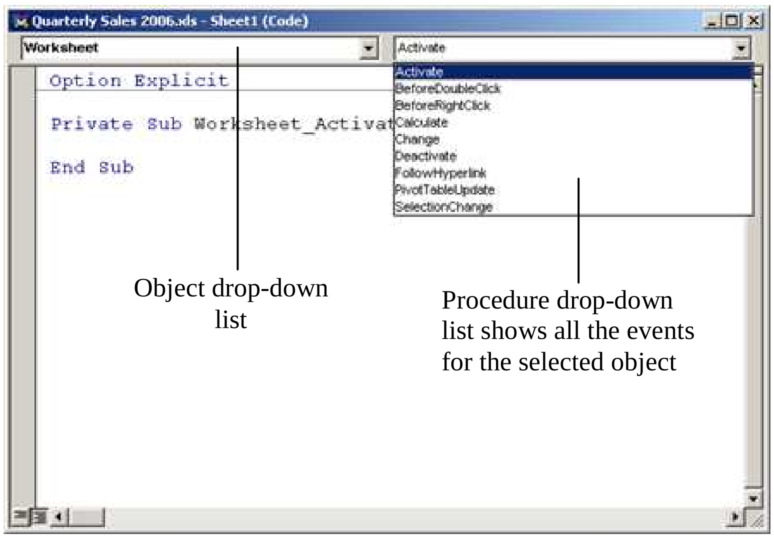
Event procedure names are created automatically. They consist of the object, followed by an underscore and the event name. These names cannot be changed. Event procedures are stored in the class module associated with the object for which they are written.

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

```
Private Sub Worksheet_Activate()
```

Creating An Event Procedure

To create an **Event Procedure**:



```
Option Explicit

Private Sub Worksheet_Activat

End Sub
```

- 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

Unit 4 Using Intrinsic Functions, Variables and Expressions

Defining Expressions And Statements

Any programming language relies on its expressions and the statements that put those expressions to use.

Expressions

An expression is a language element that, alone or in combination represents a value.
The different expression types typical of Visual basic are as follows:

String	Evaluates to a sequence of characters
Numeric	Evaluates to anything that can be interpreted as a number
Date	Evaluates to a date
Boolean	Evaluates to True or False
Object	Evaluates to an object reference

Expressions can be represented by any combination of the following language elements:

Literal	Is the actual value, explicitly stated.
Constant	Represents a value that cannot be changed during the execution of the program. (Eg. vbNo, vbCrLf)
Variable	Represents a value that can be changed during the execution of the program.
Function/Method /Property	Performs a procedure and represents the resulting value. This also includes self-defined functions
Operator	Allows the combination of expression elements +, - , * , / , >, <, =, <>

Statements

A statement is a complete unit of execution that results in an action, declaration or definition.

Statements are entered one per line and cannot span more than one line unless the line continuation character (_) is used.

Statements combine the language's key words with expressions to get things done.

Below are some examples of statements:

```
ActiveWorksheet.Name = “Quarterly Sales 2006”

Label = ActiveCell.Value

CurrentPrice = CurrentPrice * 1.1

ActiveSheet.PasteSpecial Paste:= Values _
    Operation:= None
```

Notes

How to Declare Variables

A variable is name used to represent a value. Variables are good at representing values likely to change during the procedure. The variable name identifies a unique location in memory where a value may be stored temporarily.

Variables are created by a **Declaration** statement. A variable declaration establishes its name, scope, data type and lifetime.

The syntax for a **Variable declaration** is as follows:

```
Dim/Public/Private/Static  VariableName  [As <type>]

Dim EmpName as String

Private StdCounter as Integer

Public TodaysDate As Date
```

Naming Variables

To declare a variable you give it a name. Visual Basic associates the name with a location in memory where the variable is stored.

Variable names have the following limitations:

- Must start with a letter
- Must NOT have spaces
- May include letters, numbers and underscore characters
- Must not exceed 255 characters in length
- Must not be a reserved word like **True, Range, Selection**

Assigning Values To Variables

An **Assignment** statement is used to set the value of a variable. The variable name is placed to the left of the equal sign, while the right side of the statement can be any expression that evaluates to the appropriate data type.

The syntax for a **Variable declaration** is as follows:

```
VariableName = expression

StdCounter = StdCounter + 1

SalesTotal = SalesTotal + ActiveCell.Value
```

Declaring Variables Explicitly

VBA does not require you to explicitly declare your variables. If you don't declare a variable using the **Dim** statement, VBA will automatically declare the variable for you the first time you access the variable. While this may seem like a nice feature, it has two major drawbacks:

- It doesn't ensure that you've spelled a variable name correctly
- It declares new variables as **Variants**, which are slow

Using Dim, Public, Private and Static declaration statements result in **Explicit** variable declarations.

You can force VBA to require explicit declaration be placing the statement **Option Explicit** at the very top of your code module, above any procedure declaration.

With this statement in place, a **Compiler Error - Variable Not Defined** message would appear when you attempt to run the code, and this makes it clear that you have a problem. This way you can fix the problem immediately.

Although, this forces you to declare variables, there are many advantages. If you have the wrong spelling for your variable, VBE will tell you. You are always sure that your variables are considered by VBE.

The best thing to do is tell the VBA Editor to include this statement in every new module. See **Setting Code Editor Options** on **Page 21**.

Important Note

When you declare more than one variable on a single line, each variable must be given its own type declaration. The declaration for one variable does not affect the type of any other variable. For example, the declaration:

```
Dim X, Y, Z As Single
```

is **NOT** the same as declaration

```
Dim X As Single, Y As Single, Z As Single
```

It **IS** the same as

```
Dim X As Variant, Y As Variant, Z As Single
```

For clarity, always declare each variable on a separate line of code, each with an explicit data type.

Determining Data Types

When declaring a variable you can specify a data type.

The choice of data type will impact the programs accuracy, efficiency, memory usage and its vulnerability to errors.

Data types determine the following:

- The structure and size of the memory storage unit that will hold the variable
- The kind and range of values the variable can contain. For example in the Integer data type you cannot store other characters or fractions
- The operations that can be performed with the variable such as add or subtract.

Important Info
If data type is omitted or the variable is not declared a generic type called **Variant** is used as default.

Excessive use of the **Variant** data type makes the application slow because Variants consume lots of memory and need greater value and type checks.

Notes

Numeric Data Types
Numeric data types provide memory appropriate for storing and working on numbers. You should select the smallest type that will hold the intended data so as to speed up execution and conserve memory.

Numeric operations are performed according to the order of operator precedence:

Operations inside parentheses **( )** are performed first. Excel evaluates the operators from left to right.

The following numeric operations are shown in order of precedence and can be used in with numeric data types.

Exponentiation (^)	Raises number to the power of the exponent
Negation (-)	Indicates a negative operand (as in –1)
Divide and Multiply (/ *)	Multiply and divide with floating point result
Modulus (Mod)	Divides two numbers and returns the remainder
Add and Subtract (+ -)	Adds and subtracts operands

String Data Types
The String data type is used to store one or more characters.

The following operands can be used with strings:

Concatenation (&)	Combines two string operands. If an operand is numeric it is first converted to a string-type Variant
Like *LikePattern*	Provides pattern matching strings

Notes

VBA supports the following data types:

Data type	**Storage size**	**Range**
Boolean	2 bytes	**True** or **False**
Byte	1 byte integer	0 to 255
Integer	2 bytes	-32,768 to 32,767
Long (long integer)	4 byte integer	-2,147,483,648 to 2,147,483,647
Single	4 byte floating point	Approximate range -3.40 x 10^{38} to 3.40 x 10^{38}
Double	8 byte floating point	-1.79769313486231E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Currency	8 bytes fixed point	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
String (variable-length)	10 bytes +	0 to approximately 2 billion characters
String (fixed-length)	Length of string	1 to approximately 65,400 characters
Variant (Numeric)	16 bytes	Any numeric value up to the range of a **Double**
Variant (String)	22 bytes +	Same range as for variable-length **String**
Decimal	12 byte (Only used within a Variant)	28 places to the right of the decimal; smallest non-zero number is +/- 0.0000000000000000000000000001
Date	8 byte floating point	1 January 100 to 31 December 9999
Object	4 bytes	An address reference to an **Object**

Important Info
For monetary values with up to 4 decimal places use the **Currency** data type.
Single and **Double** data types can be affected by small rounding errors.
A numeric variable of any type may be stored to a numeric variable of another type. The fractional part of a **Single** or **Double** will be rounded off when stored to an Integer type variable.

Programming with Variable Scope

The keywords used to declare variables, Dim, Static, Public or Private, define the scope of the variable. The scope of the variable determines which procedures and modules can reference the variable.

Procedure-Level Variables
These are probably the best known and widely used variables. They are declared (**Dim** or **Static**) inside the Procedure itself. Only the procedure that contains the variable declaration can use it. As soon as the Procedure finishes, the variable is destroyed.

Module-Level Variables
These are variables that are declared (**Dim** or **Private**) outside the Procedure itself in the **Declarations** section of a module.

By default, variables declared with the **Dim** statement in the **Declarations** section are scoped as private. However, by preceding the variable with the **Private** keyword, the scope is obvious in your code.

All variables declared at this level are available to all **Procedures** within the Module. Its value is retained unless the variable is referenced outside its scope, the Workbook closes or the **End Statement** is used.

Public Variables
These variables are declared at the top of any standard **Public** module. **Public** variables are available to all procedures in all modules in a project

The **Public** keyword can only be used in the **Declarations** section

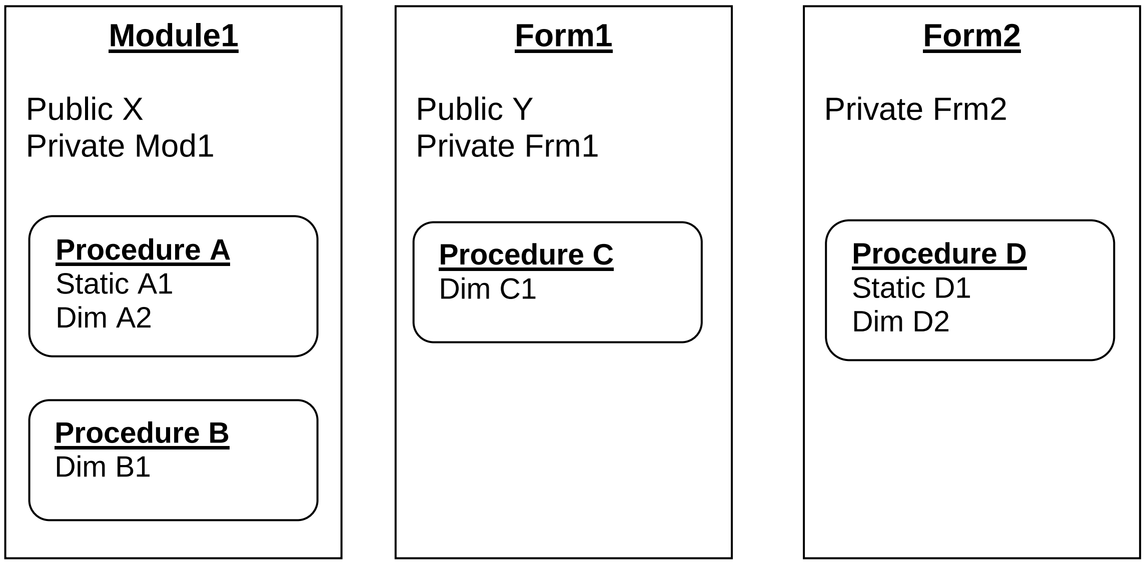
Public procedures, variables, and constants defined in other than standard or class modules, such as **Form modules** or **Report modules**, are not available to referencing projects, because these modules are private to the project in which they reside.

Variables are processed in the following order:

1. **Local (Dim)**
2. **Module-Level (Private, Dim)**
3. **Public (Public)**

Notes

The diagram below illustrates how variables can be accessed across procedures, modules and forms, based on the scope of each variable:



Each of the procedures can only see the variables as follows:

Procedure A can see: A1, A2, Mod1, X, Y

Procedure B can see: B1, Mod1, X, Y

Procedure C can see: C1, Frm1, X, Y

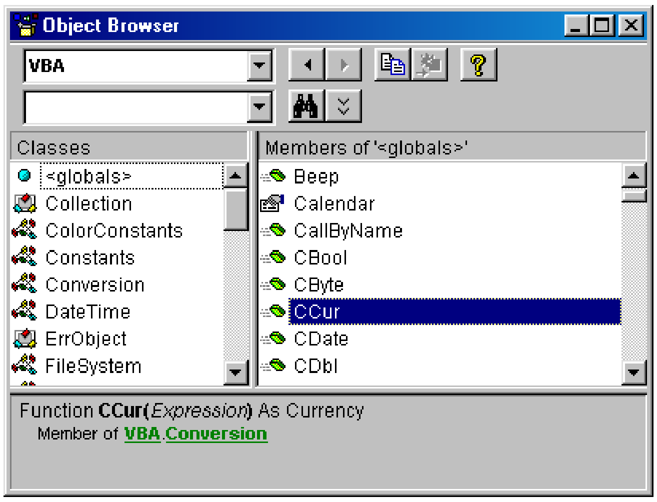
Procedure D can see: D1, D2, Frm2, X, Y

Notes

Harnessing Intrinsic Functions

An intrinsic function is similar to a function procedure in that it performs a specific task or calculation and returns a value. There are many intrinsic functions that can be used to manipulate text strings, or dates, covert data or perform calculations.

Intrinsic functions appear as methods in the **Object Browser**. To view and use them:



Function CCur(*Expression*) As Currency
Member of VBA.Conversion

- Select **VBA** from the **Project/Library** drop down list.
- Select **<globals>** in the **Classes** pane.
- Select the required intrinsic function.

For further help on a particular function, display the **Visual Basic Help** window. On the **Contents** tab:

- Expand Visual Basic Language Reference
- Expand Functions
- Expand the appropriate alphabet range
- Select the desired function.

Defining Constants and Using Intrinsic Constants

A constant is a variable that receives an initial data value that doesn't change during the programs execution. They are useful in situations where a value that is hard to remember appears over and over. The use of constants can make code more readable.

The value of the constant is also set in the declaration statement. Constants are Private by default, unless the Public keyword is used.

The syntax of a **Constant declaration** is as follows:

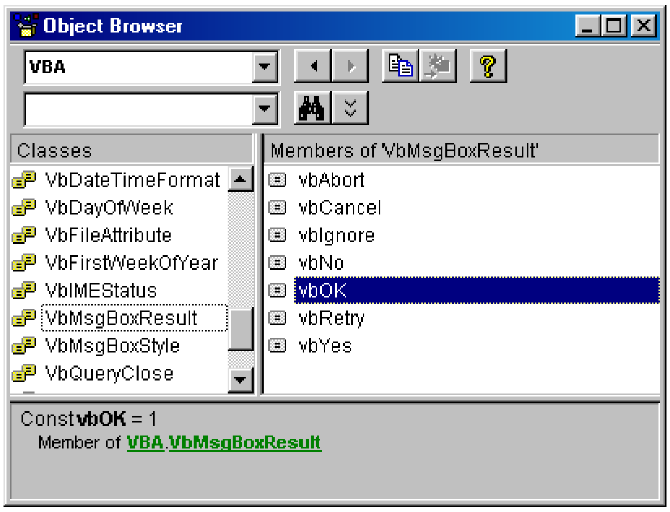
[Public] [Private] Const ConstantName [<As type>] = <*ConstantExpr*>

```
Const conPassMark As String = "C"

Public Const conMaxSpeed As Integer = 30
```

Using Intrinsic Constants

VBA has many built-in constants that can be used in expressions. VBA constants begin with the letters ***vb*** while constants belonging to the Excel object library begin with ***xl***.

To access Intrinsic constants in the **Object Browser** follow the steps below:



VbDateTimeFormat
VbDayOfWeek
VbFileAttribute
VbFirstWeekOfYear
VbIMEStatus
VbMsgBoxResult
VbMsgBoxStyle
VbQueryClose

vbAbort
vbCancel
vbIgnore
vbNo
vbOK
vbRetry
vbYes

Const **vbOK** = 1
Member of VBA.VbMsgBoxResult

- Select **VBA** from the **Project/Library** drop down list.
- Select the object you want to use in the **Classes** pane e.g. **vbMsgBoxResult**.
- Select the required intrinsic function e.g. **vbOK**

Some useful Visual Basic constants are listed below:

Constant	Equivalent to:	Same as pressing:
vbCr	Carriage Return	Enter
vbTab	Tab character	Tab
vbLf	Soft return and linefeed	Shift + Enter
vbCrLf	Combination of carriage return and linefeed	
vbBack	Backspace character	Backspace
vbNullString	Zero length string	“”

For a full list of Visual Basic Constants, search **Help** for **VB Constants** while in the Visual Basic Editor.

Adding Message Boxes

The **MsgBox Function** can be used to display messages on the screen and prompt for a user's response.

The **MsgBox Function** can display a variety of buttons, icons and custom title bar text.

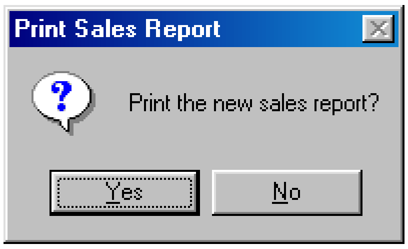
The **MsgBox Function** can be used to return a constant value that represents the button clicked by user.

The **MsgBox Function** syntax is as follows:

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

```
MyResponse = MsgBox (“Print the new sales report?”, 36, _
“Print Sales Report”)

MyResponse = MsgBox (“Print the new sales report?”, _
vbYesNo + vbQuestion, “Print Sales Report”)
```

Both **MsgBox Functions** above produce a message box with 2 buttons, a text message, an icon and a title as shown below:

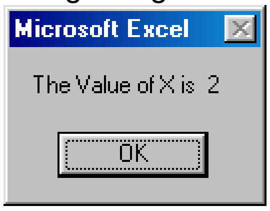

Print Sales Report

Print the new sales report?

Another example of using the message box is to return a value:

```
Sub Example()
Dim X As Integer
X = 2
MsgBox "The Value of X is " & Str(X)
End Sub
```

The **Msgbox** message must be a string (text), hence the **Str() function** is required to convert an integer to a string which is concatenated with the first string using the & operator.


Microsoft Excel

The Value of X is 2

The **MsgBox Function** has the components described below:

prompt	Required. It is a string expression displayed as the message in the dialog box. The maximum length of *prompt* is approximately 1024 characters. If *prompt* consists of more than one line, you can separate the lines by concatenating and using carriage return code **vbCrLf.**
buttons	Optional. Numeric expression that defines the set of command buttons to display, the icon style to use, the identity of the default button, and the modality of the message box. Can be specified by entering a vbConstant, the actual numeric value of the constant or the sum of constants. 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** "Microsoft Excel" is the default title
helpfile	Optional. String expression that identifies the Help file to use for the input box. If **helpfile** is provided, **context** must also be provided.
context	Optional. Numeric expression that identifies the appropriate topic in the Help file related to the message box

The values and constants for creating buttons are shown below:

Constant	**Value**	**Description**
vbOKOnly	0	**OK** button only (default)
vbOKCancel	1	**OK** and **Cancel** buttons
vbAbortRetryIgnore	2	**Abort**, **Retry**, and **Ignore** buttons
vbYesNoCancel	3	**Yes**, **No**, and **Cancel** buttons
vbYesNo	4	**Yes** and **No** buttons
vbRetryCancel	5	**Retry** and **Cancel** buttons

The values for creating icons are shown below:

Constant	**Value**	**Description**
vbCritical	16	Display the Stop icon
vbQuestion	32	Display the Question icon
vbExclamation	48	Display the Exclamation icon
vbInformation	64	Display the Information icon

The values for setting the default command button are shown below:

Constant	**Value**	**Description**
vbDefaultButton1	0	First button set as default (default)
vbDefaultButton2	256	Second button set as default
vbDefaultButton3	512	Third button set as default
vbDefaultButton4	768	Fourth button set as default

The values for controlling the modality of the message box are shown below:

Constant	**Value**	**Description**
vbApplicationModal	0	Application modal message box (default)
vbSystemModal	4096	System modal message box
vbMsgBoxHelpButton	16384	Adds Help button to the message box
VbMsgBoxSetForeground	65536	Specifies the message box window as the foreground window

To display the **OK** and **Cancel** buttons with the **Stop icon** and the second button (Cancel) set as default, the argument would be:

273 (1 + 16 +256).

It is easier to sum the constants than writing the actual values themselves:

vbOKCancel, vbCritical, vbDefaultButton2.

When adding numbers or combining constants, for the button argument, **select only one value**, from each of the listed groups.

Notes

Return Values

The **MsgBox Function** returns the value of the button that is clicked. Again this can be referenced by the number or the corresponding constant.

The Return values of the corresponding constants are as follows:

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

The return value is of no interest when the MsgBox only displays the OK button.

In this case just call the **MsgBox Function** with the syntax used to call a sub procedure as shown below:

```
MsgBox ( “You must enter a number”, vbOKOnly, “Attention”)

Or

MsgBox “You must enter a number”
```

Notes

Using Input Boxes

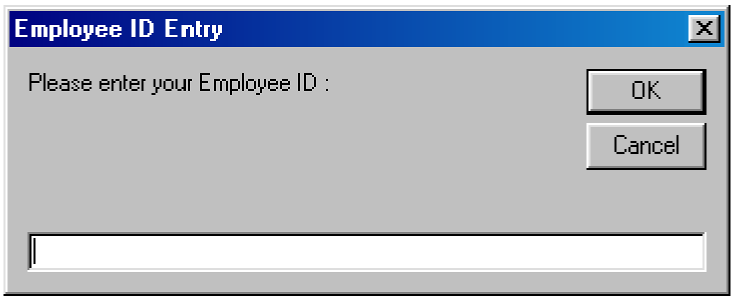
The **InputBox Function** prompts the user for a piece of information and returns it as a string.

The syntax of a **InputBox Function** is as follows:

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

```
strEmpID = InputBox (“Please enter your Employee ID :”, “Employee ID Entry”)
```

In the example the return value of the function is being stored in a variable called strEmpID.



If **OK** is clicked, the function returns the contents of the text box or a zero-length string, if nothing is entered.

If the user clicks **Cancel**, it returns a zero-length string, which may cause an error in the procedure if a value is required.

Notes

The **ImputBox Function** has the components described below:

prompt	Required. String expression displayed in the dialog box. The maximum length of prompt is approximately 1024 characters.
title	Optional. String expression displayed in the title bar of the dialog box. If you omit title "Microsoft Excel is the default title.
default	Optional. String expression displayed in the text box as the default response. If you omit default, the text box is displayed empty.
xpos	Optional. Numeric expression that specifies, in **twips**, the horizontal distance of the left edge of the dialog box from the left edge of the screen. If **xpos** is omitted **ypos** must also be omitted.
ypos	Optional. Numeric expression that specifies, in **twips**, the vertical distance of the upper edge of the dialog box from the top of the screen.
helpfile	Optional. String expression that identifies the Help file to use for the Input box. If **helpfile** is provided, **context** must also be provided.
context	Optional. Numeric expression that identifies the appropriate topic in the Help file related to the Input box

A twip is equal to 1/20th of a point.

How to Declare and Use Object Variables

You can also use variables to reference objects in order to work with their properties, methods and events. Any Excel object such as Worksheet, Chart, Range or Cell can be represented and accessed using a variable name.

The **Object Variable** syntax is as follows:

```
Dim/Public/Private/Static  VariableName  [As <Objecttype>]

Dim SalesRange As Range

Public wsSheet As Worksheet
```

Assigning values to object variables requires the keyword **Set**:

```
Set  VariableName = Objectname

Set SalesRange = ActiveSheet.Range("A1:F12")

Set wsSheet = Worksheet ("Sales 2006")
```

Once an object is assigned to an object variable, the object can be referenced by its variable name. Object variables are used to avoid typing lengthy object references.

Unit 5 Debugging the Code

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.

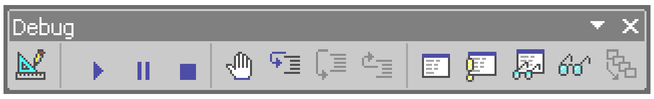
Minimizing Errors
Here are a few suggestions to help you minimize or make it easier to find errors in your code:

- 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

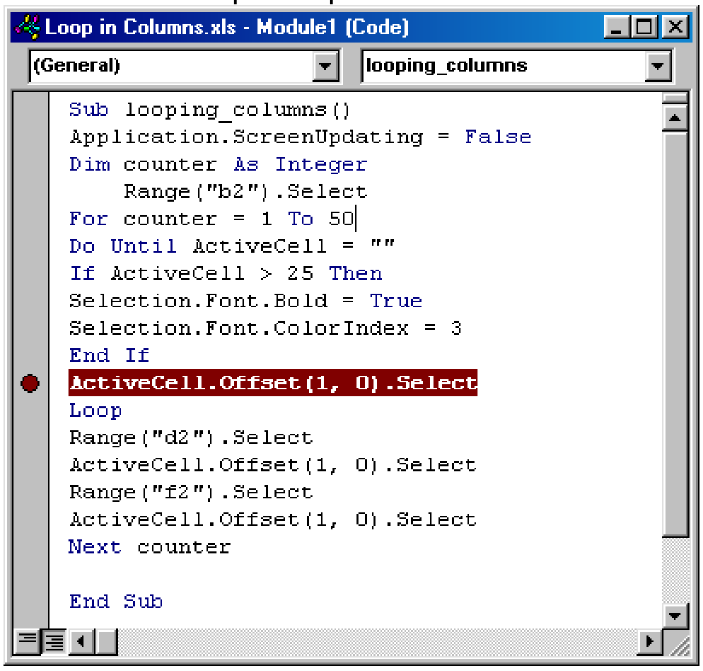
Another quick way of finding out the value of variables and expressions is the **Auto Data Tip** which displays the value of the expression where the mouse is pointing.

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:



```
Sub looping_columns()
Application.ScreenUpdating = False
Dim counter As Integer
    Range("b2").Select
For counter = 1 To 50
Do Until ActiveCell = ""
If ActiveCell > 25 Then
Selection.Font.Bold = True
Selection.Font.ColorIndex = 3
End If
ActiveCell.Offset(1, 0).Select
Loop
Range("d2").Select
ActiveCell.Offset(1, 0).Select
Range("f2").Select
ActiveCell.Offset(1, 0).Select
Next counter

End Sub
```

- 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

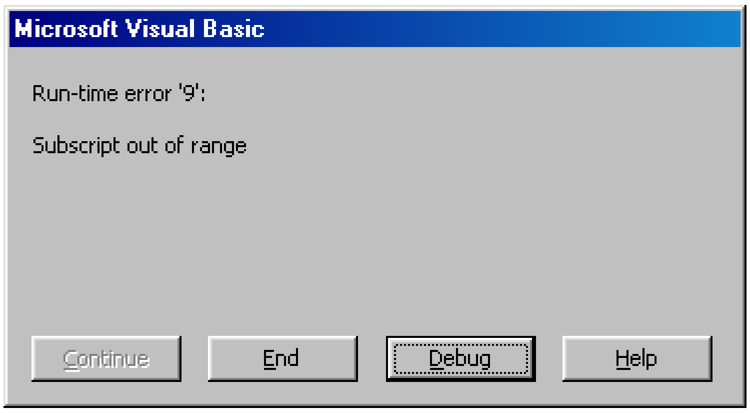
The step tools allow you to step one line at a time through the code to see exactly which statements in your procedure are being executed.

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.



Microsoft Visual Basic

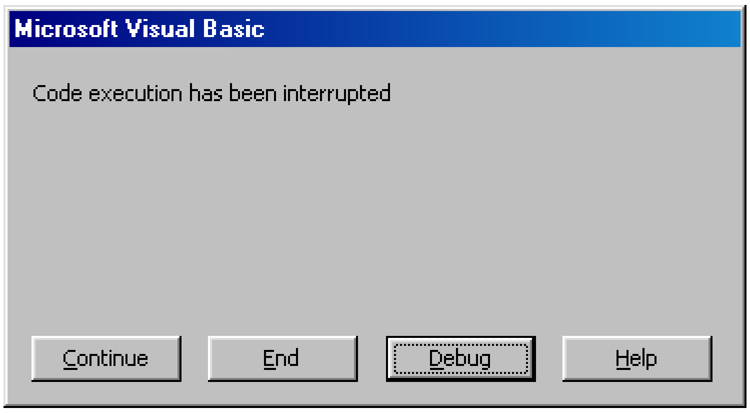
Run-time error '9':

Subscript out of range

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**.

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



Microsoft Visual Basic

Code execution has been interrupted

Notes

Unit 6 Handling Errors

Handling errors is another aspect of writing good code. VBA allows you to enter instructions into a procedure that directs the program in case of an error.

Successfully debugging code is more of an art than a science. The best results come from writing understandable and maintainable code and using the available debugging tools. When it comes to successful debugging, there is no substitute for patience, diligence, and a willingness to test relentlessly, using all the tools at your disposal.

Writing good error handlers is a matter of anticipating problems or conditions that are beyond your immediate control and that will prevent your code from executing correctly at run time. Writing a good error handler should be an integral part of the planning and design of a good procedure. It requires a thorough understanding of how the procedure works and how the procedure fits into the overall application. And, writing good procedures is an essential part of building solid Microsoft Office solutions.

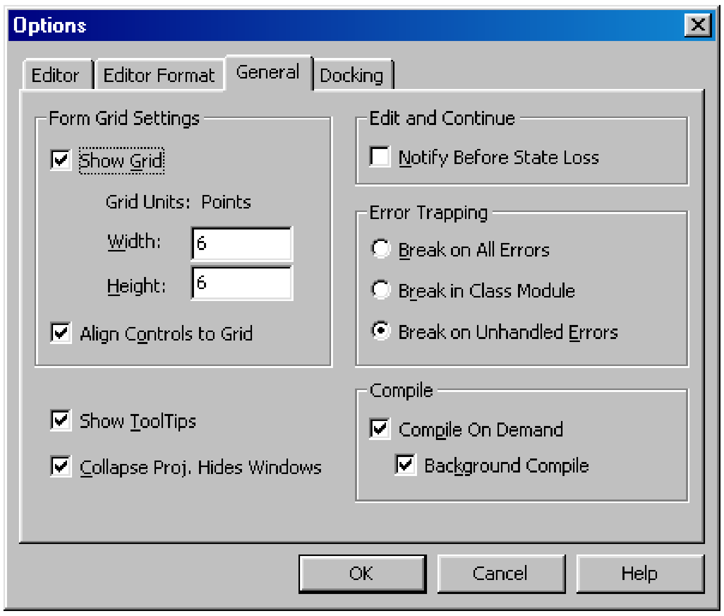
Good error handling should keep the program from terminating when an error occurs.

Defining VBA's Error Trapping Options

The error trapping mechanism can be turned on, off or otherwise modified while developing a project.

To set the **Error Handling** options:

- Open the **Tools** menu
- Select **Options**



Form Grid Settings
[✓] Show Grid
Grid Units: Points
Width: 6
Height: 6
[✓] Align Controls to Grid

[✓] Show ToolTips
[✓] Collapse Proj. Hides Windows

Edit and Continue
[ ] Notify Before State Loss

Error Trapping
( ) Break on All Errors
( ) Break in Class Module
(●) Break on Unhandled Errors

Compile
[✓] Compile On Demand
[✓] Background Compile

The **Options** dialog box appears.

The **Error Trapping options** are explained below:

Break on All Errors	Causes program to enter Break mode and display an error message regardless of whether you have written code to handle the error. This option turns the error handling mechanism off and should be used for debugging only
Break in Class Module	Causes program to enter Break mode and display an error message when an unhandled error occurs within a procedure of a class module such as a User Form. If the Debug button is clicked in the error message window, the Code window will display the line of code that generated the error highlighted. Should be used for debugging only.
Break on Unhandleded Errors	Causes the program to enter Break mode and display a message when an unhandled error occurs. **This is the setting that should be selected before distributing your application.**

For a list of trappable errors in Excel search **Help** for **Trappable Errors Constants** while in the Visual Basic Editor.

A list of the error numbers and their descriptions appears.

Capturing Errors with the On Error Statement

In a procedure, you enable an error trap with an **On Error** statement. If an error is generated after this statement in encountered, the Error handler takes over and passes control to what the **On Error** statement specifies.

The Error-Handling syntax is as follows:

On Error <*branch instruction*>

```
On Error GoTo ErrorHandler

On Error Resume Next
```

Once a On Error statement has trapped an error, the error needs to be handled. Below are the 3 basic styles that VBA uses for handling errors:

Write an Error handler	This uses the **On Error GoTo** statement. It would include statements to handle one or more errors for the procedure.
Ignore the Error	If the error is inconsequential, use the **On Error Resume Next** statement to both trap and handle the error. The program continues on the next line of code.
Use in-line error handling	Use the **On Error Resume Next** statement to trap the error. Then enter code to check for errors immediately following any statements expected to generate errors.

On Error GoTo 0
This statement disables the error-handling for the procedure at least until another **On Error statement** is encountered. This is an alternative to changing the **Error Trapping** settings to **Break on All Errors** as it only affects the procedure it is in. Once the issue is resolved remove the statement from the procedure.

Error trapping is defined on a procedure-by-procedure basis. VBA does not allow you to specify a global error trap.

Determining the Err Object

When an error occurs, VBA uses the **Err** object to store information about that error. The **Err** object can only contain information about one error at a time

The properties of the **Err** object contain information such as the Error **Number**, **Description**, and **Source**.

The **Err** object's **Raise** method is used to generate errors, and its **Clear** method is used to remove any existing error information.

Using the Raise methods to force an error can help in error testing routines.

The following statement generates a “Division By Zero” error message:

```
Err.Raise 11
```

Coding an Error-Handling Routine

The On Error Go To statement is used to branch to a block of code within the same procedure which handles errors. This block is known as the error-handling routine and is identified by a line label.

The routine is always stored at the bottom of the procedure, preceded by an **Exit** statement that prevents the routine from being executed unless an error has occurred.

Common line labels used to identify an Error-handling routine are “ErrorHandler” and “EH”. You can use one of these or create a personal one to handle all your error-handling routines.

Line labels only have to be unique within the procedure.

The benefit of using this style is that all the error-handling logic is at the bottom rather than being mixed up with the main logic of the procedure making the procedure easier to read and understand.

The example below illustrates a error-handling routine for a sub procedure:

```
Sub RunFormula()

On Error GoTo ErrorHandler

Dim A As Double
Dim B As Double

A = InputBox("Type in the value for A")
B = InputBox("Type in the value for B")

MsgBox A / B

Exit Sub

ErrorHandler:

If Err.Number = 11 Then
    B = InputBox(Err.Description & " is not allowed.  Enter a non-zero number.")
    Resume
Else
    MsgBox "Unexpected Error.  Type " & Err.Description
End If

End Sub
```

When an execution has passed into an error routine the following list shows how to specify which code to be used next:

Resume	Execution continues on the same line within the procedure that caused the error.
Resume Next	Execution continues on the line within the procedure that follows the line that caused the error.
Resume <*Line Label*>	Execution continues on the line identified by the line label. This usually points to another routine within the procedure that performs a "clean-up" be releasing variables and deleting temporary files.
End Sub / End Function	Used to exit the procedure normally by reaching the End Sub or end Function command
Exit Sub / Exit Function	Immediately exits the procedure in which it appears. Execution continues with the statement following the statement that called the procedure.

Notes

Using Inline Error Handling

Using this method you place the code to handle errors directly into the body of the procedure, rather than placing it at the end of the routine.

To do this, place the **On Error Resume Next** statement into the procedure. The error handling code is then placed immediately after the line where the code is expected to cause error. This method may be simpler to use in very long procedures where two or more errors are anticipated.

```
Sub ProcFileOpen()

On Error Resume Next

Open "C:\My Documents\Sales2006.xls" For Input As #1
Select Case Err
   Case 53
      MsgBox "File not found:  C:\My Documents\Sales2006.xls"
    Case 55
      MsgBox "File in use:  C:\My Documents\Sales2006.xls"
    Case Else
      MsgBox "Err Number: " & Err.Number & vbLf & _
      "Error Descriptoion: " & Err.Description
End Select

Err.Clear

End Sub
```

Notes

Unit 7 Managing Program Execution

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
```

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.

Unit 8 Harnessing Forms And Controls

Defining UserForms

Dialog boxes are used in applications to interface with the user. VBA allows you to create custom dialog boxes that can display information or retrieve information from the user as required. These are known as **UserForms** or just **Forms**.

A UserForm serves as a container for control objects, such as labels, command buttons, combo boxes, etc. These controls depend on the kind of functionality you want in the form. When a new UserForm is added to the project, the UserForm window appears with a blank form, together with a toolbox containing the available controls. Controls are added by dragging icons from the toolbox to the UserForm. The new control appears on the form with 8 handles that can be used to resize the control. The grid dots on the form help align the controls on the form.

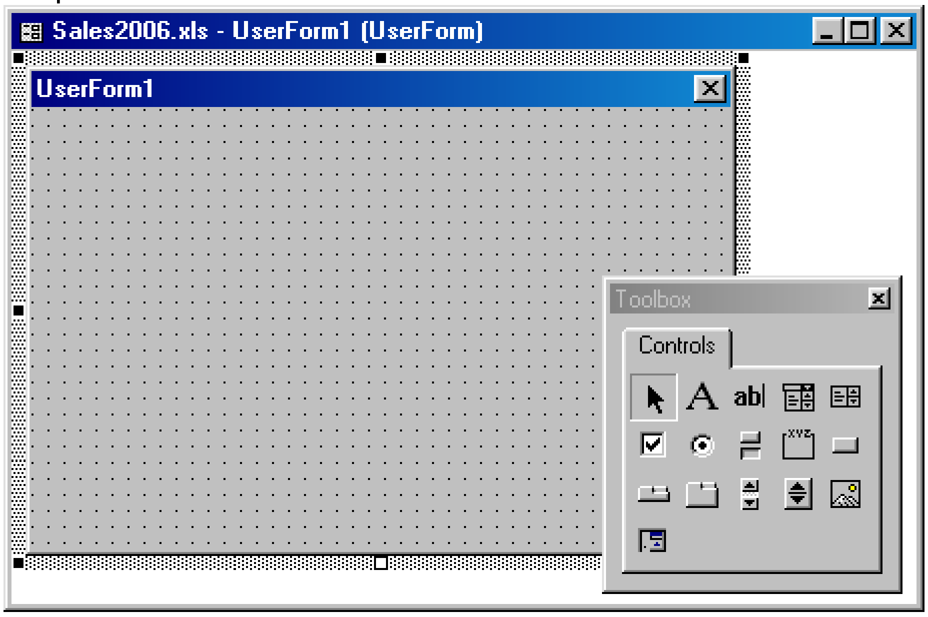
To add a **UserForm** to a project:
In the **Visual Basic Editor**, select the desired Project name in the Project Explorer.

To insert a **UserForm** do one of the following:

- Open the **Insert** menu
- Select **UserForm**. **OR**

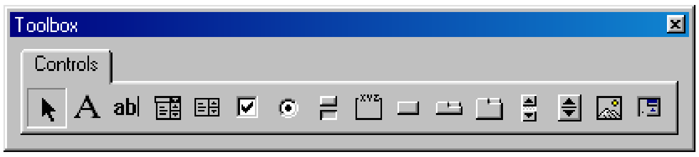
- Right-click the project name
- Select **Insert** and choose **UserForm**.

A blank user form appears together with the toolbox.
Press **F7** to display the code window of the selected form and **F4** to display the Properties window.



Utilising the Toolbox

While working on a form the toolbox is displayed but becomes hidden when another window in the Visual Basic Editor is selected. Controls are added to forms to build a desired interface and add functionality.



The default set of controls, from left to right, on the above toolbox are described below:

Select Objects	Makes the mouse behave as a pointer for selecting a control on a form.
Label	Creates a box for static text
Text Box	Creates a box for text input or display.
Combo Box	Creates the combination of a drop-down list and textbox. The user can select an option or type the choice.
List Box	Creates a scrollable list of choices
Check Box	Creates a logical check box
Option Button	Creates an option button that allows exclusive choice from a set of options.
Toggle Button	Creates a toggle button that when selected indicates a **Yes**, **True** or **On** status.
Frame	Creates a visual or functional border.
Command Button	Creates a standard command button.
Tab Strip	Creates a collection of tabs that can be used to display different sets of similar information.
MultiPage	Creates a collection of pages. Unlike the Tab Strip each page can have a unique layout.
Scroll Bar	Creates a tool that returns a value of for a different control according to the position of the scroll box on the scroll bar
Spin Button	Creates a tool that increments numbers.
Image	Creates an area to display a graphic image.
RefEdit	Displays the address of a range of cells selected on one or more worksheets.

Double-click a toolbox icon and it remains selected allowing multiple controls to be drawn.

Using UserForm Properties, Events And Methods

Every **UserForm** has its own set of properties, events and methods. Properties can be set in both the Properties window and through code in the Code window.

Properties

All forms share the same basic set of properties. Initially every form is the same. As you change the form visually, in the UserForm window, you are also changing its properties. For example if you resize a form window, you change the Height and Width properties.

The following list describes the more commonly used properties of a UserForm:

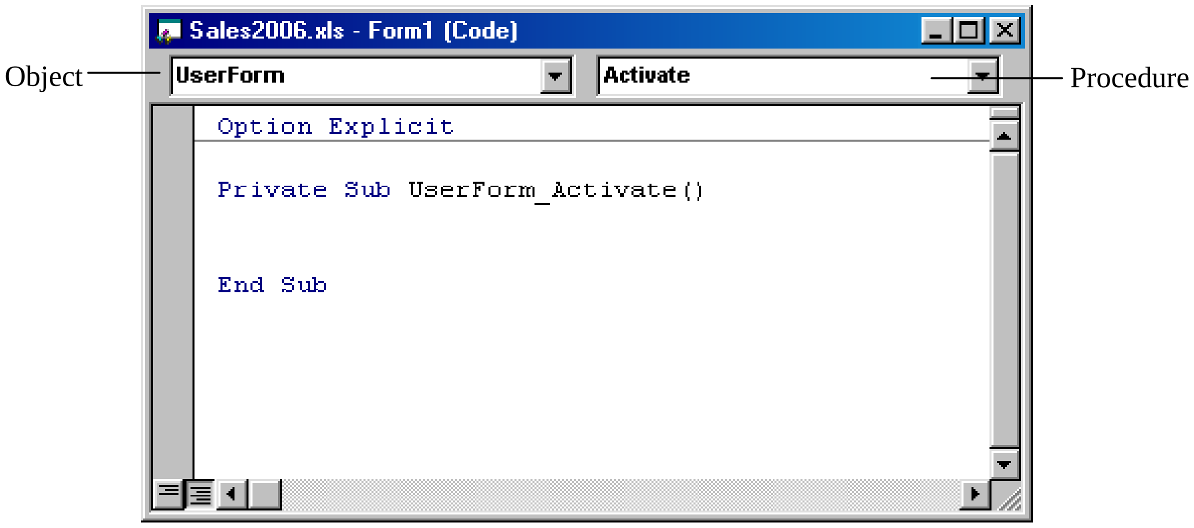
Property	**Description**
BackColor	Sets the background colour of a form.
BorderStyle	Sets the border style for the form.
Caption	Sets the form’s title in the title bar.
Enabled	Determines whether the form can respond to user-generated events.
Height	Sets the height of the form.
HelpCOntextID	Associates a context-sensitive Help topic with a form.
MousePointer	Sets the shape of the mouse pointer when the mouse is positioned over the form.
Picture	Specifies picture to display in the form.
StartUpPosition	Sets where on the screen the form will be displayed.
Width	Sets the width of the form.

Notes

Events

All **UserForms** share a set of events they recognize and to which they respond by executing a procedure. You create the code to execute for a form event the same way as you create other event procedures:

- Display the code window for the form
- Select the **UserForm** object
- Select the event from the **Procedure** list.



```
Option Explicit

Private Sub UserForm_Activate()


End Sub
```

Methods

UserForms also share methods that can be used to execute built-in procedures. Methods are normally used to perform an action in the form.

The three most useful methods are explained below:

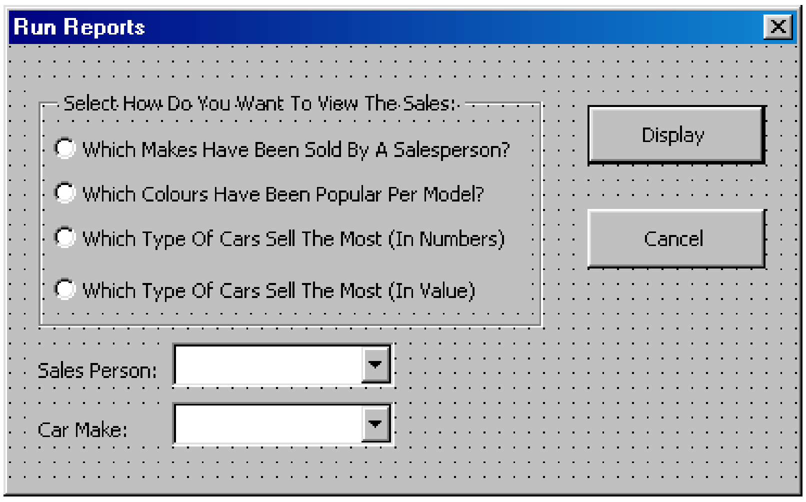
Show	Displays the form; can be used to load a form if not already loaded.
Hide	Hides the form without unloading it from memory.
Unload	Removes the form from memory.

Use the keyword **Me** in the UserForm's code module instead of its name to refer to the active form and access its properties and methods.

Understanding Controls

A control is an object placed on a form to enable user interaction. Some controls accept user input while others display output. Like all other objects controls can be defined by their properties, methods and events.

Below is an example of a form containing commonly used controls:


Run Reports
Select How Do You Want To View The Sales:
Which Makes Have Been Sold By A Salesperson?
Which Colours Have Been Popular Per Model?
Which Type Of Cars Sell The Most (In Numbers)
Which Type Of Cars Sell The Most (In Value)
Display
Cancel
Sales Person:
Car Make:

Control properties can be viewed and assigned manually via the Properties window. While each type of control is unique many share similar attributes.

The following list contains properties that are common among several controls:

Property	Description
ControlTipText	Specifies a string to be displayed when the mouse pointer is paused over the control
Enabled	Determines if the user can access the control.
Font	Sets the control text type and size.
Height	Sets the height of the control
MousePointer	Sets the shape of the mouse pointer when the mouse is positioned over the object
TabIndex	Determines the order in which the user tabs through the controls on a form.
TabStop	Determines whether a control can be accessed using the tab key.
Visible	Determines if a control is visible
Width	Sets the width of a control.

All controls have a default property that can be referred by simply referencing the name of the control. In one example the **Caption** property is the default property of the **Label** control.

This makes the two statements below equivalent:

```
Label1 = “Salary”
Label1.Caption = “Salary”
```

As with forms many controls respond to system events.

The following are the more common events that controls can detect and react to:

Click	Occurs when the user clicks the mouse button while the pointer is on the control
GotFocus	Occurs when a control receives focus
LostFocus	Occurs when a control loses focus
MouseMove	Occurs when a user moves the mouse pointer over a control.

Naming Conventions

It’s a good practice to use a prefix that identifies the control type when you assign a name to the control.

Below is a list of several control object name prefix conventions:

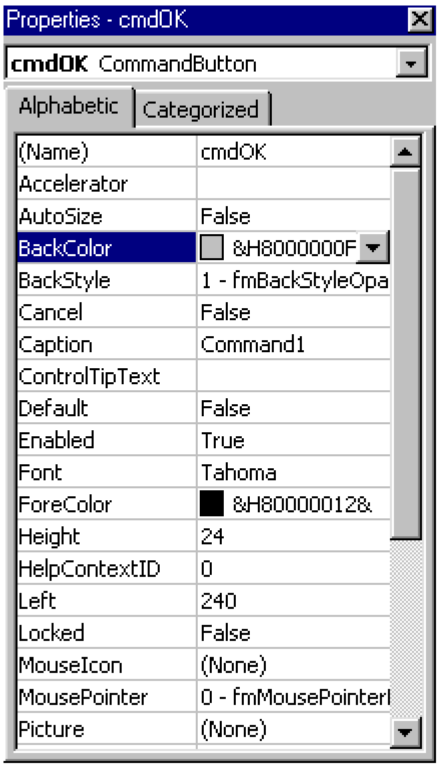
Object	**Prefix**
Check box	chk
Combo box	cbo
Command button	cmd
Frame	fra
Image	img
Label	lbl
List box	lst
Option button	opt
Text box	txt

Setting Control Properties in the Properties Window

Each control has a set of properties that can be set in the design environment using the Properties window. Categories for the property window vary per object.

Frequently used categories are behaviour, font, and position.

To set **Control Properties** in the Properties Window:


Properties - cmdOK

cmdOK CommandButton

Alphabetic | Categorized

(Name)	cmdOK
Accelerator	
AutoSize	False
BackColor	&H8000000F
BackStyle	1 - fmBackStyleOpa
Cancel	False
Caption	Command1
ControlTipText	
Default	False
Enabled	True
Font	Tahoma
ForeColor	&H80000012&
Height	24
HelpContextID	0
Left	240
Locked	False
MouseIcon	(None)
MousePointer	0 - fmMousePointerI
Picture	(None)

- Display the **Properties Window**
- Click the **Alphabetic** tab to display properties in alphabetic order **OR**
- Click the **Categorized** tab to display properties by category

To change a property setting:

- Select the desired control in the UserForm window or from the drop down list in the Properties window
- Scroll to the desired property and use the appropriate method to change the setting in the value column.

Notes

Using the Label Control

The **Label** control is used to display text on a form that cannot be modified by the user.

It can be modified in the procedure by using the **Caption** property.

Below are some unique properties of the **Label** control:

Property	Description
TextAlign	Determines the alignment of the text inside the label.
AutoSize	Determines if the dimensions of the label will automatically resize to fit the caption.
Caption	Sets the displayed text of the field.
WordWrap	Determines if a label expands horizontally or vertically as text is added. Used in conjunction with the **AutoSize** property.

Using the Text Box Control

The **Text Box** control allows the user to add or edit text. Both string and numeric values can be stored in the Text property of the control.

Below are some important properties of the **Text Box** control:

Property	Description
MaxLength	Specifies the maximum number of characters that can be typed into a text box. The default is 0 which indicates no limit.
MultiLine	Indicates if a box can contain more than one line.
ScrollBars	Determines if a multi-line text box has horizontal and/or vertical scroll bars.
Text	Contains the string displayed in the text box.

Using the Command Button Control

Command buttons are used to get feedback from the user. Command buttons are among the most important controls for initiating event procedures.

The most used event associated with the **Command Button** is the **Click** event.

Below are two unique properties of the **Command button** control:

Property	Description
Cancel	Allows the **Esc** key to "click" a command button. This property can only be set for one command button per form.
Default	Allows the **Enter** key to "click" a command button. This property can only be set for one command button per form.

Using the Combo Box Control

The **Combo Box** control allows you to display a list of items in a drop-down list box. The user can select a choice from the list or type an entry.

The items displayed on the list can be added in code using the **AddItem** method.

Below are some important properties of the **Combo Box** control:

Property	Description
ListRows	Sets the number of rows that will display in the list.
MatchRequired	Determines whether the user can enter a value that is not on the list.
Text	Returns or sets the text of the selected row on the list.

Some important methods that belong to the **Combo Box** are explained below:

AddItem *item_name, index*	Adds the specific item to the bottom of the list. If the index number is specified after the item name its added to that position on the table
RemoveItem *index*	Removes the item referred to by the index number.
Clear	Clears the entire list.

Using the Frame Control

The **Frame** control is used to group a set of controls either functionally or logically within an area of a **UserForm**. Buttons placed within a frame are usually related logically so setting the value of one affects the values of others in the group.

Option buttons is a frame are mutually exclusive, which means when one is set to true the others will be set to false.

Using Option Button Controls

An **Option Button** control displays a button that can be set to on or off. Option buttons are typically presented within a group in which one button may be selected at a time.

The **Value** property of the button indicates the on and off state.

Using Control Appearance

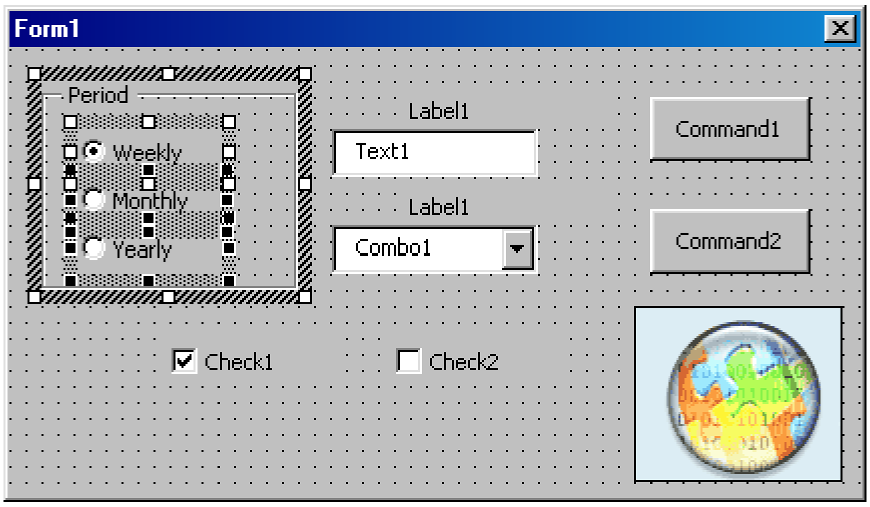
The UserForm toolbar provides several tools that are used to manipulate the appearance of the controls on the form.

Many of the tools on the UserForm toolbar require the user to select multiple controls. To do this:

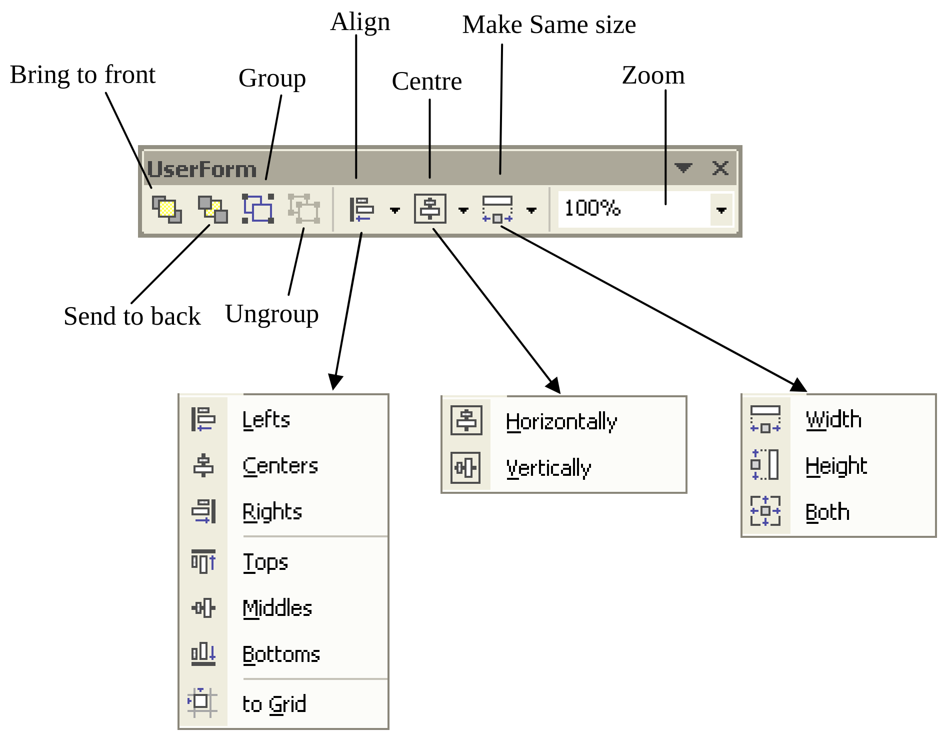
- Click the first control
- Hold down the **Shift** key
- Click any additional controls

Controls will be aligned or sized according to the first control selected. The first control selected is identified by its white selection handles.

Below is an illustration of a UserForm with multiple controls selected:

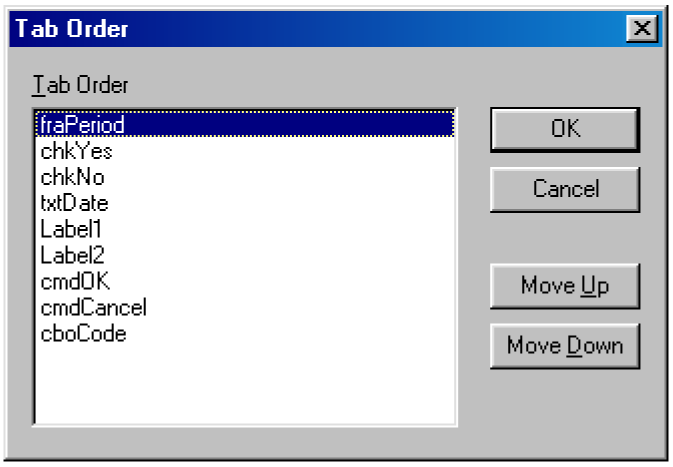


Below is an illustration of the **UserForm** toolbar together with the options for **Align**, **Centre** and **Make Same Size**.



Setting the Tab Order

The tab order is the order by which pressing the **Tab** key moves focus from control to control on the form. While the form is being built the tab order is determined by the order in which you place the controls on the form. If the controls are rearranged you may nee to manually reset the tab order. To set the tab order:



- View the desired form in the **UserForm** window
- Open the **View** menu
- Choose **Tab Order**
- Select the desired control from the list
- Click **Move Up** to move the control up the list
- Click **Move Down** to move the control down the list

Although **Labels** are listed on the **Tab Order** dialog box, they are not included in the tab order.

Filling a Control

A list box or combo box control placed on the form is not functional until the data that will appear on the list is added.

This is done by writing code in the sub procedure associated with the **Initialize** event. This triggers when the form is loaded. The **AddItem** method is used to specify the text that appears in the list.

The code below shows items added to a combo box named cboCourses:

```
With cboCourses
    .AddItem “Excel”
    .AddItem “Word”
    .AddItem “PowerPoint”
End With
```

Adding Code to Controls

As seen, forms and their controls are capable of responding to various events. Adding code to forms and control events are accomplished the same way as adding code to events of other objects.

How to Launch a Form in Code

The **Show** method of the form object is used to launch a form within a procedure.

Creating a procedure to launch a form enables you to launch a form from a toolbar, or menu as well as from an event such as opening a workbook.

Below is the syntax used to launch a form:

```
FormName.Show

frmNewData.Show
```

Unit 9 Using the PivotTable Object

Understanding PivotTables

A pivot table is a table that can be used to summarize data from a worksheet or an external source such as a database.

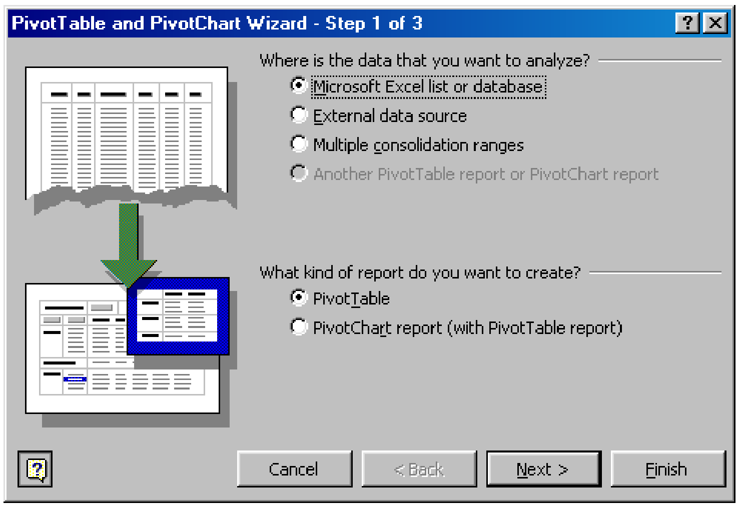
A Pivot table can only be created using the Pivot table wizard.

Creating A PivotTable

The wizard makes the creation of the pivot table quite easy. By following a series of prompts the wizard takes over and creates the pivot table for you. To do this:

- Pull down the **Data** menu
- Select **Pivot Table and Pivot Chart Report…**

The **PivotTable and PivotChart Wizard – Step 1 of 3** dialog box appears.



PivotTable and PivotChart Wizard - Step 1 of 3

Where is the data that you want to analyze?

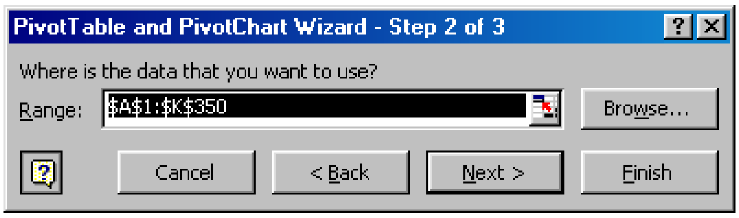
- [✓ Microsoft Excel list or database]
- External data source
- Multiple consolidation ranges
- Another PivotTable report or PivotChart report

What kind of report do you want to create?

- [✓ PivotTable]
- PivotChart report (with PivotTable report)

- Select **Where the data is that you want to analyze**
- Select **What kind of report you want to create**
- Click **Next**.

The **PivotTable and PivotChart Wizard – Step 2 of 3** dialog box appears.



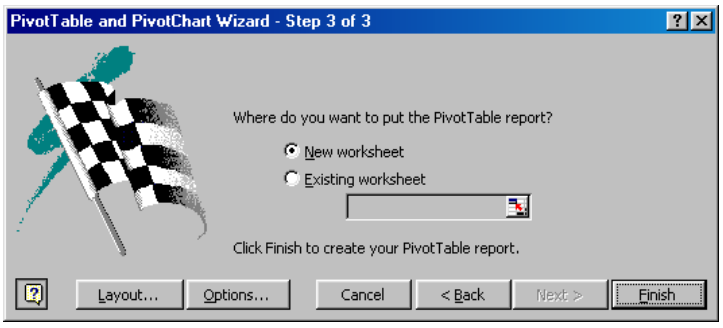
PivotTable and PivotChart Wizard - Step 2 of 3

Where is the data that you want to use?

Range: $A$1:$K$350

- The selected range appears in the **Range** window
- Change the range if needed
- Click **Next**.

The **PivotTable and PivotChart Wizard – Step 3 of 3** dialog box appears.



Where do you want to put the PivotTable report?

(● New worksheet)
○ Existing worksheet

Click Finish to create your PivotTable report.

- Select Where do you want to put the Pivot Table
- Click **Finish**
- Drag the field buttons to the desired page, row, column and data fields.

Using the PivotTable Wizard Method

The **PivotTable Wizard** method of the Worksheet object can be used to create a pivot table in code without displaying the wizard.

The **PivotTable Wizard** method has many arguments. The main ones are described below:

Argument	Definition
SourceType	The source of the PivotTable data. The SourceData argument must also be specified when using this.
SourceData	A range object that specifies the data for the PivotTable.
TableDestination	A range object indicating where the table will be placed.
TableName	The name by which the table can be referred.

An example of the **PivotTable Wizard** method is shown below:

```
Sub MakePivot ()

Dim DataRange As Range
Dim Destination As Range
Dim PvtTable As PivotTable

Set Destination = Worksheets("Sales Summary").Range("A12")
Set DataRange = Range("A9", Range("J9").End(xlDown))

ActiveSheet.PivotTableWizard SourceType:=xlDatabase, _
SourceData:=DataRange, TableDestination:=Destination, TableName:="SalesInfo"

End Sub
```

This code runs the PivotTable wizard, capturing the data in the current worksheet then placing a pivot table in the worksheet called “Sales Summary”. In this instance the PivotTable contains no data, because the row, column and data fields haven’t been assigned.

Using PivotFields

Once a PivotTable is created pivot fields must be assigned. The **PivotFields** collection is a member of the PivotTable object containing the data in the data source with each Pivot Field getting its name from the column header. PivotFields can be set to page, row, column and data fields in the PivotTable.

In the Sales – April 2004 the fields are: Sales Date, Make, Model, Type, Colour, Year, VIN Number, Dealer Price, Selling Price, Salesperson.

The table below lists the PivotTable destinations for PivotFields.

Destination	Constant
Row Field	**xlRowField**
Column Field	**xlColumnField**
Page Field	**xlPageField**
Data Field	**xlDataField**
To Hide A Field	**xlHidden**

The following syntax shows how a PivotField is defined by setting its Orientation property to the desired destination column:

```
.PivotTables(Index).PivotFields(Index).Orientation = Destination

.PivotTables(“SalesInfo”).PivotFields(“Salesperson”).Orientation = xlPageField

PivotTables(“SalesInfo”).PivotFields(“Colour”).Orientation = xlRowField
```

To optimize the setting of the Pivot Table orientation use the With Statement:

```
Set PvtTable = Sheets(“Sales Summary”).PivotTables(“SalesInfo”)

With PvtTable

        .PivotFields(“Salesperson”).Orientation = xlPageField
        .PivotFields(“Year”).Orientation = xlRowField
        .PivotFields(“Make”).Orientation = xlColumnField
        .PivotFields(“Selling Price”).Orientation = xlDataField

End With
```

Notes

Excel VBA – Quick Reference Guide

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