Microsoft Office Training Series

Excel VBA

Intermediate

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

STL

MicrosoftTraining.net

Learning & Performance Institute
Accredited Learning Provider

ilm
Recognised Provider

Microsoft Partner
Certified Silver Partner

Welcome to your Excel Intermediate training course

- Work with variables in VBA codes
- Create message boxes & input boxes
- Create user defined functions
- Work with error handling
- Create user forms

Microsoft Office Training Series

Professional Development Series

Microsoft Technical Series

MicrosoftTraining.net/Feedback

Contents

Unit 1 Using Intrinsic Functions, Variables and Expressions

In this unit you will learn how to:

- Defining Expressions and Statements
- Declare variables
- Data type variables & object variables
- Assign values to variables
- Program with variables
- Create constants

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
```

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.

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

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

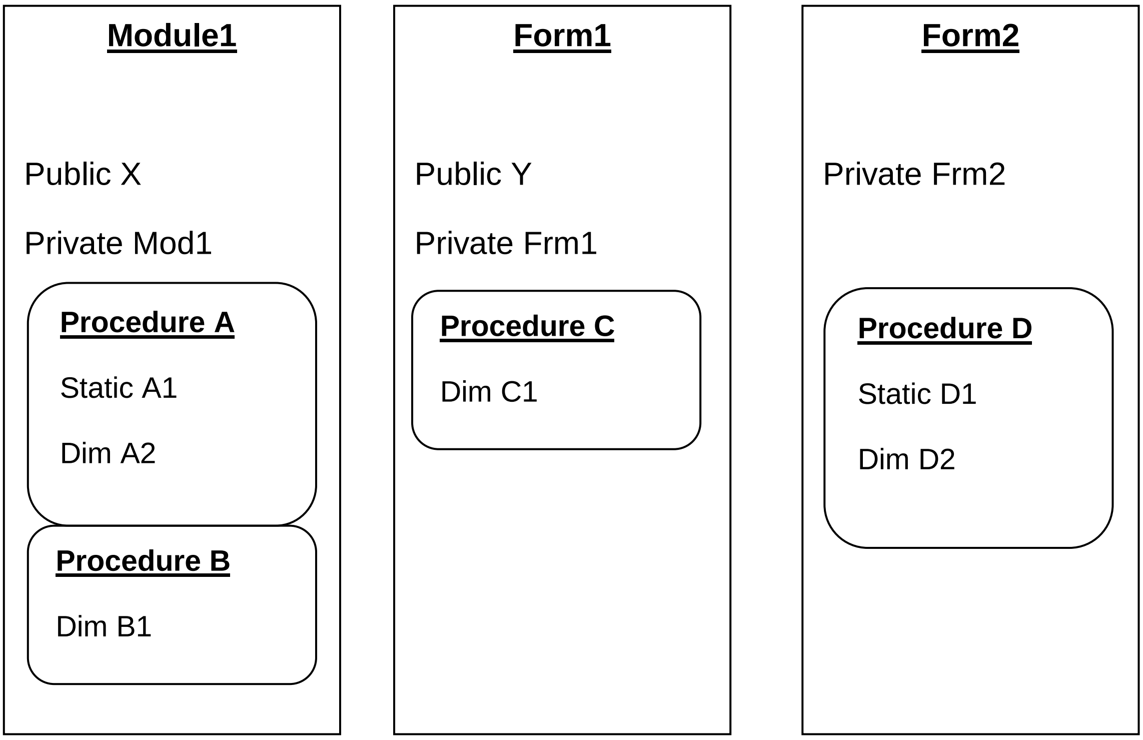
The **Public** keyword can only be used in the **Declarations** section

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)**

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

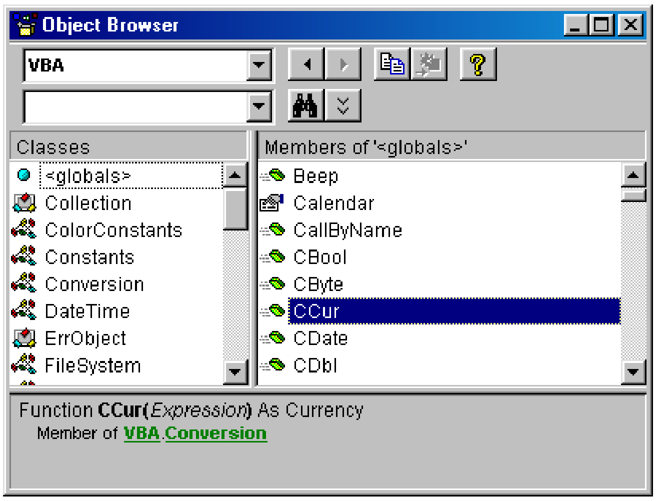
Procedure C can see: C1, Frm1, X, Y

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

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*>

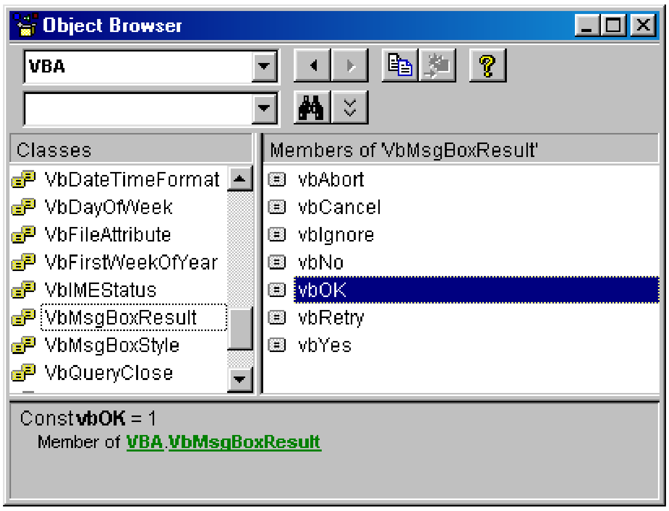
Const conPassMark As String = “C”

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:



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	☐ 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.

Unit 2 Creating User Defined Functions

In this unit you will learn how to:

- Create user defined functions (UDF)
- Call a UDF from procedure and worksheet

Creating a Function Procedure

While Excel Worksheet functions can be called within the code window there may be times when none of these functions meet your needs. You may also wish to simplify or obscure a complex function that has already been written for the benefit of the end user. This is when a user-defined function can be created.

Unlike a sub procedure, a function procedure only returns a value. It cannot move around a workbook, select worksheets, select cells, change properties nor apply methods to objects. Like the regular Excel and VBA functions (eg. SUM, COUNT, LOOKUP, IF, PMT, DATE, etc...), they are limited to carrying out a calculation and returning the result.

A function is enclosed by the statements **Function** and **End Function**. The function procedure must have a name that is often followed by arguments inside brackets. The arguments are variables that the function needs to calculate and resolve to a value.

Once written in the code window, a function can then be called in Excel by simply typing the name of the function after an equals sign, just as you would any other Excel function. The function name should be suggested in the formula drop-down.

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 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.

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

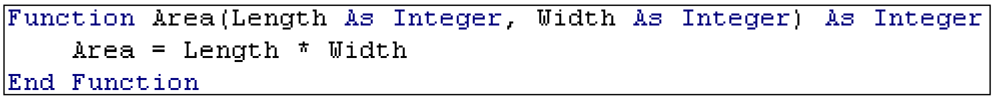
- Open the **Insert** menu
- Select **Procedure**.

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:

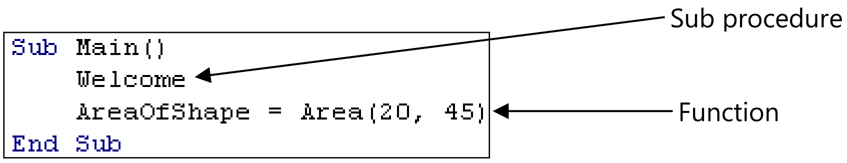


```
Function Area(Length As Integer, Width As Integer) As Integer
    Area = Length * Width
End Function
```

Calling a UDF

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.

Using a function within an Excel Workbook

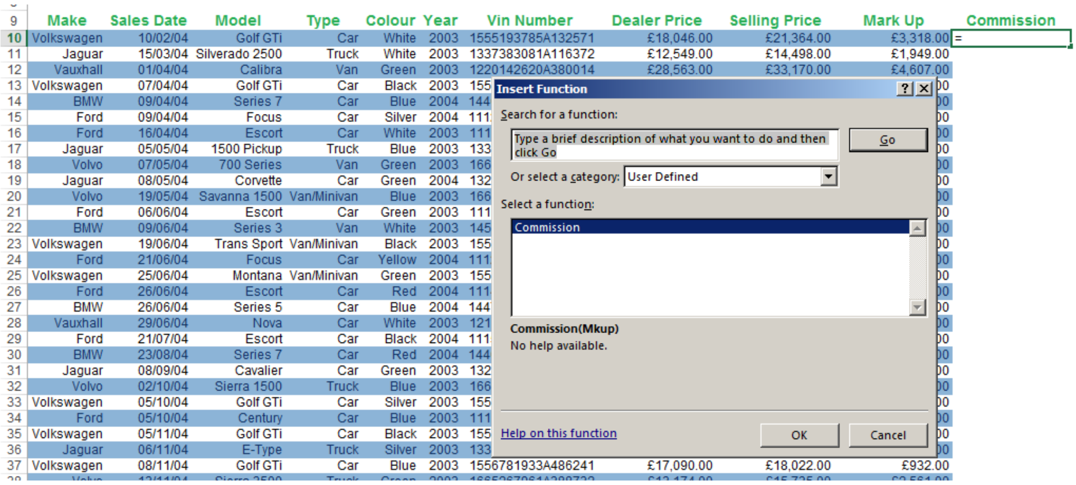
As well as being called by procedures functions can also be used from within an Excel worksheet. For example, a commission function typed into the code window can be used in a worksheet to calculate the correct commission for a sales value.

First the code for Commission function is typed within a new Module named as MyFunctions:

```
Function Commission(Mkup As Currency)



   If Mkup >= 1000 Then

     Commission = Mkup * 0.1

     ElseIf Mkup >= 500 Then

        Commission = Mkup * 0.05

        ElseIf Mkup >= 100 Then

           Commission = Mkup * 0.01

     Else

        Commission = 0
   End If


End Function
```

The Mkup argument refers to the difference between selling and dealer price to which the commission calculation is applied.

To use this function select a blank cell and click the **Formulas** tab**, Insert Function** and choose the function from the **User-Defined** category.



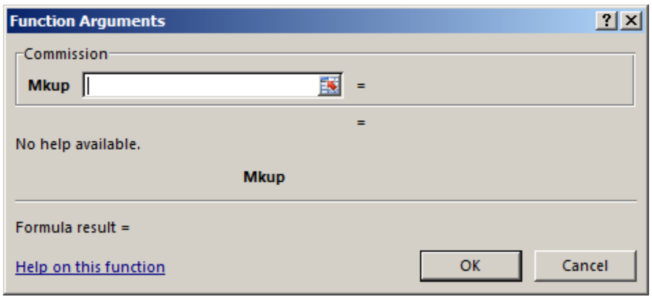
Insert Function

Search for a function: Type a brief description of what you want to do and then click Go

Or select a category: User Defined

Select a function: Commission

Commission(Mkup)
No help available.



Function Arguments

Commission

Mkup =

No help available.

Mkup

Formula result =

Select the Mark Up cell or type J10 and press OK to enter the commission result. The same result can be achieved by typing:

=commission(J10)

Into the blank commission cell.

Unit 3 Message Boxes and Input Boxes

In this unit you will learn how to:

- Create a message box
- Add interactivity to a message box
- Create input boxes

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])

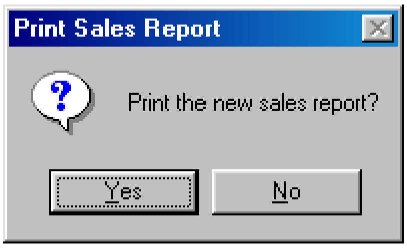
```
MyResponse = MsgBox (“Print the new sales report?”, 36, _
        “Print Sales Report”)

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:

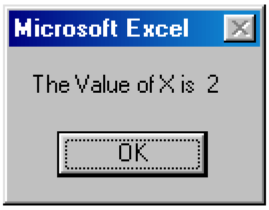
Another example of using the message box is to return a value:

```
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.



Print the new sales report?



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.

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.

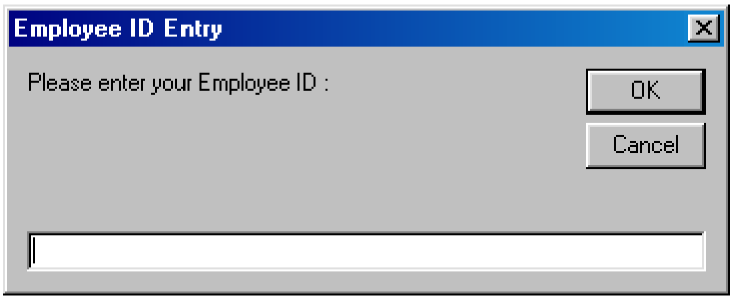
In this case just call the **MsgBox Function** with the syntax used to call a sub procedure as shown below:

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

Or

MsgBox “You must enter a number”
```

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.

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 4 Handling Errors

In this unit you will learn how to:

- Trap errors
- Capture errors with on error statement
- Create error handling routine
- Use inline error handling

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.

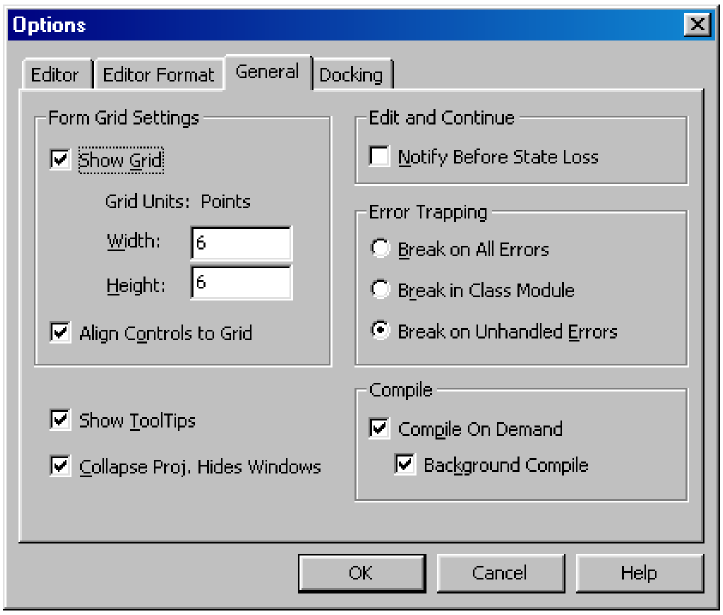
Defining VBA's Error Trapping Options

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** - The **Options** dialog box appears.

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



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

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.

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
```

Unit 5 Creating Forms and Controls

In this unit you will learn how to:

- Understand a userform
- Use the tool box
- Create user form events

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:

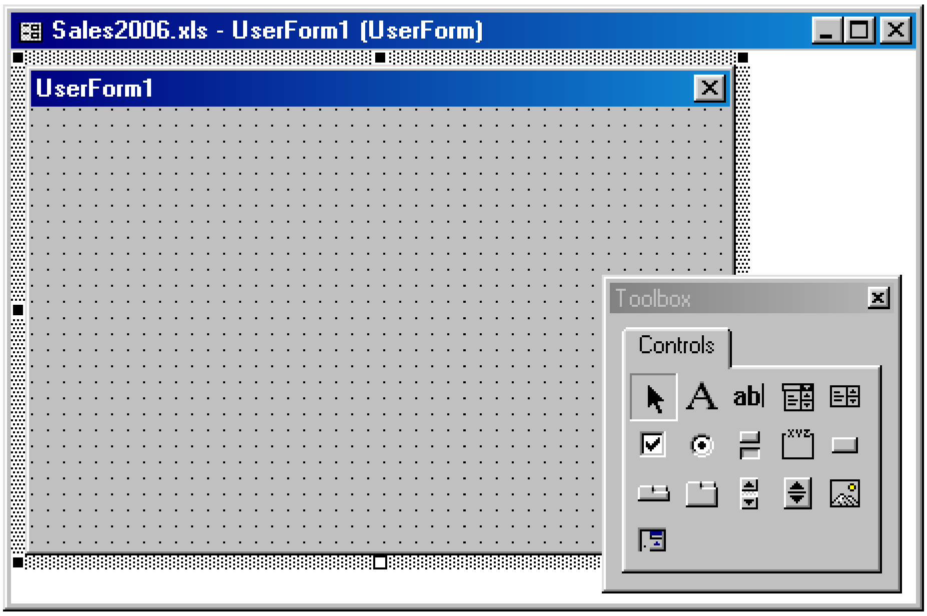
In the **Visual Basic Editor**, select the desired Project name in the Project Explorer.

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**.

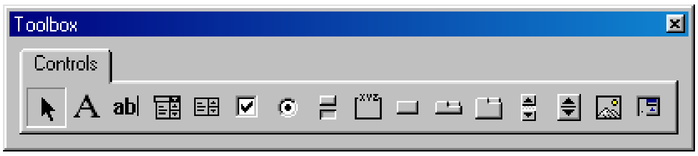
A blank user form appears together with the toolbox.

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.

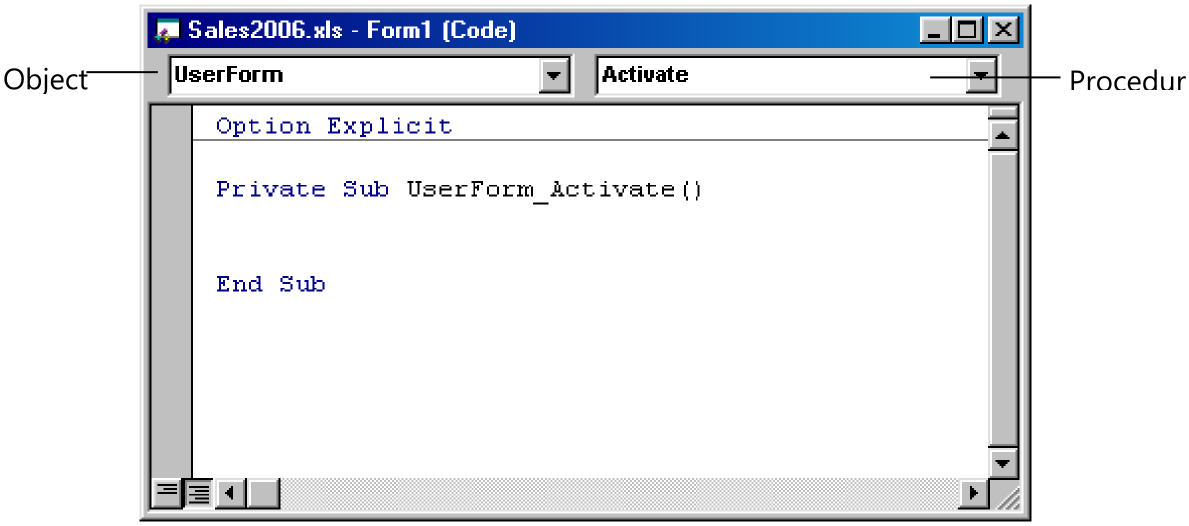
The following list describes the more commonly used properties of a UserForm:

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.

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:

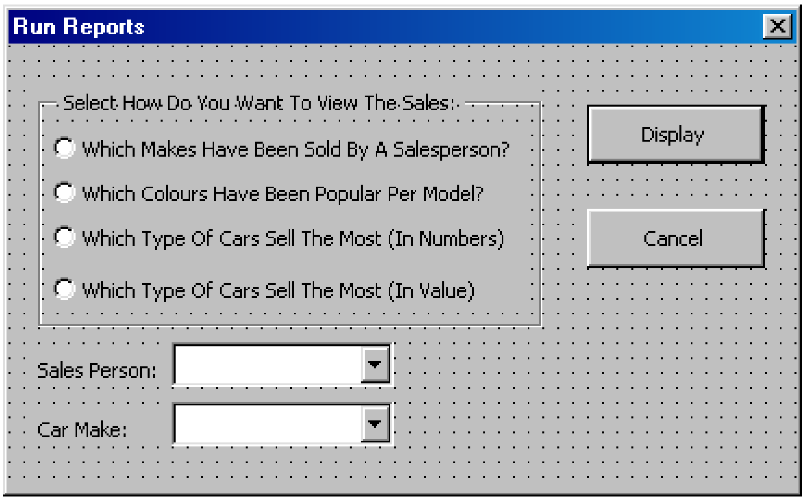
Show	Displays the form; can be used to load a form if not already loaded.
Hide	Hides the form without unloading it from memory.
Unload	Removes the form from memory.

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:

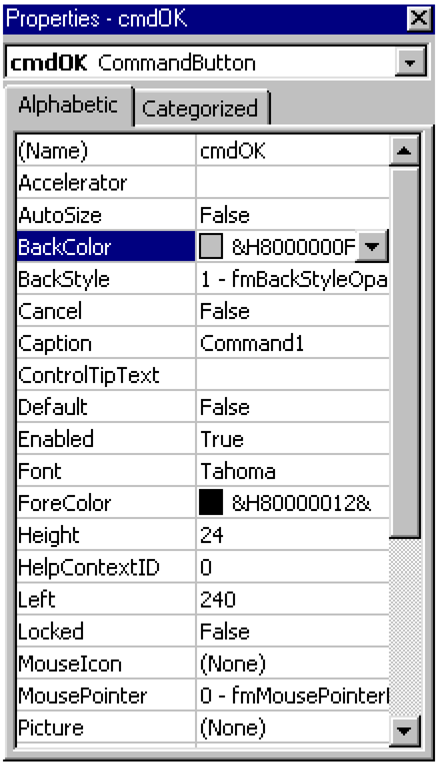
Object	**Prefix**
Check box	chk
Combo box	cbo
Command button	cmd
Frame	fra
Image	img
Label	lbl
List box	lst
Option button	opt
Text box	txt

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

(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 - fmMousePointerl
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.

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.

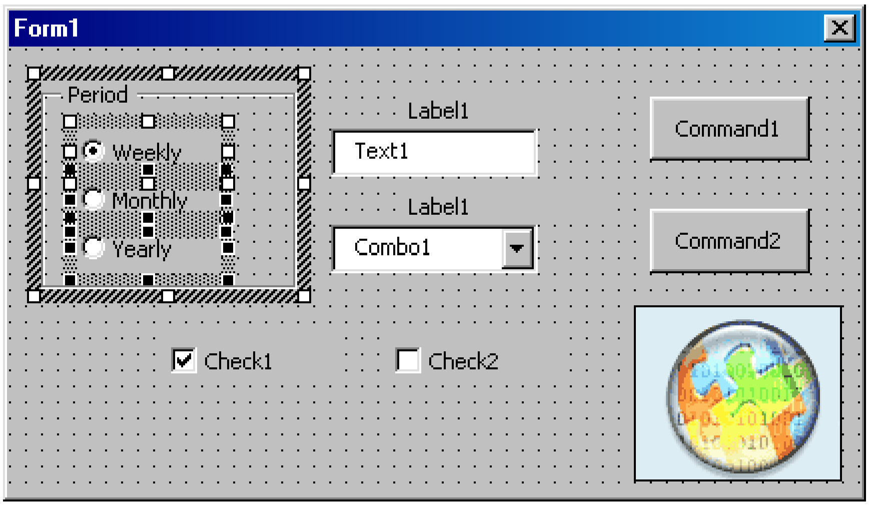
Using Control Appearance

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

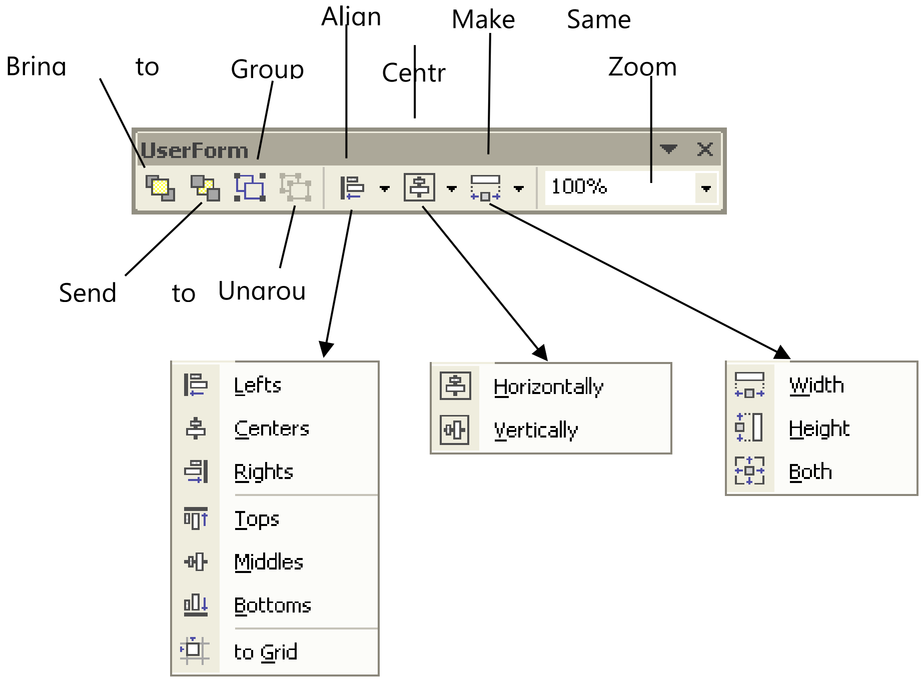
Many of the tools on the UserForm toolbar require the user to select multiple controls. To do this:

- 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.

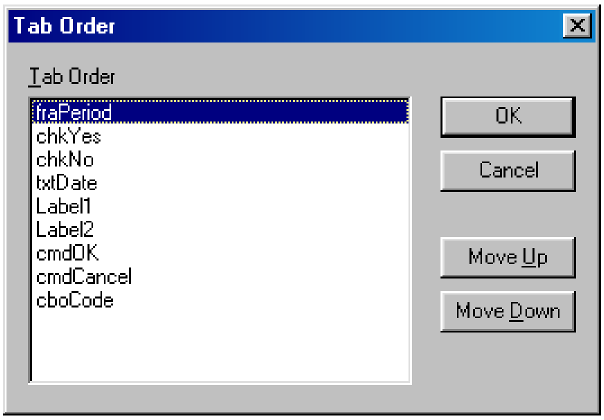


Below is an illustration of a UserForm with multiple controls selected: Below is an illustration of the **UserForm** toolbar together with the options for **Align**, **Centre** and **Make Same Size**.



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
```

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: Loals 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	Type “VB Constants” in VBE Help to view the selection of VB Constants available

```
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
```

	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

E&OE

STL Training reserves the right to revise this publication and make changes from time to time in its content without notice.