Creating Applications using Excel
Macros/Visual Basic for Applications (VBA)

A *macro* is a sequence of instructions that tells Microsoft Excel what to do. These macros allow you to automate everyday tasks and custom features to suit your needs and even create applications. The instructions for the macros are written in a computer programming language called Visual Basic or Visual Basic for Applications (VBA).

The VBA environment is different from that of worksheets and charts; instead of working with cells, rows, and columns, you will be using tools to help you communicate in the language of Excel.

With VBA you can create custom commands, menus, dialog boxes, messages, and buttons.

All *code*, known as VBA statements, are stored in a *module*. A module is a special type of sheet that allows you to enter the code. A module sheet can be created by using the VB Editor (which is new to Excel 97, but first developed in Excel Version 5.0) by choosing the **Macro** command from the **Tools** menu and then selecting **Visual Basic Editor**. The VB Editor can also be opened by pressing Alt+F11.

In earlier versions of Excel, the VB Editor was not present. The module was created by selecting the **Macro** command for the **Insert** menu and choosing **Module**. Then a module tab appears at the bottom of the worksheet alongside the worksheet tabs.

All of your code is written in the VB Editor in the module. Initially, the VB Editor shows a blank screen with two windows labeled: Project - VBA and Properties - Module1.

To insert a module in the VB Editor, choose the **Module** command from the **Insert** menu. This displays the code window. The code window allows you to write, display, and edit VBA statements.

Automating Repeated Tasks

When working in Excel, you will sometimes find yourself performing certain tasks over and over again. These tasks can be automated using Visual Basic programming language in Excel.

Initially, you don't need to understand a programming language to put VBA to work. MS Excel includes a Macro Recorder, which is a tool that creates VB code for you. We will discuss the basic steps for using the Macro Recorder to automate a simple task.

After recording a macro, we can edit and customize it to suit our needs.

Recall that a macro is a series of commands that are automatically carried out by Excel. For example, to format a range we would:

> Choose Cells command from the Format menu. Then select the Font tab. Select a font name, style, and size. Then choose OK.

With a macro, we can combine these tasks into a single step. So, by recording the macro, we can work more efficiently with Excel.

After a macro is recorded, we can assign it to a menu item or a button. Then running the macro is as simple as clicking a button.

Whenever you find yourself regularly typing the same keystrokes, choosing the same commands, or going through the same sequence of actions, you should consider recording a macro..

For example, suppose each time you set up a worksheet, you needed to:

Turn off gridlines
Select cell C3
Enter the title "West Coast Sales"
Format the title and select the font and font size
Make the title bold and italic
Place a border around the title cell
Apply dark blue color to the title cell
Widen column C to accommodate the title

To speed up this process, we can record a macro. Then the next time we open a workbook, we can run the macro and Excel will follow the same sequence of steps.

Recording a Macro

In Excel, the Macro Recorder stores the actions you perform or commands you choose as you work. The steps to recording a macro are as follows:

- From the **Developer's tab**, choose the **Macro** command and then choose **Record New Macro**.
- In the Macro Name box, type a name for the macro
- In the Description box, type a description of the macro
- Choose OK
  While the Macro Recorder is working, the Stop Macro button appears on your screen on its own toolbar.
- Carry out the actions you want to record
- Click Stop Macro button

If you don't give the macro a name, Excel will name it automatically [Macro*n*, where *n* is the first number that gives the macro a unique name].

To run the recorded macro, perform the following actions:

- Switch to a new worksheet
- Select cell A1
- From the **Developer's tab**, choose **Macros…**
- In Macros dialog box, select the Macro Name
- Choose Run

The macro will then run the series of recorded steps/actions.

After you have recorded the macro, you can:

- Edit the recorded macro
- Add the macro to the tools menu
- Assign it to a button on a sheet
- Assign it to a button on a toolbar.

Here are some tips to using the Macro Recorder:

- Plan what you want to do before you begin recording
- Select the cells and objects first and then start recording
- Switch to the appropriate workbook and select the appropriate sheet before you turn on the recorder
- Use the macro recorder as a learning tool

Creating Applications

Before you can create Excel applications, you must have not only a good understanding of the functionality that various Excel objects offer but also a firm knowledge of the structure of the Excel object model. An *object* in Excel is something that can be programmed or, in essence, controlled. Excel's object model contains 128 different objects, ranging from simple objects such as rectangles and textboxes to complicated objects such as pivot tables and charts. Each object in Excel has You can create applications in Excel by tying these objects together using Visual Basic for Applications (VBA), Excel's new macro language.

The Excel Object Hierarchy

Of the 128 different objects in Excel, not all exist on the same level--that is, some objects are contained within others. You can think of these levels as tiers in a hierarchy. The topmost tier of the Excel object hierarchy is occupied by a single object: *Application*. The Application object represents Excel itself, and all other Excel objects fall under Application. To manipulate the properties and methods of an object, you must sometimes reference all objects that lie on the hierarchical path to that object. You traverse that path down to a specific object by using the dot operator (.). For example, let's suppose you are writing a VBA macro to set the Value property of a Range object that represents the first cell in the first worksheet of the first workbook in Excel. Using the full hierarchical path, the reference to Range appears as follows:

Application.Workbooks(1).Worksheets(1).Range("A1").Value = 1

Objects: Their Properties and Methods

All objects in Excel have what are known as *properties* and *methods*. VBA is a tool through which you can control Excel objects by manipulating their properties and methods. *Properties* are attributes that control the appearance or behavior of an object. You can set and return values of properties for objects in you Visual Basic code.

In addition to properties, objects also have *methods* -- actions that objects can do. You use methods in VB code to cause objects to do things we want them to do.

When using objects, the code you write usually does one of three things:

- Changes the conditions of an object by setting the value of one of the object's properties
- Examines the conditions of an object by returning the value of one of the object's properties
- Causes the object to perform a task by using a method of the object

For example, suppose you want to determine if a range on a worksheet is empty. If it is not, empty it. Once the range is empty, either because it was already or you cleared it, you want to assign a formula to the range. The VB code would:

- Use a Range object to identify the range you want to examine
- Return the value of the Value property of the range to determine whether the range is empty
- Use the Clear method of the range to clear all cells if the range is not empty
- Set the Formula property of the range to assign a formula to the range

Some properties of the Range object are:

Property	**Description**
Column	Returns the first column of the first area in the range
Formula	Returns the range's formula, in A1-style notation
Height	Returns the height of a range, in points (1/72 inch)
WrapText	Determine whether text wraps inside the range
Width	Returns the width of the column in a range

When you refer to a property, the object whose property you want to set or return comes first, followed by a period and then the name of the property. For example, the code ***Cells.ColumnWidth = 16*** refers to the ColumnWidth property of the Range object. The Cells method returns a Range object. Note that the object and it's property are separated by a period.

When dealing with properties using VBA in Excel, you can perform two types of actions: You can set the value of property, or you can return the value of a property setting. The value of a property is one of three types: numeric, character string, or Boolean (True/False). The syntax for setting a property is:

object.property = expression

where *object* is a reference to an object, *property* is the name of a property of the object, and *expression* is the value to which you want to set the property. The following statements demonstrate how you set properties:

Cells.ColumnWidth = 16
ActiveCell.RowHeight = 14
ActiveCell.Value = "Annual Totals"

You return a property value when you want to determine the condition of an object before your procedure performs additional actions. For example, you can return the Value property of a cell to determine the contents of the cell before running code to change the values of the cell. To return the value of a property, you use the following syntax:

variable = object.property

Where *variable* stands for a variable or another property in which you store the returned property value, *object* is a reference to an object, and *property* is the name of a property of the object.

Some common properties to many objects in Excel are important properties as you learn and start using VBA. The following table lists some of these properties and a brief description:

Property	**Description**
ActiveCell	The active cell of the active window
ActiveSheet	The active sheet of the active workbook
ActiveWorkbook	The active workbook in Excel
Bold or Italic	The type style of the text displayed by a Font object
Column or Row	A number that identifies the first column or row of the first area in a range
ColumnWidth	The width of all columns in the specified range
Height or Width	The height or width of an object, in points. This property applies to many different objects in Excel.
RowHeight	The height of all the rows in the specified range, in points
Selection	The object currently selected in the active window of the Application object, or the current selection in the given Window object
Value	The value of a cell

Actions with Methods

Methods are a part of objects just as properties are. The difference between methods and properties is that properties have values which you set or return, while methods are actions you want an object to perform. Also, most properties take a single value, whereas methods can take on one or more arguments.

When using a method in your procedure, how you write the code depends on whether the method takes arguments. If the method does not take arguments, the syntax is:

object.method

where *object* is the name of the object and *method* is the name of the method. For example, the **Justify** method of a Range object doesn't take argument. To justify cells in a Range object named Price, you write

Price.Justify

If the method does take arguments, the syntax depends on whether you want to save the value returned by the method. If you don't want to save the return value, the arguments appears without parentheses, as show below

object.method arguments

An example would be:

Range("Price").Table Cells(2,1) Cells(3,1)

If you do wish to save the return value, you must enclose the arguments in parentheses. In this case, the above code would be written as:

Range("Price").Table(Cells(2,1) Cells(3,1))

Some common methods used on objects are:

Method	**Description**
Calculate	Calculates a specified range of cells in a sheet
Clear	Clears the contents of an entire range
Copy	Copies the range to the Clipboard

Justify	Rearranges the text so that it fills the range evenly
Table	Creates a data table based on input values and formulas that you define

Collections

In Excel, a *collection* is a set of related objects. For example, the Worksheet collection contains all worksheets in a given workbook. Each object within a collection is called an element of that collection. Collections are objects themselves, which means they have their own properties and methods. You can use properties and methods to control individual elements of a collection or all of the elements of a collection.

Some common collections are:

Collection	Description
Sheets	Contains all the sheet objects in a workbook
Worksheets	Contains all the worksheets in a workbook
Charts	Contains all the charts in a workbook
Workbooks	Contains all the open workbooks in Excel

VBA Procedures

A procedure is the basic unit of code in VBA. It is a block of code that tell Excel what to do. VBA has two main procedures: *Sub* and *Function*. Procedures begin with the word *Sub* or *Function* followed by a space, the name you give the procedure, and a set of parentheses. A procedure ends with the corresponding words *End Sub* or *End Function*. In between are your lines of code; the structure looks like this:

```
Sub EnterDateAndTime()
        … Code
        … Code
End Sub

Function SafeSqr(arguments)
        … Code
        … Code
        … SafeSqr = expression
End Function
```

A *Sub* procedure, which represents a subroutine, performs actions but does not return a value. The *Function* procedure is similar to a *Sub* procedure, but it returns a value. When you call a Function procedure that has arguments and you use its return value, you must enclose the arguments in parentheses.

To specify a module when you call a procedure, type the name of the module enclosed in square brackets, followed by a period and the name of the procedure.

To call a procedure in another workbook, you need to establish a reference from the Visual Basic Editor to that workbook. After establishing a reference, the modules in that workbook are available to you.

To establish a reference, do the following:

- If you are working in a workbook, and the workbook to which you want to establish a reference (a second workbook) has not been saved, then switch to the second workbook and save it.
- Switch back to the first workbook and, with the VB Editor active, choose References from the Tools menu.
- In the Available References box, select the name of the second workbook. If the workbook name does not appear, then choose the Browse button.
- Select the check box next to the name of the second workbook, if it is not already selected.
- Choose the OK button

To call the procedure, type the name of the workbook enclosed in square brackets, followed by a period, and the name of the module enclosed in square brackets, followed by a period and the name of the procedure. The syntax is as follows:

[Workbookname.xls].[modulename].procedurename

You can name procedures however you want as long as the first character is a letter and you leave out any spaces or periods. You can run a procedure from the module in which it is located or by choosing the **Run** button on the VBA toolbar within the VB Editor.

Performing Multiple Actions on Objects

Procedures often need to perform several different actions on the same object. For example,

```
ActiveSheet.Cells(1,1).Formula = "=SIN(180)"
ActiveSheet.Cells(1,1).Font.Name = "Ariel"
ActiveSheet.Cells(1,1).Font.Bold = True
ActiveSheet.Cells(1,1).Font.Size = 8
```

Note that the statements use the same object reference. You can make this code easier to enter using a *With* statement:

```
With ActiveSheet.Cells(1,1)
        .Formula = "SIN(180)"
        .Font.Name = "Ariel"
        .Font.Bold = True
        .Font.Size = 8
End With
```

or we can nest *With* statements:

```
With ActiveSheet.Cells(1,1)
        .Formula = "SIN(180)"
        With .Font
                .Name = "Ariel"
                .Bold = True
                .Size = 8
        End With
End With
```

Range object:

The Range object can consist of:

- A cell
- Row or columns
- One or more selections of cells
- A 3-Dimensional Range

The most common way to identify a Range object is with the Cells method. The Cells method returns a Range object which is a collection of cells.

Cells are assigned in rows and columns, so the best way to specify a cell is using indexes. Identify rows and columns by passing one index for the row and another for the column. For example, cell A1 on the active worksheet can be identified as:

```
Cells(1,1).Value = 24       'set value of cell A1 to 24
                    or
Cells(1,"A").Value = 24     'set value of cell A1 to 24
```

Using the number index is more efficient because it will allow you to use the cell "addresses" in the procedures as variables.

For simple procedures, a shortcut syntax can be used: Enclose the absolute, A1-style reference in square brackets. For example, *[A1].Value = 24*.

The Option Base Statement

Note that VBA allows lower bounds of arrays to start at either 0 or 1 by default, as governed by an *Option Base* statement at the beginning of a VBA module. Option Base 0 makes the default lower bound of the array 0, and Option Base 1 makes the default lower bound of the array 1. In the absence of an Option Base statement, array lower bounds are 0 by default.

Calling One Macro from Another

In VBA, it's possible to call, or execute, one macro from another macro using the CALL statement, which transfers control to a Sub procedure, Function procedure, or DLL procedure. This capability allows you to separate your code into logical segments. You gain two advantages by separating code into multiple subroutines. First, if you want to use a VBA routine repeatedly, you need only write this routine once and store it in a macro that can be called by any macro that requires it. Second, you can separate your VBA code into discrete, logical segments that are easy to code, debug, and maintain.

Syntax

Call *name* [*argumentlist*]

The **Call** statement syntax has these parts:

Part	Description
Call	keyword. If specified, you must enclose *argumentlist* in parentheses. For example: Call MyProc(0)
name	Required. Name of the procedure to call.
argumentlist	Optional. Comma-delimited list of variables, arrays, or expressions to pass to the procedure. Components of *argumentlist* may include the keywords **ByVal** or **ByRef** to describe how the arguments are treated by the called procedure. However, **ByVal** and **ByRef** can be used with **Call** only when calling a DLL procedure. On the Macintosh, **ByVal** and **ByRef** can be used with **Call** when making a call to a Macintosh code resource.

In the following example, a Display Message macro is called to display a message (although this code may not be practical):

```
Sub CallSecondMacro()
        Dim Range1 As Range
        Set Range1 = Worksheets(1).Range("A1")
        Range1.Value = 500
        Call DisplayMessage
End Sub

Sub DisplayMessage()
        MsgBox "Data has been entered."
End Sub
```

Declarations

The part of the module before the first procedure is called the declaration section; it contains the code that affects the entire module or workbook. This is the section where you declare your variables and constants, and options. You can place the items in the declarations section any way you want, but programmers convention usually specify options, variables, and then constants. An example of declarations follows:

```
'Declarations Section

'VBA Options
Option Explicit
Option Compare Binary

'Declare Variables
Dim OriginalPrice
Dim NumberOfShares
Dim SalePrice

'Declare Constants
Public Const DOWJONES = 3500
Public Const S_P500 = 290
```

' End of Declarations Section. Procedures Follow.

The following table lists the possible statements that may occur in a declaration statement:

Statement	**Description**
Option Private Module	Procedures and variables in the module are available only within the workbook
Option Explicit	Requires that all variables within the current module be explicitly declared
Option Base 0 Option Base 1	These two statements define the default lower bound of arrays
Option Compare Binary	When comparing strings in the module, lowercase and uppercase letters are equivalent
Public *variable_name*	Declares a variable for use within any procedure in the workbook. If Option Private is not used, the variable will be available to procedures in other workbooks
Private *variable_name*	Declares a variable for use only within the current module
Dim *variable_name*	Equivalent to Public *variable_name*.
Const = *expression*	Defines a string that can be used in the place of a number.
Declare Sub() Declare Function	These two statements define a reference to a DLL or Windows API
Type *Type_Name* End Type	These two statements create a user-defined data type.

Note: Well written code is well-documented. VBA provides an easy method for entering comments: Everything to the right of a single quotation mark is a *comment*. The single quotation mark can be at the start of the line or before the first word in the line.

Also, you don't have to identify variables and arguments before you use them. VB will recognize and handle the items automatically. But sometimes it is good to use the *Option Explicit* statement at the top of the module to "force" you to declare all of the variables.

Variable Data Types

There are 12 VBA *data types*. A data type is a characteristic of a variable that determines what kind of data it can hold. The following table lists the data types and a brief description.

Data Type	Description
Integer	Any whole number between -32,768 and 32,767
Boolean	Has only two values, True or False
Long	Any whole number from -2.1B to 2.1B
String	Used to build text for message boxes, status bars, and input boxes. Can have fixed-length and variable-length.
Single and Double	Whenever you use a value with a decimal. Use Single for numbers approximately $\pm 10^{40}$; use Double for numbers larger than that range.
Currency	Accepts values ± 1 quadrillion with four decimal places
Date	Whenever variable is date or time
Object	Points to any object
Variant	Any of the other data types, including arrays. This is the default setting for all undefined/undeclared variables.
Constants	Value cannot change during program execution
Arrays	Blocks of stored data

You can declare variables of a specific type using the Dim statement; supply a name for the variable and the data type. The syntax is as follows:

Dim Variablename As DataType

TypeName function lets us check to see if a variable is of a particular type.
IsNumeric function determines if a variable contains a value that can be used as numeric.

Control Structures

The statements in VBA that control decision making and looping are control structures. VBA procedures can test conditions and then, depending on the results of that test, perform different operations. The decision structure supported by VBA are:

- If … Then
- If … Then … Else
- Select Case

If … Then

This is the control structure used to perform various actions on the basis of whether an expression evaluates to True or False. The simplest form of the expression is the ***If … End If***. A simple example would be:

```
If ActiveSheet.Name <> "Financial Quotes" Then
        MsgBox "You are on the incorrect sheet."
End If
```

Another form of the If statement uses the word ***Else*** to take an alternative if the statement evaluates False.

```
If ActiveSheet.Name <> "Financial Quotes" Then
        MsgBox "You are on the incorrect sheet."
Else
        ActiveSheet.Calculate
End If
```

You can add the ElseIf statement to test several conditions (this replaces several nested **If … Then** statements, making the code shorter and easier to read).

```
If ActiveSheet.Name = "Financial Quotes" Then
        ActiveSheet.Calculate
ElseIf ActiveSheet.Name = "Income" Then
```

```
        DialogSheets("EnterTransaction").Show
Else
        MsgBox "You are on the incorrect sheet."
End If
```

Select Case

The Select Case control structure is an alternative to the If … Then … Else statement for comparing the same expression to several different values. Use the Select Case control structure to take a variety of actions on the basis of the different values an expression or variable can have. You must define various cases when using a Select Case control structure. If the expression equals the case, then the subsequent code runs.

```
Sub DemoCase()

  Age = CInt(InputBox("Enter your Age:"))
  If Age = 0 Then Exit Sub
  Select Case Age
    Case 1
      MsgBox "You are one year old."
    Case 2 To 12
      MsgBox "You are a child."
    Case 12 To 19
      MsgBox "You are a teenager."
    Case 20 To 29
      MsgBox "You are a young adult."
    Case 30 To 39
      MsgBox "You are an adult."
    Case Else
      MsgBox "You are old."
  End Select

End Sub
```

Loops

Use *loops* to execute one or more lines of code repetitively. The loop structures supported by VBA are:

- For … Next
- For Each … Next
- Do … Loop

For … Next

Use this control structure to perform an action a specific number of times. The control structure takes two lines. The first line tells how many time to cycle through the code; the last line is simply the work *Next*.

```
For Ctr = 1 To Sheets Step 1
        MsgBox "Sheet" & Ctr & " is " & ThisWorkbook.Sheet(Ctr).Name & "."
Next
```

The counter increments by one each time the loop is processed unless a different increment is specified by the keyword *Step* followed by a step number.

Do … Loop

Use the Do … Loop control structures if you want to perform actions until (or while) a certain condition is True. The four forms of the Do … Loop are:

- *Do Until <Boolean expression>*
  …
  Loop

- *Do*
  …
  Loop Until <Boolean expression>

- *Do While <Boolean expression>*
  …
  Loop

- *Do*
  …
  Loop While <Boolean expression>

The following code displays a counter until the specified number is reached:

```
Dim Ctr As Integer
Dim CountTo As Double

CountTo = CDbl(InputBox("How high should I count?"))

Do Until Ctr = CountTo
        Ctr = Ctr + 1
        Applications.StatusBar = Ctr
Loop
```

For Each … Next

Unlike the normal For … Next loop, this control structure does not require you to define an upper limit or how many times the loop will run. Instead, you define an object variable and the collection you want to apply it to. Within the loop, the variable refers to a single item in the collection.

```
Sub MakeListOfSheets()
        Dim Sheet As Object

        Workbooks.Add (Worksheet)
        For Each Sheet In ThisWorkbook.Sheets
                [a1].Offset(Sheet.Index – 1).Value = Sheet.Name
        Next
End Sub
```

The syntax for the For Each … Next statement is

```
For Each element In group
        Statements
Next
```

To exit any of the control structures and procedures, just use an **Exit** statement. For example, you can use the **Exit For** statement to exit directly from a **For … Next** loop and the **Exit Do** statement to exit directly from a **Do** loop.

On-line VBA Help

Full on-line help is available for all objects, properties, and methods in Excel. By choosing **Contents** from the **Help** menu in Excel and then double-clicking **Microsoft Excel Visual Basic Reference**, and then double-clicking **Visual Basic Reference**

you can access help topics on all excel objects as well as on their associated properties and methods. You can also get a listing of all Excel objects by selecting Objects from the main Visual Basic Reference Contents screen. And to search for a particular object, you simply choose the Search button and then enter the name of the object.

MsgBox() Buttons Arguments

Constant	Value	Description
VbOKOnly	0	Displays only the OK button
VbOKCancel	1	Displays the OK and Cancel buttons
VbAbortRetryIgnore	2	Displays the Abort, Retry, and Ignore buttons
VbYesNoCancel	3	Displays the Yes, No, and Cancel buttons
VbYesNo	4	Displays the Yes and No buttons
VbRetryCancel	5	Displays the Retry and Cancel buttons
VbCritical	16	Displays the Critical Message icon
VbQuestion	32	Displays the Question mark icon
VbExclamation	48	Displays the Exclamation icon
VbInformation	64	Displays the Information icon
VbDefaultButton1	0	Makes the first button the default
VbDefaultButton2	256	Makes the second button the default
VbDefaultButton3	512	Makes the button the default
VbApplicationModal	0	No further processing occurs in Excel until the user closes the message box
VbSystemModal	4096	All applications are suspended until the user responds to the message box