

Programming Microsoft Excel

Objectives

- Explore the top-level objects in the Excel object model.
- Show how to work with data in cells on a worksheet.
- Explain how to save form data to a worksheet.
- Create a chart from VBA code.

The Microsoft Excel Object Model

Microsoft Excel's object model includes just about every feature normally available to you in Microsoft Excel. As object models go, it's sophisticated and fairly complex. Fortunately, to get started programming with Excel, you only need to understand how to work with a few objects. From there, you'll find that you can figure out what you need to know as you go along.

Figure 1 shows a simplified version of the Microsoft Excel object model. For the complete object model, see Online Help or go to the Microsoft web site, at <http://www.microsoft.com/OfficeDev/Articles/OMG/default.htm>.

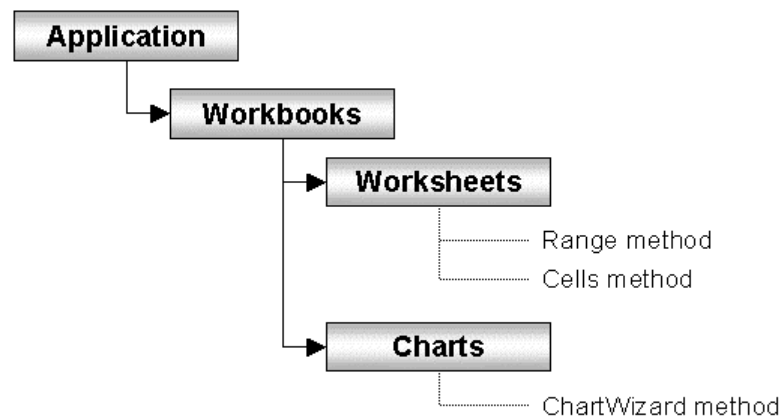


Figure 1. A portion of the Microsoft Excel object model.

TIP: If you want to explore Microsoft Excel's objects, properties, and methods, head to the Object Browser, and press **F1** for context-sensitive Help.

Microsoft Excel's Top-Level Objects

The Application Object

The Application object represents Microsoft Excel itself. It contains all of the other objects and collections in the object model and contains properties and methods of its own.

When you're programming in Microsoft Excel, you don't always need to refer to the Application object explicitly. If you're using a property or method that appears under the Globals class in the Object Browser, then you don't need to specify the Application object. However, if you're using a property or method that appears only under the Application class, and not also in the Globals class, then you do need to explicitly refer to the Application object.

For example, you don't need to use the Application object when you refer to a member of the Workbooks collection, because the Workbooks property is global. This code makes a workbook the active workbook in Microsoft Excel:

```
Workbooks(1).Activate
```

However, you do need to use it when you call a method of the Application object. The GetOpenFileName method of the Application object shows the File Open dialog box, and returns a string containing the name of the file chosen by the user:

```
Dim strFileName As String
Const conFilter = "All Files (*.*) *.*"

strFileName = Application.GetOpenFilename(conFilter)
```

The Workbook Object

The Workbook object represents a workbook already open in Microsoft Excel and can refer only to opened workbooks.

To use a Workbook object, refer to it in one of two ways:

- Use the Workbooks property of the Application object with an index value or the name of the Workbook:

```
' No need to say Application.Workbooks(1)
' because Application is assumed
Workbooks(1).Activate
or
Workbooks("vba.xls").Activate
```

- Use the `ActiveWorkbook` property of the `Application` object to return a reference to the active workbook. The active workbook is the one currently displayed in the Microsoft Excel user interface.

```
ActiveWorkbook.Save
```

The `ActiveWorkbook` property is convenient, but be careful when you use it. If more than one workbook is open, the active workbook may not be the one your code is expecting! Use the `Activate` method to make an open workbook the active workbook.

NOTE Microsoft Excel's collections are all one-based collections, rather than zero-based. In other words, the first object in a collection has an index of 1 rather than 0.

Some Common Methods and Properties

What kinds of things can you do with a `Workbook` object? Since it represents a file that's open in Microsoft Excel, you can do quite a lot with it. Here are just a few examples of some common properties and methods of a `Workbook` object.

You can open a workbook using the `Open` method of the `Workbooks` collection, and assign it to an object variable:

```
Dim wkb As Workbook
Set wkb = Workbooks.Open("vba.xls")
```

You can create a new workbook with the `Add` method, and save it with the `SaveAs` method:

```
Dim wkb As Workbook
Set wkb = Workbooks.Add
Set wkb = Workbooks.SaveAs("NewWorkbook.xls")
```

You can return the workbook's name, its path, or both:

```
With ActiveWorkbook
    Debug.Print .Name
    Debug.Print .Path
    Debug.Print .FullName
End With
```

The Worksheet Object

Next in line is the Worksheet object which belongs to the Worksheets collection of a Workbook object and represents a worksheet in a workbook.

Referring to a worksheet is the same as referring to a workbook. You can:

- Use the Worksheets property with an index value or the worksheet's name.

```
Debug.Print ActiveWorkbook.Worksheets(1).Name
```

- Use the ActiveSheet property to refer to the active worksheet.

```
ActiveWorkbook.ActiveSheet.Copy
```

The Copy method copies a worksheet to another workbook or to another position in the current workbook.

The Worksheet object is used most often when referring to another object. So let's move on to the objects that appear beneath it in the object model.

The Range Object

*See
basObjectModel*

The Range object is probably the most important member of the Microsoft Excel object model. It is the object that gives you access to the values within the cells of a worksheet and refers to a block of cells on a worksheet.

Ranges can be many different shapes and sizes and can include a single cell, or several rows and columns.

The Range object belongs to a Worksheet object; there is no Ranges collection. Use the Range property of the Worksheet object to refer to a Range object.

Referring to a Range

In order to refer to a range, you need to specify either a name or an address for the range. The following are just a few of the ways to refer to a range.

- To refer to a single cell, you can use the `Range("A1")` format:

```
Debug.Print Worksheets(1).Range("A1").Value
```

- To refer to a multi-cell range, you can use a format like `Range("A1:B5")`:

```
Worksheets(1).Range("A1:B5").Font.Bold = True
```

- To refer to a named range, you can just use the name of the range:

```
Debug.Print _  
    Worksheets(1).Range("ProductName").Rows.Count
```

<p>TIP: To create a named range in Microsoft Excel, select the cells you want to include in the range, then type a name for the range into the Name Box in the upper left-hand corner of Microsoft Excel. You can also create a named range programmatically, by using the <code>Add</code> method of the <code>Names</code> collection.</p>

The CurrentRegion Property

The `Range` object has a `CurrentRegion` property, which returns another `Range` object representing the contiguous data region around the active cell. The range returned by the `CurrentRegion` property is bounded by an empty row and an empty column. The `CurrentRegion` property is useful when you need to work with a specific set of data on a worksheet.

To use the `CurrentRegion` property, activate a cell within the data region by using the `Activate` method on the range that represents that cell.

The following example uses the `CurrentRegion` property to get information about the data region on a worksheet:

```
Sub DataRegion()  
    Dim rng As Range
```

```
' Activate cell A1.
ActiveSheet.Range("A1").Activate
' Get the data region around cell A1.
Set rng = _
    ActiveSheet.Range("A1").CurrentRegion
' Print information about the data region.
With rng
    Debug.Print .Address
    Debug.Print .Columns.Count
    Debug.Print .Rows.Count
    Debug.Print .End(xlDown).Address
End With
End Sub
```

The Cells Property

The Cells property allows you to refer to a specific cell within a range, using row and column coordinates. It helps you to navigate through the range in order to set or return cell values.

For example, suppose you want to get the value of the first cell in the ProductName range. Rather than figuring out the address for that cell, you can use the Cells property as follows:

```
Dim rng As Range
Set rng = Worksheets(1).Range("ProductName")
Debug.Print rng.Cells(1, 1).Value
```

The Cells property takes as arguments the row and column you're interested in within the range. In other words, Cells(1, 1) returns the cell in the upper leftmost corner of the range. Cells(2, 1) returns the cell in the first column and second row of the range, and so on. The Cells property evaluates rows and columns relative to the range, so it doesn't matter where the range is on the worksheet – Cells(1, 1) always returns the first cell.

The Selection Object

When the user selects a group of cells, you can return a range based on their selection. For example, the following procedure performs an autofill on the cells selected by the user and fills them with the days of the week.

To try this example, view the Schedule worksheet in vba.xls. Select a group of two or more cells, then click the AutoFill Weekdays toolbar button as shown in Figure 2.

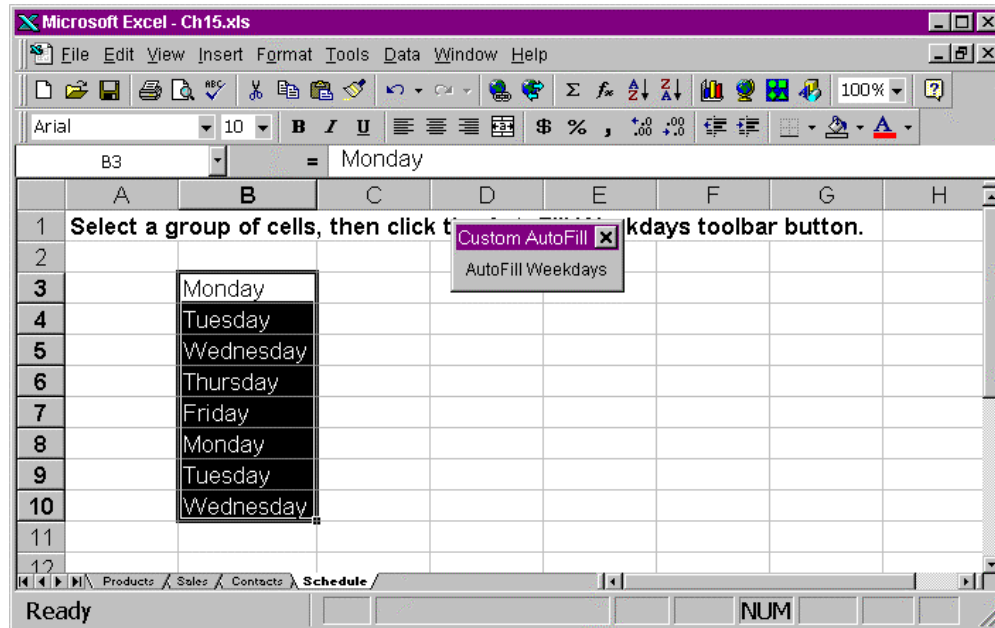


Figure 2. Clicking the toolbar button fills the selected cells with a repeating series of weekdays.

See **FillWeekdays**
in **basFill**

Here's the code that performs the autofill:


```
Sub FillWeekdays()  
    Dim rng As Range  
    Dim rngStart As Range  
  
    ' Return range defined by selection  
    Set rng = Selection  
    ' Check that user has selected  
    ' at least two cells.  
    If rng.Cells.Count < 2 Then  
        MsgBox "You must select at least two cells."  
        GoTo Exit_FillWeekdays  
    End If  
  
    ' If user selected more than one column,  
    ' use only first column.  
    If rng.Columns.Count > 1 Then  
        ' Redefine range as single-column.  
        Set rng = rng.Columns(1)  
    End If  
  
    ' Return first cell in that range.  
    Set rngStart = rng.Cells(1, 1)  
  
    ' Insert first value.  
    rngStart.Value = "Monday"  
    ' Autofill selection with weekdays.  
    rngStart.AutoFill rng, xlFillWeekdays  
  
Exit_FillWeekdays:  
    Exit Sub  
End Sub
```

The first thing this procedure does is to use the Selection property to return a range representing the user's selection. It then counts the number of cells in the range to make sure the user has selected more than one cell. If not, then the procedure exits.

Assuming the user has selected at least two cells, the procedure then checks to see how many columns the user has selected. If it's more than one, then it redefines the range to include only the first column.

Next, the procedure returns a range, *rngStart*, that represents the first cell in the selection, and gives that cell the seed value of "Monday". Finally, it calls the *AutoFill* method on *rngStart*, specifying the cells to be filled as the selection range, and that they are to be filled with weekdays.

Saving Form Data in Microsoft Excel

When you're working with forms, you may find that you want to save the data the user enters into the form. There are many different ways to do this, but one simple way is just to write the data to a worksheet. Using a form for data entry allows you to perform validation on the data before it's submitted to the worksheet.

See *frmSaveData*

First, take a look at *frmSaveData*. Run the form, enter some data into the text boxes, and click OK. The data is saved to the first row in the Microsoft Excel worksheet beneath the column headings (see Figure 3). The next set of data entered is saved to the next row of the worksheet, and so on.

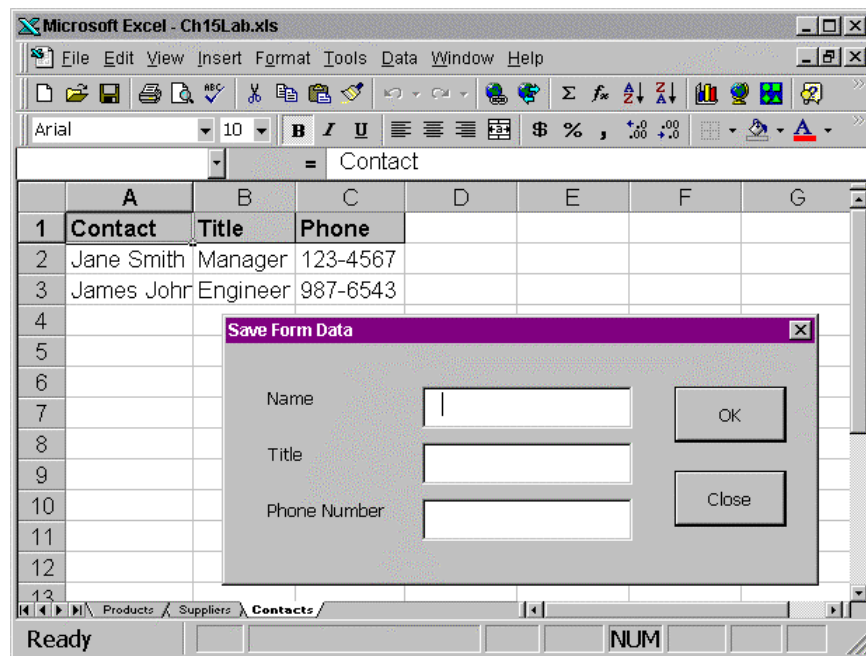


Figure 3. Saving data from a form to a Microsoft Excel worksheet.

When the *UserForm_Initialize* event occurs, the *Contacts* worksheet becomes the active worksheet, and cell A1 on the *Contacts* worksheet becomes the active cell. Run the *UserForm_Initialize* event procedure often using the following code, to make sure you are saving the data in the proper place:

```
Private Sub UserForm_Initialize()  
    Worksheets("Contacts").Activate  
    ActiveSheet.Range("A1").Activate  
End Sub
```

The Click event procedure for the OK button contains the code that inserts the new data into the worksheet:

```
Private Sub cmdOK_Click()  
    Dim intCount As Integer  
  
    ' Add data to next empty row.  
    intCount = ActiveCell.CurrentRegion.Rows.Count + 1  
    ActiveSheet.Cells(intCount, 1).Value = txtName.Value  
    ActiveSheet.Cells(intCount, 2).Value = _  
        txtTitle.Value  
    ActiveSheet.Cells(intCount, 3).Value = _  
        txtPhone.Value  
  
    ' Clear form.  
    txtName.Value = ""  
    txtTitle.Value = ""  
    txtPhone.Value = ""  
  
    ' Set focus to txtName.  
    txtName.SetFocus  
End Sub
```

What's this code doing? First, it uses the `CurrentRegion` property to return a range containing all of the contiguous data around the active cell. The range returned by the `CurrentRegion` property is bounded by at least one empty row and one empty column. By adding 1 to the number of rows in the current region, you can be sure you're working with an empty row. You can use the resulting row number as an argument to the `Cells` property.

Next, the procedure assigns the value of the `Value` property for each text box on the form to the `Value` property for a cell in the active worksheet. Finally, it clears the data from the form and sets the focus to the first text box, so the user can enter another record.

Working with the Chart Object

You can use the Chart object and its associated objects and collections to do just about anything you want to with a chart in Microsoft Excel. The chart portion of the object model is complex, but fortunately, it's relatively easy to create a chart from VBA.

Charts vs. ChartObjects

No, we're not being redundant. Believe it or not, Microsoft Excel has two types of charts in its object model. The Chart object exists on its own sheet, while the ChartObject object is embedded within a worksheet.

These two objects hold different places in the object model. The Chart object is a member of the workbook's Charts collection, and the ChartObject object is a member of a worksheet's ChartObjects collection (see Figure 4).

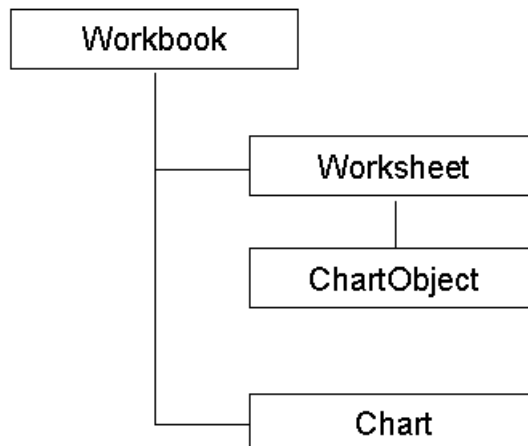


Figure 4. Charts and ChartObjects occupy different positions in the object model.

The primary differences between the Chart and ChartObject are that Microsoft Excel provides properties for the ChartObject's frame within the worksheet. For this chapter, we'll just focus on Chart objects.

The ChartWizard Method

The object model includes all the methods and properties you need to create a chart from scratch. But the ChartWizard method is a faster and easier way to

create charts. This method emulates the behavior of the Microsoft Excel ChartWizard, which comes up when you choose the **Insert|Chart** menu item.

Here's the syntax for the ChartWizard method:

```
Expression.ChartWizard(Source, Gallery, Format, PlotBy,  
CategoryLabels, SeriesLabels, HasLegend, Title,  
CategoryTitle, ValueTitle, ExtraTitle)
```

There are quite a few arguments, but don't be daunted. All of the arguments are optional, so let's take a look at a few of the important ones:

- **Expression:** A **Chart** object.
- **Source:** The range containing the data you want to chart. You can chart data from non-continuous ranges by first using the Union method to join multiple ranges together.
- **Gallery:** A constant that specifies the chart type. Here you can pick from lots of cool-looking charts, such as three-dimensional bar or pie charts. You can find the constants in Help or in the Object Browser
- **PlotBy:** A constant that specifies whether data is plotted by rows or by columns.

Creating a Chart

See *CreateChart* in
basBuildChart

Now let's try creating a chart using the ChartWizard method. Open the `basBuildChart` module in `vba.xls` and run the `CreateChart` procedure to practice this exercise.

The chart we're creating plots sales data from the Sales worksheet. To make it easy to identify, we've created a named range called `TotalSales` that encompasses all the data on the worksheet. The procedure will use this range as the source for the chart. The following table shows the data in the range, as it appears in Microsoft Excel:

Employee	TotalSales
Davolio, Nancy	\$202,143.71
Fuller, Andrew	\$177,749.26
Leverling, Janet	\$213,051.30
Peacock, Margaret	\$250,187.45
Buchanan, Steven	\$75,567.75
Suyama, Michael	\$78,198.10
King, Robert	\$141,295.99
Callahan, Laura	\$133,301.03
Dodsworth, Anne	\$82,964.00

Here's the `CreateChart` procedure. It adds a new chart to the Charts collection, then uses the `ChartWizard` method to provide all the data needed to create the chart we want. Finally, it adds some additional formatting to the chart.

```
Sub CreateChart()  
    Dim wbk As Workbook  
    Dim wks As Worksheet  
    Dim cht As Chart  
    Dim rng As Range  
    Dim intCount As Integer  
  
    ' Return object variables.  
    Set wbk = Workbooks("vba.xls")  
    Set wks = wbk.Worksheets("Sales")  
    Set rng = wks.Range("TotalSales")  
  
    ' Add chart.  
    Set cht = wbk.Charts.Add(After:=wks)  
    ' Use ChartWizard method to create chart.  
    ' Source is TotalSales range from worksheet.  
    ' Other arguments control chart appearance.  
    With cht  
        .ChartWizard Source:=rng, _  
        Gallery:=xl3DColumn, Format:=4, _  
        PlotBy:=xlRows, CategoryLabels:=1, _  
        SeriesLabels:=1, HasLegend:=2, _  
        Title:=wks.Name  
  
        ' Apply some formatting  
        .Name = wks.Name & " Chart"  
        With .ChartGroups(1)  
            .GapWidth = 20  
            .VaryByCategories = True  
        End With  
        With .ChartTitle  
            .Font.Size = 16  
            .Shadow = True  
            .Border.LineStyle = xlSolid  
        End With  
    End With  
End Sub
```


We end up with a chart that looks like the one shown in Figure 5:

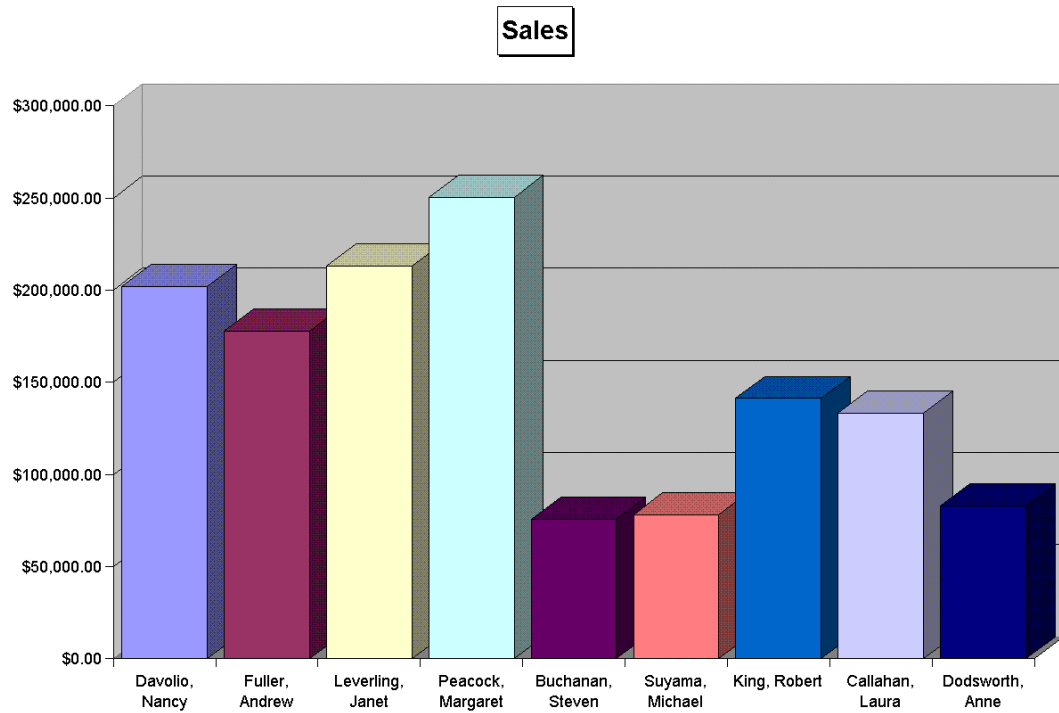


Figure 5. The Sales chart created using the ChartWizard method.

Summary

- Some of the key objects in the Microsoft Excel object model are: Application, Workbook, Worksheet, Range, and Chart.
- The Range object is the object you use to work with sets of cells. You can refer to ranges using an address such as A1:B5, or by using a name if you've created a named range.
- The Cells property of the Range object enables you to set or retrieve a value from a cell.
- If you want to use a form for data entry, you can save form data to a worksheet.
- The Chart object represents a chart, and is a member of the Charts collection of the Workbook object.
- You can generate a chart from VBA by using the ChartWizard method.