Learning to Program the Excel Object Model Using VBA

2nd Edition

Writing

Excel Macros

with VBA

O'REILLY®

Steven Roman

SECOND EDITION

Writing Excel Macros with VBA

Steven Roman

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 17

The Workbook Object

In this chapter, we discuss the Workbook object and the Workbooks collection. Figure 17-1 shows the portion of the Excel object model that relates directly to workbooks.

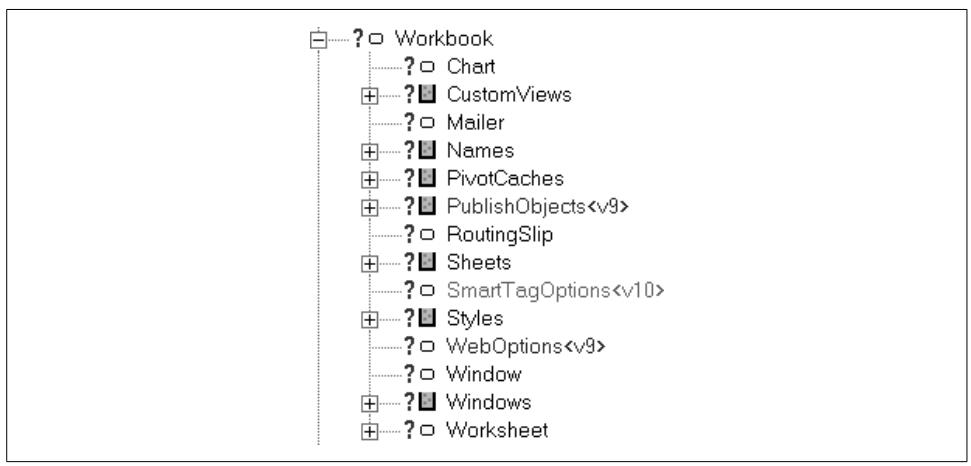


Figure 17-1. The Workbook object

The Workbooks Collection

The Application object has a Workbooks property that returns a Workbooks collection, which contains all of the Workbook objects for the currently open instance of Excel. For instance, the following code displays the number of open workbooks:

```
Dim wbs As Workbooks
Set wbs = Application.Workbooks
MsgBox wbs.Count
```

Let us look at a few of the properties and methods of the Workbooks collection.

Add Method

The Add method creates a new workbook, which is then added to the Workbooks collection. The new workbook becomes the active workbook. The syntax is:

```
WorkbooksObject.Add(Template)
```

where the optional *Template* parameter determines how the new workbook is created. If this argument is a string specifying the name of an existing Excel template file, the new workbook is created with that file as a template.

As you may know, a template is an Excel workbook that may contain content (such as row and column labels), formatting, and macros and other customizations (menus and toolbars, for instance). When you base a new workbook on a template, the new workbook receives the content, formatting, and customization from the template.

The *Template* argument can also be one of the following constants:

```
Enum XlWBATemplate
     xlWBATWorksheet = -4167
     xlWBATChart = -4109
     xlWBATExcel4MacroSheet = 3
     xlWBATExcel4IntlMacroSheet = 4
End Enum
```

In this case, the new workbook will contain a single sheet of the specified type. If the *Template* argument is omitted, Excel will create a new workbook with the number of blank sheets set by the Application object's SheetsInNewWorkbook property.

Close Method

The Close method closes all open workbooks. The syntax is simply:

```
WorksbooksObject.Close
```

Count Property

Most collection objects have a Count property, and the Workbooks collection is no exception. This property simply returns the number of currently open workbooks.

Item Property

The Item property returns a particular workbook in the Workbooks collection. For instance:

```
Workbooks.Item(1)
```

returns the Workbook object associated with the first workbook in the Workbooks collection. Since the Item property is the default property, we can also write this as:

```
Workbooks(1)
```

Note that we cannot rely on the fact that a certain workbook will have a certain index. (This applies to all collections.) Thus, to refer to a particular workbook, you should always use its name, as in:

```
Workbooks("Book1.xls")
```

It is important to note that if a user creates a new workbook named, say, Book2, using the New menu item on the File menu, then we may refer to this workbook in code by writing:

```
Workbooks("Book2")
```

but the code:

```
Workbooks("Book2.xls")
```

will generate an error (subscript out of range) until the workbook is actually saved to disk.

Open Method

This method opens an existing workbook. The rather complex syntax is:

```
WorkbooksObject.Open(FileName, UpdateLinks, ReadOnly, _
   Format, Password, WriteResPassword, IgnoreReadOnlyRecommended, _
   Origin, Delimiter, Editable, Notify, Converter, AddToMRU)
```

Most of these parameters are rarely used (several of them relate to opening text files, for instance). We discuss the most commonly used parameters and refer the reader to the help files for more information. Note that all of the parameters are optional except *FileName*.

FileName is the file name of the workbook to be opened. To open the workbook in read-only mode, set the *ReadOnly* parameter to `True`.

If a password is required to open the workbook, the *Password* parameter should be set to this password. If a password is required but you do not specify the password, Excel will ask for it.

The *AddToMru* parameter should be set to `True` to add this workbook to the list of recently used files. The default value is `False`.

OpenText Method

This method will load a text file as a new workbook. The method will parse the text data and place it in a single worksheet. The rather complex syntax is:

```
WorkbooksObject.OpenText(Filename, Origin, StartRow, _
   DataType, TextQualifier, ConsecutiveDelimiter, Tab, _
   Semicolon, Comma, Space, Other, OtherChar, FieldInfo)
```

Note first that all of the parameters to this method are optional except the *FileName* parameter.

The *Filename* parameter specifies the filename of the text file to be opened.

The *Origin* parameter specifies the origin of the text file and can be one of the following XlPlatform constants:

```
Enum XlPlatform
     xlMacintosh = 1
     xlWindows = 2
     xlMSDOS = 3
End Enum
```

Note that the xlWindows value specifies an ANSI text file, whereas the xlMSDOS constant specifies an ASCII file. If this argument is omitted, the current setting of the File Origin option in the Text Import Wizard will be used.

The *StartRow* parameter specifies the row number at which to start parsing text from the text file. The default value is 1.

The optional *DataType* parameter specifies the format of the text in the file and can be one of the following XlTextParsingType constants:

```
Enum XlTextParsingType
     xlDelimited = 1         ' Default
     xlFixedWidth = 2
End Enum
```

The *TextQualifier* parameter is the text qualifier. It can be one of the following XlTextQualifier constants:

```
Enum XlTextQualifier
     xlTextQualifierNone = -4142
     xlTextQualifierDoubleQuote = 1   ' Default
     xlTextQualifierSingleQuote = 2
End Enum
```

The *ConsecutiveDelimiter* parameter should be set to True for Excel to consider consecutive delimiters as one delimiter. The default value is False.

There are several parameters that require that *DataType* be xlDelimited. When any one of these parameters is set to True, it indicates that Excel should use the corresponding character as the text delimiter. They are described here (all default values are False):

Tab
: Set to True to use the tab character as the delimiter.

Semicolon
: Set to True to use a semicolon as the delimiter.

Comma
: Set to True to use a comma as the delimiter.

Space
: Set to True to use a space as the delimiter.

Other
: Set to True to use a character that is specified by the *OtherChar* argument as the delimiter.

When *Other* is True, *OtherChar* specifies the delimiter character. If *OtherChar* contains more than one character, only the first character is used.

The *FieldInfo* parameter is an array containing parse information for the individual source columns. The interpretation of *FieldInfo* depends on the value of *DataType*.

When *DataType* is xlDelimited, the *FieldInfo* argument should be an array whose size is the same as or smaller than the number of columns of converted data. The first element of a two-element array is the column number (starting with the number 1), and the second element is one of the following numbers that specifies how the column is parsed:

Value	Description
1	General
2	Text
3	MDY date
4	DMY date
5	YMD date
6	MYD date
7	DYM date
8	YDM date
9	Skip the column

If a two-element array for a given column is missing, then the column is parsed with the General setting. For instance, the following value for *FieldInfo* causes the first column to be parsed as text and the third column to be skipped:

```
Array(Array(1, 2), Array(3, 9))
```

All other columns will be parsed as general data.

To illustrate, consider a text file with the following contents:

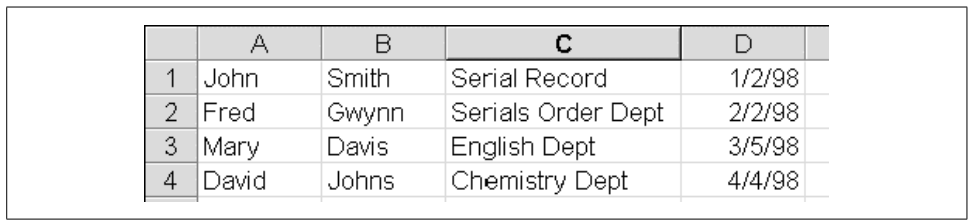
```
"John","Smith","Serial Record",1/2/98
"Fred","Gwynn","Serials Order Dept",2/2/98
"Mary","Davis","English Dept",3/5/98
"David","Johns","Chemistry Dept",4/4/98
```

The code:

```
Workbooks.OpenText _
  FileName:="d:\excel\temp.txt", _
  Origin:=xlMSDOS, _
```

```
    StartRow:=1, _
    DataType:=xlDelimited, _
    TextQualifier:=xlTextQualifierDoubleQuote, _
    ConsecutiveDelimiter:=True, _
    Comma:=True, _
    FieldInfo:=Array(Array(1, 2), _
       Array(2, 2), Array(3, 2), Array(4, 6))
```

produces the worksheet shown in Figure 17-2. Note that the cells in column D are formatted as dates.



	A	B	C	D
1	John	Smith	Serial Record	1/2/98
2	Fred	Gwynn	Serials Order Dept	2/2/98
3	Mary	Davis	English Dept	3/5/98
4	David	Johns	Chemistry Dept	4/4/98

Figure 17-2. A comma-delimited text file opened in Excel

On the other hand, if *DataType* is `xlFixedWidth`, the first element of each two-element array specifies the starting character position in the column (0 being the first character) and the second element specifies the parse option (1–9) for the resulting column, as described earlier.

To illustrate, consider the text file whose contents are as follows:

```
0-125-689
2-523-489
3-424-664
4-125-160
```

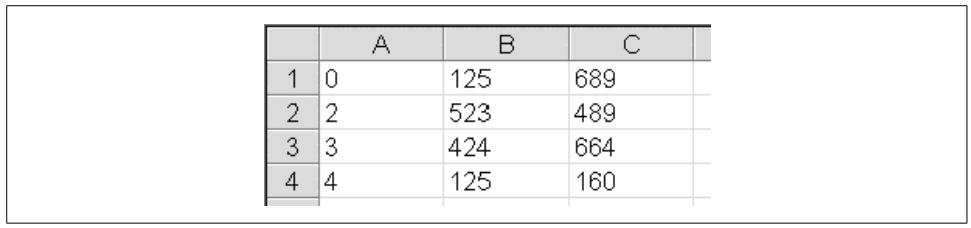
The code:

```
Workbooks.OpenText _
  FileName:="d:\excel\temp.txt", _
  Origin:=xlMSDOS, _
  StartRow:=1, _
  DataType:=xlFixedWidth, _
  FieldInfo:=Array(Array(0, 2), _
    Array(1, 9), Array(2, 2), Array(5, 9), _
    Array(6, 2))
```

produces the worksheet in Figure 17-3. (Note how we included arrays to skip the hyphens.)

Finally, it is important to observe that the text file is opened in Excel, but not converted to an Excel workbook file. To do so, we can invoke the SaveAs method, as in:

```
Application.ActiveSheet.SaveAs _
   FileName:="d:\excel\temp.xls", _
   FileFormat:=xlWorkbookNormal
```



	A	B	C
1	0	125	689
2	2	523	489
3	3	424	664
4	4	125	160

Figure 17-3. A fixed-width text file opened in Excel

The Workbook Object

A Workbook object represents an open Excel workbook. As we have discussed, Workbook objects are stored in a Workbooks collection.

The Workbook object has a total of 103 properties and methods, as shown in Table 17-1.

Table 17-1. Members of the Workbook object

_CodeName	FullName	RefreshAll
_PrintOut<v9>	FullNameURLEncoded<v10>	RejectAllChanges
_Protect<v10>	HasMailer	ReloadAs<v9>
_ReadOnlyRecommended<v10>	HasPassword	RemovePersonalInformation<v10>
_SaveAs<v10>	HasRoutingSlip	RemoveUser
AcceptAllChanges	HighlightChangesOnScreen	Reply
AcceptLabelsInFormulas	HighlightChangesOptions	ReplyAll
Activate	HTMLProject<v9>	ReplyWithChanges<v10>
ActiveChart	IsAddin	ResetColors
ActiveSheet	IsInplace	RevisionNumber
AddToFavorites	KeepChangeHistory	Route
Application	Keywords	Routed
Author	LinkInfo	RoutingSlip
AutoUpdateFrequency	LinkSources	RunAutoMacros
AutoUpdateSaveChanges	ListChangesOnNewSheet	Save
BreakLink<v10>	Mailer	SaveAs
BuiltinDocumentProperties	MergeWorkbook	SaveCopyAs
CalculationVersion<v9>	Modules	Saved
CanCheckIn<v10>	MultiUserEditing	SaveLinkValues
ChangeFileAccess	Name	sblt<v9>

Table 17-1. Members of the Workbook object

ChangeHistoryDuration	Names	SendForReview<v10>
ChangeLink	NewWindow	SendMail
Charts	OnSave	SendMailer
CheckIn<v10>	OnSheetActivate	SetLinkOnData
Close	OnSheetDeactivate	SetPasswordEncryptionOptions<v10>
CodeName	OpenLinks	Sheets
Colors	Parent	ShowConflictHistory
CommandBars	Password<v10>	ShowPivotTableFieldList<v10>
Comments	PasswordEncryptionAlgorithm<v10>	SmartTagOptions<v10>
ConflictResolution	PasswordEncryptionFileProperties<v10>	Styles
Container	PasswordEncryptionKeyLength<v10>	Subject
CreateBackup	PasswordEncryptionProvider<v10>	TemplateRemoveExtData
Creator	Path	Title
CustomDocumentProperties	PersonalViewListSettings	Unprotect
CustomViews	PersonalViewPrintSettings	UnprotectSharing
Date1904	PivotCaches	UpdateFromFile
DeleteNumberFormat	PivotTableWizard	UpdateLink
DialogSheets	Post	UpdateLinks<v10>
DisplayDrawingObjects	PrecisionAsDisplayed	UpdateRemoteReferences
Dummy16<v10>	PrintOut	UserControl
Dummy17<v10>	PrintPreview	UserStatus
EnableAutoRecover<v10>	Protect	VBASigned<v9>
EndReview<v10>	ProtectSharing	VBProject
EnvelopeVisible<v9>	ProtectStructure	WebOptions<v9>
Excel4IntlMacroSheets	ProtectWindows	WebPagePreview<v9>
Excel4MacroSheets	PublishObjects<v9>	Windows
ExclusiveAccess	PurgeChangeHistoryNow	Worksheets
FileFormat	ReadOnly	WritePassword<v10>
FollowHyperlink	ReadOnlyRecommended	WriteReserved
ForwardMailer	RecheckSmartTags<v10>	WriteReservedBy

Several of the members listed in Table 17-1 exist solely to return the children of the Workbook object. The children are shown in Figure 17-4.

Table 17-2 gives the members of the Workbook object that return children.

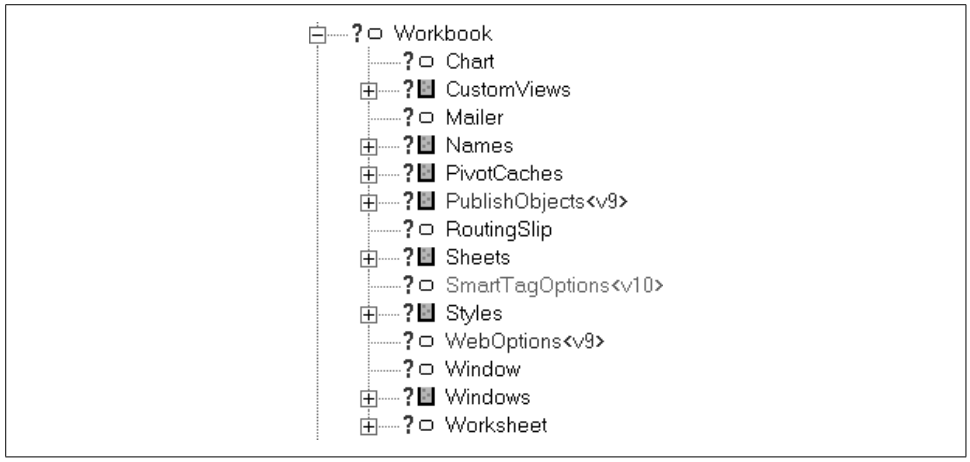


Figure 17-4. Children of the Workbook object

Table 17-2. Members of Workbook that return children

Name	ReturnType
ActiveChart	Chart
Application	Application
Charts	Sheets
CustomViews	CustomViews
DialogSheets	Sheets
Excel4IntlMacroSheets	Sheets
Excel4MacroSheets	Sheets
Mailer	Mailer
Modules	Sheets
Names	Names
NewWindow	Window
PivotCaches	PivotCaches
PublishObjects	PublishObjects
RoutingSlip	RoutingSlip
Sheets	Sheets
SmartTagOptions	SmartTagOptions
Styles	Styles
WebOptions	WebOptions
Windows	Windows
Worksheets	Sheets

There are a few items worth noting about Table 17-2. First, the ActiveSheet property may return either a Chart object or a Worksheet object, depending upon what type of object is currently active.

Second, the Charts, Sheets, and Worksheets properties all return a (different) Sheets collection. In particular, the Charts object returns the Sheets collection that contains all of the chart sheets in the workbook. (This does not include charts that are embedded in worksheets.) The Worksheets property returns the Sheets collection of all worksheets in the workbook. Finally, the Sheets property returns the Sheets collection of all worksheets and chart sheets. This is a relatively rare example of a collection that contains objects of more than one type. Note that there is no Sheet object in the Excel object model.

Let us look at a few of the more commonly used members from Table 17-1.

Activate Method

This method activates the workbook. The syntax is straightforward, as in:

```
Workbooks("MyWorkBook").Activate
```

Note that Workbooks is global, so we do not need to qualify it with the `Application` keyword.

Close Method

The Close method closes the workbook. Its syntax is:

```
WorkbookObject.Close(SaveChanges, FileName, RouteWorkbook)
```

Note that the Close method of the Workbook object has three parameters, unlike the Close method of the Workbooks object, which has none.

The optional *SaveChanges* parameter is used to save changes to the workbook before closing. In particular, if there are no changes to the workbook, the argument is ignored. It is also ignored if the workbook appears in other open windows. On the other hand, if there are changes to the workbook and it does not appear in any other open windows, the argument takes effect.

In this case, if *SaveChanges* is `True`, the changes are saved. If there is not yet a filename associated with the workbook (that is, if it has not been previously saved), then the name given in *FileName* is used. If *FileName* is also omitted, Excel will prompt the user for a filename. If *SaveChanges* is `False`, changes are not saved. Finally, if the *SaveChanges* argument is omitted, Excel will display a dialog box asking whether the changes should be saved. In short, this method behaves as you would hope.

The optional *RouteWorkbook* refers to routing issues; we refer the interested reader to the Excel VBA help file for more information.

It is important to note that the Close method checks the Saved property of the workbook to determine whether or not to prompt the user to save changes. If we set the Saved property to `True`, then the Close method will simply close the workbook with no warning and without saving any unsaved changes.

DisplayDrawingObjects Property

This property returns or sets a value indicating how shapes are displayed. It can be one of the following `XlDisplayShapes` constants:

```
Enum XlDisplayShapes
     XlDisplayShapes = -4104
     xlPlaceholders = 2
     xlHide = 3
End Enum
```

FileFormat Property (Read-Only Long)

This property returns the file format or type of the workbook. It can be one of the following `XlFileFormat` constants:

```
Enum XlFileFormat
    xlAddIn = 18
    xlCSV = 6
    xlCSVMac = 22
    xlCSVMSDOS = 24
    xlCSVWindows = 23
    xlCurrentPlatformText = -4158
    xlDBF2 = 7
    xlDBF3 = 8
    xlDBF4 = 11
    xlDIF = 9
    xlExcel2 = 16
    xlExcel2FarEast = 27
    xlExcel3 = 29
    xlExcel4 = 33
    xlExcel4Workbook = 35
    xlExcel5 = 39
    xlExcel7 = 39
    xlExcel9795 = 43
    xlHtml = 44
    xlIntlAddIn = 26
    xlIntlMacro = 25
    xlSYLK = 2
    xlTemplate = 17
    xlTextMac = 19
    xlTextMSDOS = 21
    xlTextPrinter = 36
    xlTextWindows = 20
    xlUnicodeText = 42
    xlWebArchive = 45
    xlWJ2WD1 = 14
```

```
    xlWJ3 = 40
    xlWJ3FJ3 = 41
    xlWK1 = 5
    xlWK1ALL = 31
    xlWK1FMT = 30
    xlWK3 = 15
    xlWK3FM3 = 32
    xlWK4 = 38
    xlWKS = 4
    xlWorkbookNormal = -4143
    xlWorks2FarEast = 28
    xlWQ1 = 34
    xlXMLSpreadsheet = 46
End Enum
```

Name, FullName, and Path Properties

The Name property returns the name of the workbook, the Path property returns the path to the workbook file, and FullName returns the fully qualified (path and filename) of the workbook file. All of these properties are read-only.

Note that using the Path property without a qualifier is equivalent to:

```
Application.Path
```

and thus returns the path to Excel itself (rather than to a workbook).

HasPassword Property (Read-Only Boolean)

This read-only property is `True` if the workbook has password protection. Note that a password can be assigned as one of the parameters to the SaveAs method.

PrecisionAsDisplayed Property (R/W Boolean)

When this property is `True`, calculations in the workbook will be done using only the precision of the numbers as they are displayed, rather than as they are stored. Its default value is `False`; calculations are based on the values of numbers as they are stored.

PrintOut Method

The PrintOut method prints an entire workbook. (This method applies to a host of other objects as well, such as Range, Worksheet, and Chart.) The syntax is:

```
WorkbookObject.PrintOut(From, To, Copies, _
   Preview, ActivePrinter, PrintToFile, Collate)
```

Note that all of the parameters to this method are optional.

The *From* parameter specifies the page number of the first page to print, and the *To* parameter specifies the last page to print. If omitted, the entire object (range, worksheet, etc.) is printed.

The *Copies* parameter specifies the number of copies to print. The default is 1.

Set *Preview* to True to invoke print preview rather than printing immediately. The default is False.

ActivePrinter sets the name of the active printer. On the other hand, setting *PrintToFile* to True causes Excel to print to a file. Excel will prompt the user for the name of the output file. (Unfortunately, there is no way to specify the name of the output file in code.)

The *Collate* parameter should be set to True to collate multiple multipage copies.

PrintPreview Method

This method invokes Excel's print preview feature. Its syntax is:

```
WorkbookObject.PrintPreview
```

Note that the PrintPreview method applies to the same set of objects as the PrintOut method.

Protect Method

This method protects a workbook so that it cannot be modified. Its syntax is:

```
WorkbookObject.Protect(Password, Structure, Windows)
```

The method also applies to charts and worksheets, with a different syntax.

The optional *Password* parameter specifies a password (as a case-sensitive string). If this argument is omitted, the workbook will not require a password to unprotect it.

Set the optional *Structure* parameter to True to protect the structure of the workbook—that is, the relative position of the sheets in the workbook. The default value is False.

Set the optional *Windows* parameter to True to protect the workbook windows. The default is False.

ReadOnly Property (Read-Only Boolean)

This property is True if the workbook has been opened as read-only.

RefreshAll Method

This method refreshes all external data ranges and pivot tables in the workbook. The syntax is:

```
WorkbookObject.RefreshAll
```

Save Method

This method simply saves any changes to the workbook. Its syntax is:

```
WorkbookObject.Save
```

SaveAs Method

This method saves changes to a workbook in the specified file. The syntax is:

```
expression.SaveAs(Filename, FileFormat, Password, WriteResPassword, _
   ReadOnlyRecommended, CreateBackup, AccessMode, ConflictResolution, _
   AddToMru, TextCodePage, TextVisualLayout)
```

The *Filename* parameter specifies the filename to use for the newly saved disk file. If a path is not included, Excel will use the current folder.

The *FileFormat* parameter specifies the file format to use when saving the file. Its value is one of the `XlFileFormat` constants described in our discussion of the FileFormat property.

The *Password* parameter specifies the password to use when saving the file and can be set to any case-sensitive string of up to 15 characters.

The *WriteResPassword* is a string that specifies the write-reservation password for this file. If a file is saved with a write-reservation password and this password is not supplied when the file is next opened, the file will be opened as read-only.

We can set the *ReadOnlyRecommended* parameter to `True` to display a message when the file is opened, recommending that the file be opened as read-only.

Set the *CreateBackup* parameter to `True` to create a backup file.

The *AccessMode* and *ConflictResolution* parameters refer to sharing issues. We refer the interested reader to the Excel VBA help file for details.

Set the *AddToMru* parameter to `True` to add the workbook to the list of recently used files. The default value is `False`.

The remaining parameters are not used in the U.S. English version of Excel.

SaveCopyAs Method

This method saves a copy of the workbook to a file but does not modify the open workbook itself. The syntax is:

```
WorkbookObject.SaveCopyAs(Filename)
```

where *Filename* specifies the filename for the copy of the original file.

Saved Property (R/W Boolean)

This property is `True` if no changes have been made to the specified workbook since it was last saved. Note that this property is read/write, which means we can set the property to `True` even if the workbook has been changed since it was last saved. As discussed earlier, we can set this property to `True`, then close a modified workbook without being prompted to save the current changes.

Children of the Workbook Object

Figure 17-5 shows the children of the *Workbook* object. (This is a repeat of Figure 17-4.)

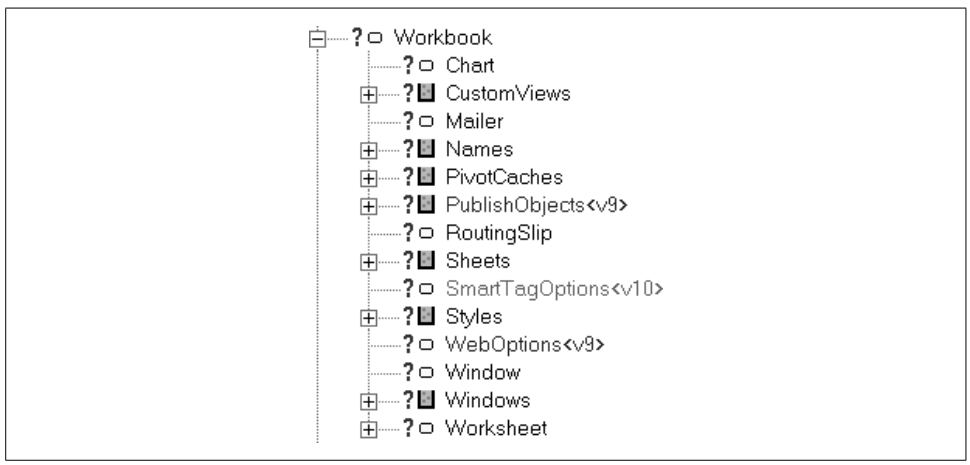


Figure 17-5. Children of the Workbook object

Let us take a quick look at some of these children. (We will discuss the Window, Worksheet, and WorkbookEvents objects later in the book.)

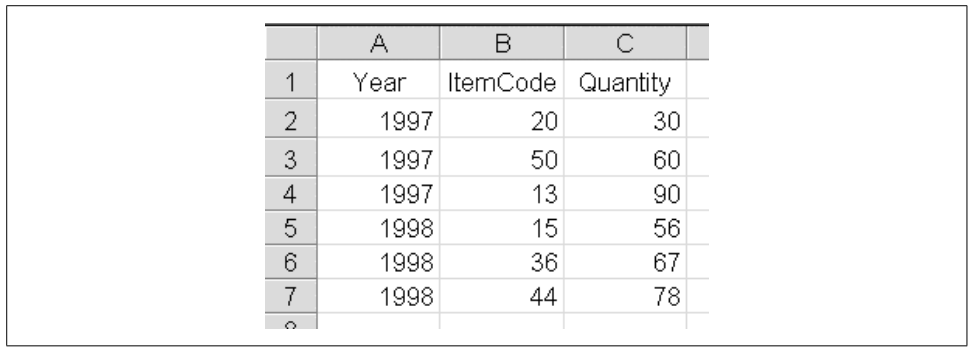
The CustomView Object

The CustomViews property returns the CustomViews collection. Each CustomView object in this collection represents a custom view of the workbook. CustomView objects are pretty straightforward, so we will just consider an example. Look at the sheet shown in Figure 17-6.

Now suppose we use the Autofilter command to filter on the year, as shown in Figure 17-7.

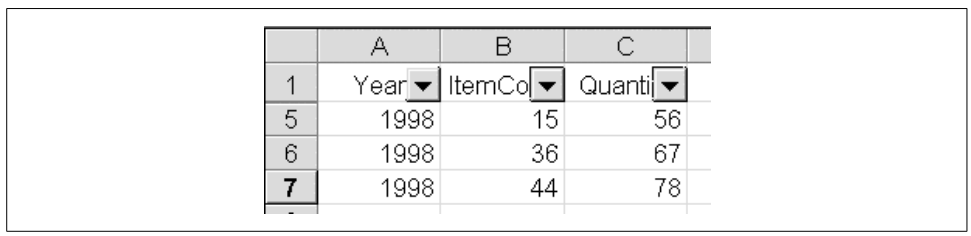
The following code will give this custom view the name *View1998*:

```
ThisWorkbook.CustomViews.Add "View1998"
```



	A	B	C
1	Year	ItemCode	Quantity
2	1997	20	30
3	1997	50	60
4	1997	13	90
5	1998	15	56
6	1998	36	67
7	1998	44	78

Figure 17-6. Example of the CustomView object



	A	B	C
1	Year	ItemCo	Quanti
5	1998	15	56
6	1998	36	67
7	1998	44	78

Figure 17-7. A filtered view

Now we can display this view at any time with the code:

```
ThisWorkbook.CustomViews!View1998.Show
```

or:

```
strView = "View1998"
ActiveWorkbook.CustomViews(strView).Show
```

The Names Collection

As with the Application object, the Workbook object has a Names property that returns a Names collection. This collection represents the Name objects associated with the workbook. For details on Name objects, see Chapter 16, *The Application Object*.

The Sheets Collection

The Sheets property returns a Sheets collection that contains a Worksheet object for each worksheet and a Chart object for each chartsheet in the workbook. We will discuss Worksheet objects and Chart objects later in the book.

The Styles Collection and the Style Object

A Style object represents a set of formatting options for a range. Each workbook has a Styles collection containing all Style objects for the workbook.

To apply a style to a range, we simply write:

```
RangeObject.Style = StyleName
```

where *StyleName* is the name of a style.

To create a Style object, use the Add method, whose syntax is:

```
WorkbookObject.Add(Name, BasedOn)
```

Note that the Add method returns the newly created Style object.

The *Name* parameter specifies the name of the style, and the optional *BasedOn* parameter specifies a Range object that refers to a cell whose style is used as a basis for the new style. If this argument is omitted, the newly created style is based on the Normal style.

Note that, according to the documentation, if a style with the specified name already exists, the Add method will redefine the existing style based on the cell specified in *BasedOn*. (However, on my system, Excel issues an error message instead, so you should check this carefully.)

The properties of the Style object reflect the various formatting features, such as font name, font size, number format, alignment, and so on. There are also several built-in styles, such as Normal, Currency, and Percent. These built-in styles can be found in the Style name box of the Style dialog box (under the Format menu).

To illustrate, the following code creates a style and then applies it to an arbitrary range of the current worksheet:

```
Dim st As Style
' Delete style if it exists
For Each st In ActiveWorkbook.Styles
  If st.Name = "Bordered" Then st.Delete
Next
' Create style
With ActiveWorkbook.Styles.Add(Name:="Bordered")
  .Borders(xlTop).LineStyle = xlDouble
  .Borders(xlBottom).LineStyle = xlDouble
  .Borders(xlLeft).LineStyle = xlDouble
  .Borders(xlRight).LineStyle = xlDouble
  .Font.Bold = True
  .Font.Name = "arial"
  .Font.Size = 36
End With
' Apply style
Application.ActiveSheet.Range("A1:B3").Style = "Bordered"
```

Example: Sorting Sheets in a Workbook

Let us add a new utility to our SRXUtils application. If you work with workbooks that contain many sheets (worksheets and chartsheets), then you may want to sort the sheets in alphabetical order.

The basis for the code to order the sheets is the Move method of the Worksheet and Chart objects. Its syntax is:

```
SheetsObject.Move(Before, After)
```

Of course, to use this method effectively, we need a sorted list of sheet names.

The first step is to augment the DataSheet worksheet for SRXUtils by adding a new row for the new utility, as shown in Figure 17-8. (The order of the rows in this DataSheet is based on the order in which we want the items to appear in the custom menu.)

	A	B	C	D	E	F	G	H
1	**Utility**	**OnAction Proc**	**Procedure**	**In Workbook**	**Menu Item**	**SubMenu Item**	**On Wks Menu**	**On Chart Menu**
2	Activate Sheet	RunUtility	ActivateSheet	ThisWorkbook	&Activate Sheet		TRUE	TRUE
3	Print Charts	RunUtility	PrintCharts	Print.utl	&Print	Embedded &Charts	TRUE	TRUE
4	Print Pivot Tables	RunUtility	PrintPivotTables	Print.utl		&Pivot Tables	TRUE	TRUE
5	Print Sheets	RunUtility	PrintSheets	Print.utl		&Sheets	TRUE	TRUE
6	Sort Sheets	RunUtility	SortSheets	ThisWorkbook	&Sort Sheets		TRUE	TRUE

Figure 17-8. Augmenting the DataSheet worksheet

Next, we insert a new code module called *basSortSheets*, which will contain the code to implement this utility.

We shall include two procedures in *basSortSheets*. The first procedure verifies that the user really wants to sort the sheets. If so, it calls the second procedure, which does the work. The first procedure is shown in Example 17-1. It displays the dialog box shown in Figure 17-8.

Example 17-1. The SortSheets Procedure

```
Sub SortSheets()
  If MsgBox("Sort the sheets in this workbook?", _
  vbOKCancel + vbQuestion, "Sort Sheets") = vbOK Then
    SortAllSheets
  End If
End Sub
```

The action takes place in the procedure shown in Example 17-2. The procedure first collects the sheet names in an array, then places the array in a new worksheet. It then uses the Sort method (applied to a Range object, discussed in Chapter 19) to sort the names. Then, it refills the array and finally, reorders the sheets using the Move method.

Example 17-2. The SortAllSheets Procedure

```
Sub SortAllSheets()

' Sort worksheets
Dim wb As Workbook
Dim ws As Worksheet
Dim rng As Range
Dim cSheets As Integer
Dim sSheets() As String
Dim i As Integer

Set wb = ActiveWorkbook

' Get true dimension for array
cSheets = wb.Sheets.Count
ReDim sSheets(1 To cSheets)

' Fill array with worksheet names
For i = 1 To cSheets
  sSheets(i) = wb.Sheets(i).Name
Next

' Create new sheet and put names in first column
Set ws = wb.Worksheets.Add
For i = 1 To cSheets
  ws.Cells(i, 1).Value = sSheets(i)
Next

' Sort column
ws.Columns(1).Sort Key1:=ws.Columns(1), _
   Order1:=xlAscending

' Refill array
For i = 1 To cSheets
  sSheets(i) = ws.Cells(i, 1).Value
Next

' Delete extraneous sheet
Application.DisplayAlerts = False
ws.Delete
Application.DisplayAlerts = True

' Reorder sheets by moving each one to the end
For i = 1 To cSheets
  wb.Sheets(sSheets(i)).Move After:=wb.Sheets(cSheets)
Next

End Sub
```

Once the code is inserted, you can save the *SRXUtils.xls* workbook as an add-in. Don't forget to unload the add-in first, or Excel will complain.