ARTICLE 3

Integrating Outlook and Other Applications with VBA

Every application in both the immediate suite of Microsoft® Office tools and the extended family of Office applications uses Microsoft Visual Basic for Applications (VBA) as its macro and programming language. This greatly simplifies the task of writing code to manipulate more than one Office application. After you understand the object models of the individual applications, you can write code to automate any of them from your application of choice.

Note

The extended Office family includes products such as Microsoft Visio, Microsoft OneNote, Microsoft Publisher, Microsoft InfoPath, and Microsoft Project, among others. They are all VBA-enabled applications, which makes them ideal for integrating into Office solutions.

This chapter picks up where Article 2, "Using VBA in Outlook," left off. That chapter introduced you to VBA, its essential components, and some simple procedures. Now this chapter looks at some basic ways you can use VBA to make Office applications interact. After you understand these fundamentals, you can extend your knowledge by learning each application's object model and finding new ways to make the products work in harmony.

Note

You'll find the code used to create the VBA applications in this chapter in the Author Extras section of the companion CD.

Starting Other Office Applications

Although Microsoft Office Outlook 2007 might be your primary Office application, you probably need to use a variety of Office programs. For example, you might work with tasks in Outlook and then review the task information in a Microsoft Word document or a Microsoft Excel spreadsheet. To do this easily, you need to be able to launch these other programs from Outlook.

The task of opening another Office application is simple because the applications share a common language and the Office object model. To make the process easier, you tell the VBA environment about the application you intend to use by adding a reference to that application.

Outlook then gathers information about all the possible operations of the application you referenced and can then display the possible operations as you work in the environment, as if this were part of the native Outlook object model. This process is known as *early binding*.

To add a reference to Excel, for example, follow these steps:

1. Start Outlook and choose Tools, Macro, Visual Basic Editor (or press Alt+F11) to open the VBA Editor.
2. In the VBA Editor, choose Tools, References to open the References dialog box (see Figure A3-1).

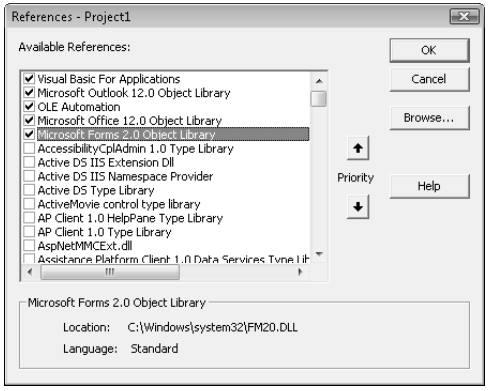

Figure A3-1 In this dialog box, you can add a reference to the Excel Object Library.

3. Click the entry for Microsoft Excel 12.0 Object Library.
4. Click OK.

Now that you've added a reference to the Excel object model, you can open an instance of Excel and any specific worksheet in it. The following code opens a specific Excel file:

```
Dim objExpenseEntry As Excel.Application
Set objExpenseEntry = New Excel.Application
objExpenseEntry.Visible = False
objExpenseEntry.Workbooks.Open "c:\temp\OutlookBook.xls"
objExpenseEntry.Visible = True
```

Because you added a reference to Excel, you can declare and instantiate the *Excel.Application* object just as you can any other object in Outlook. After you have an instance of Excel, you use the *Visible* and *Open* methods to open an Excel spreadsheet and display the Excel application.

You can use the same approach to open any other Office application. In general, do the following for the application you want to use:

1. Add a reference to the application.
2. Create an instance of the application in your code.
3. Manipulate the application using its standard object model.
4. Use the Visible property of the object to make the application visible.

Instantiation vs. Assignment

Instantiating refers to the process of creating an instance of an object. When you click the Word icon on your desktop, for example, you are instantiating a copy of Word that you can then use. The same is true when you use the following code from VBA:

```
Dim objX as Word.Application
```

This code says that at some time you'll use a *Word.Application* object and *objX* will be the placeholder for it. In the next line, you instantiate that object (create a new instance of Word):

```
    Set objX = New Word.Application
```

This process is different from an assignment. For objects, an assignment takes a new variable you've created and points it at an already existing instance of an object rather than creating a completely new instance. The following code declares a placeholder for an Outlook folder type object: Dim objY as Outlook.MAPIFolder The next line indicates that you have a placeholder for a folder type object and want it to point to the Tasks folder:

```
Set objY = Application.GetNamespace("MAPI").GetDefaultFolder(olFolderTasks)
```

Everything you do with this folder now affects the Tasks folder in Outlook.

To add code that opens Excel from within Outlook, follow these steps:

1. Start Outlook and choose Tools, Macro, Visual Basic Editor (or press Alt+F11) to open the VBA Editor.
2. In the Project window, select Project1.
3. Expand the tree until you see the heading ThisOutlookSession and then select it.
4. Press F7 to open the Code window.
5. Choose Tools, References to open the References dialog box.
6. Scroll through the list in the dialog box until you find an entry for Microsoft Excel 11.0 Object Library and select it. Close the dialog box.
7. Enter the following code in the Code window:

```
Public Sub OpenExcel()
    Dim objExcel As Excel.Application
    Set objExcel = New Excel.Application
    With objExcel
        .Visible = False
        .Workbooks.Open "c:\temp\OutlookBook.xls"
        .Visible = True
    End With
End Sub
```

8. Create an Excel file named OutlookBook.xls and place it in the C:\Temp directory.
9. Return to the main Outlook window and choose Tools, Macro, Macros.
10. Select ThisOutlookSession.OpenExcel and click Run. The new Excel spreadsheet that you created opens in Excel.

Using the Object Browser

To find out what you can do with an object for which you've created a reference, you can use a tool called the Object Browser, which can be opened from the View menu or by pressing F2 in the VBA Editor. With this tool, you can view either all the referenced libraries or a specific one. You can also look at all the declared properties, methods, events, and constants that are available for use. You can search on a keyword to see whether a property or function with that name is supported and to see how it's defined. For more information on using the Object Browser, consult the Help files.

Exchanging Data with Excel

Now that Excel is open, along with the workbook OutlookBook.xls, it's easy to exchange data with the workbook. For example, in Article 2, you created an application that changed the company name for all contacts in a specific company (see "Creating an Outlook Application with VBA," page A32). Here, you can add a routine that saves the contact details as an Excel spreadsheet that you can then distribute to others in the company. The following procedure performs this routine:

```
Public Sub ExportContacts(strCompany As String)
   Dim fdrContacts As Outlook.MAPIFolder
   Dim fdrContactsByCompany As Outlook.Items
   Dim objExcel As Excel.Application
   Dim objWorkbook As Excel.Workbook
   Dim objWorksheet As Excel.Worksheet
   Dim itmContacts As Outlook.ContactItem
   Dim iCol As Integer
   Dim iRow As Integer Set fdrContacts = _
     Application.GetNamespace("MAPI") _
     .GetDefaultFolder(olFolderContacts)
   Set fdrContactsByCompany = _
     fdrContacts.Items.Restrict("[CompanyName] = '" & _
     strCompany & "'")
   Set objExcel = New Excel.Application
   Set objWorkbook = objExcel.Workbooks.Add
   Set objWorksheet = objWorkbook.Worksheets.Add
   objWorksheet.Name = "Contacts for " & strCompany
   Set itmContacts = fdrContactsByCompany.GetFirst
   If itmContacts Is Nothing Then
       MsgBox "There are no contacts for that company. " & _
            "Please enter a different company name."
       Exit Sub
   End If
   iRow = 1
   For iCol = 0 To itmContacts.ItemProperties.Count - 1
       objWorksheet.Cells(iRow, iCol + 1) = _
           itmContacts.ItemProperties(iCol).Name
   Next iCol
   iRow = iRow + 1
   For Each itmContacts In fdrContacts.Items _
     .Restrict("[CompanyName] = '" & strCompany & "'")
       For iCol = 0 To itmContacts.ItemProperties.Count - 1
           Debug.Print itmContacts.ItemProperties(iCol).Name
           If itmContacts.ItemProperties(iCol).Type = olText Then
           objWorksheet.Cells(iRow, iCol + 1) = _
           itmContacts.ItemProperties(iCol).Value
           End If
       Next iCol
       iRow = iRow + 1
   Next
   objExcel.Visible = True
End Sub
```

You should be familiar with a large portion of this code from Article 2. Here's the part of the code that opens a new copy of Excel, adds a workbook and worksheet, and names the worksheet after the company name used as the filter:

```
Set objExcel = New Excel.Application
Set objWorkbook = objExcel.Workbooks.Add
Set objWorksheet = objWorkbook.Worksheets.Add
objWorksheet.Name = "Contacts for " & strCompany
```

After an Excel object is available, it's time to start adding data to it. Begin by adding a row that will be the header for the columns of data. To do this, you use a collection associated with a contact item that holds all the different properties associated with that item. Each of these properties has a name, so it's a simple task to loop through them, writing the name value to cells in the first row of the Excel worksheet using the worksheet object's *Cells* method. This method allows you to specify a row and column location for the cell to access:

```
iRow = 1
For iCol = 0 To itmContacts.ItemProperties.Count - 1
   objWorksheet.Cells(iRow, iCol + 1) = _
   itmContacts.ItemProperties(iCol).Name
Next iCol
```

After all the headings have been added, it's time to add data to the worksheet. You do this using two nested loops. The outside loop steps through each contact item that matches the company name value. The inner loop then steps through each property of the specific item; if it's a text type property, it's saved in the appropriate cell in the worksheet:

```
iRow = iRow + 1
For Each itmContacts In fdrContacts.Items _
   .Restrict("[CompanyName] = '" & strCompany & "'")
   For iCol = 0 To itmContacts.ItemProperties.Count - 1
   If itmContacts.ItemProperties(iCol).Type = olText Then
       objWorksheet.Cells(iRow, iCol + 1) = _
       itmContacts.ItemProperties(iCol).Value
   End If
   Next iCol
   iRow = iRow + 1
Next
```

The `If itmContacts.ItemProperties(iCol).Type = olText Then` test is performed to ensure that any data an Excel cell can't store, such as data of type object, is not written to Excel. To guard against errors, check each property before writing it to Excel to make sure it's a valid type. Another way to handle this issue would be to specify exactly which properties you want to write to Excel and then, rather than looping through every property of the item object, just write lines of code to export each selected property.

To add code that opens Excel and exports contacts from Outlook into an Excel workbook, follow these steps:

1. Start Outlook and choose Tools, Macro, Visual Basic Editor (or press Alt+F11) to open the VBA Editor.
2. In the Project window, select Project1.
3. Expand the tree to one level, expand the Modules branch, and then select the basChangeCompanyName module (which you built in Article 2).
4. Press F7 to open the Code window.
5. Enter the following code in the Code window:

```
Public Sub ExportContacts(strCompany As String)
    Dim fdrContacts As Outlook.MAPIFolder
    Dim fdrContactsByCompany As Outlook.Items
    Dim objExcel As Excel.Application
    Dim objWorkbook As Excel.Workbook
    Dim objWorksheet As Excel.Worksheet
    Dim itmContacts As Outlook.ContactItem
    Dim iCol As Integer
    Dim iRow As Integer
    Set fdrContacts = _
      GetNamespace("MAPI").GetDefaultFolder(olFolderContacts)
    Set fdrContactsByCompany = _
     fdrContacts.Items.Restrict("[CompanyName] = '" & _
     strCompany & "'")
    Set objExcel = New Excel.Application
    Set objWorkbook = objExcel.Workbooks.Add
    Set objWorksheet = objWorkbook.Worksheets.Add
    objWorksheet.Name = "Contacts for " & strCompany
    Set itmContacts = fdrContactsByCompany.GetFirst
    If itmContacts Is Nothing Then
       MsgBox "There are no contacts for that company. " & _
                "Please enter a different company name."
       Exit Sub
    End If
    iRow = 1
    For iCol = 0 To itmContacts.ItemProperties.Count - 1
       objWorksheet.Cells(iRow, iCol + 1) = _
       itmContacts.ItemProperties(iCol).Name
    Next iCol
    iRow = iRow + 1
    For Each itmContacts In _
      fdrContacts.Items.Restrict("[CompanyName] = '" & _
      strCompany & "'")
        For iCol = 0 To itmContacts.ItemProperties.Count - 1
        Debug.Print itmContacts.ItemProperties(iCol).Name
```

```
            If itmContacts.ItemProperties(iCol).Type = olText Then
                objWorksheet.Cells(iRow, iCol + 1) = _
                itmContacts.ItemProperties(iCol).Value
            End If
            Next iCol
            iRow = iRow + 1
        Next
        objExcel.Visible = True
    End Sub
```

6. Open the frmCompanyNameChange form (which you created in Article 2) and add a button to it. Give the button the caption Excel and the name cmdExcel.

7. Press F7 to open the Code window and enter this code:

```
Private Sub cmdExcel_Click()
    ExportContacts cmbFrom
End Sub
```

8. Return to Outlook and choose Tools, Macro, Macros. Select AddToolbar from the list and click Run. Click the Change Company Name button (which you created in Article 2). The form appears with the new Excel button.

9. Select a company in the combo box and click Excel. Excel opens, displaying a list of all the contacts for the selected company.

Exchanging Data with Word

A great way to use Outlook, Word, and VBA is to send a letter to a contact while viewing that contact's information. You can accomplish this by using some VBA code to automate Word.

Before you start, add a reference to the Word object model the same way you did for Excel on page A46, except that this time you select Microsoft Word 12.0 Object Library in the list. This gives you access to all the objects in Word that are available using VBA.

The following procedure uses Word to create a letter document for each of the currently selected contacts:

```
Private Sub SendLetterToContact()
   Dim itmContact As Outlook.ContactItem
   Dim selContacts As Selection
   Dim objWord As Word.Application
   Dim objLetter As Word.Document
   Dim secNewArea As Word.Section
   Set selContacts = Application.ActiveExplorer.Selection
   If selContacts.Count > 0 Then
       Set objWord = New Word.Application
       For Each itmContact In selContacts
           Set objLetter = objWord.Documents.Add
           objLetter.Select
```

```
            objWord.Selection.InsertAfter
            itmContact.FullName
            objLetter.Paragraphs.Add
            If itmContact.CompanyName <> "" Then
            objWord.Selection.InsertAfter
            itmContact.CompanyName
            objLetter.Paragraphs.Add
            End If
            objWord.Selection.InsertAfter
            itmContact.BusinessAddress
            objWord.Selection.Paragraphs.Alignment = wdAlignParagraphRight
            With objLetter
            .Paragraphs.Add
            .Paragraphs.Add
            End With
            With objWord.Selection
            .Collapse wdCollapseEnd
            .InsertAfter "Dear " & itmContact.FullName
            .Paragraphs.Alignment = wdAlignParagraphLeft
            End With
            Set secNewArea = objLetter.Sections.Add(Start:=wdSectionContinuous)
            With secNewArea.Range
            .Paragraphs.Add
            .Paragraphs.Add
            .InsertAfter "<Insert text of letter here>"
            .Paragraphs.Add
            .Paragraphs.Add
            End With
            Set secNewArea = objLetter.Sections.Add(Start:=wdSectionContinuous)
            With secNewArea.Range
            .Paragraphs.Add
            .InsertAfter "Regards"
            .Paragraphs.Add
            .Paragraphs.Add
            .InsertAfter Application.GetNamespace("MAPI").CurrentUser
            End With
        Next
        objWord.Visible = True
    End If
End Sub
```

The first section of code declares all the variables and objects you'll use. Notice the use of both Outlook and Word objects:

```
Dim itmContact As Outlook.ContactItem
Dim selContacts As Selection
Dim objWord As Word.Application
Dim objLetter As Word.Document
Dim secNewArea As Word.Section
```

The second section investigates Outlook to access the selected items. The program creates a *Selection* object from the currently selected items; as long as at least one item is selected, an instance of Word is created using the *New Word.Application* call. The program then loops through each item in the *Selection* collection and displays Word by setting the *Word.Application* object's *Visible* property to True:

```
Set selContacts = Application.ActiveExplorer.Selection
If selContacts.Count > 0 Then
   Set objWord = New Word.Application
   For Each itmContact In selContacts
       'Construct Letter Here
   Next
   objWord.Visible = True
End If
```

The heart of the letter construction is performed in the loop, where the documents are created and populated. The program starts by creating a new document in Word using the *Add* method of the document's collection:

```
Set objLetter = objWord.Documents.Add
```

After this, you can start inserting data into the newly created document. You use the *InsertAfter* method of the *Word.Selection* object, which adds lines of text, and the *Add* method of the *Paragraphs* object, which adds new paragraphs (blank lines) to the document:

```
objLetter.Select
objWord.Selection.InsertAfter itmContact.FullName
objLetter.Paragraphs.Add
If itmContact.CompanyName <> "" Then
   objWord.Selection.InsertAfter itmContact.CompanyName
   objLetter.Paragraphs.Add
End If
objWord.Selection.InsertAfter itmContact.BusinessAddress
objWord.Selection.Paragraphs.Alignment = wdAlignParagraphRight
With objLetter
   .Paragraphs.Add
   .Paragraphs.Add
End With
With objWord.Selection
   .Collapse wdCollapseEnd
   .InsertAfter "Dear " & itmContact.FullName
   .Paragraphs.Alignment = wdAlignParagraphLeft
End With
Set secNewArea = objLetter.Sections.Add(Start:=wdSectionContinuous)
With secNewArea.Range
   .Paragraphs.Add
   .Paragraphs.Add
   .InsertAfter "<Insert text of letter here>"
   .Paragraphs.Add
   .Paragraphs.Add
End With
Set secNewArea = objLetter.Sections.Add(Start:=wdSectionContinuous)
```

To finish the letter, you need to add a signoff. The program extracts the name of the current user of Outlook and inserts this in the letter:

```
With secNewArea.Range
   .Paragraphs.Add .InsertAfter "Regards"
   .Paragraphs.Add
   .Paragraphs.Add
   .InsertAfter Application.GetNamespace("MAPI").CurrentUser
End With
```

When you've finished, the final letter looks similar to the one shown in Figure A3-2.

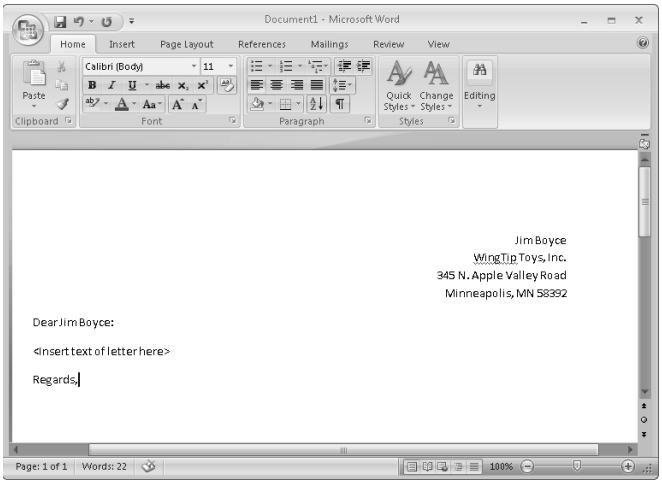

Figure A3-2. This letter in Word was generated from an Outlook contact item.

This solution leaves Word visible for the user to work with. It would be simple to completely automate the letter production by inserting any text for the body of the letter at this time and then saving it automatically:

```
objLetter.SaveAs "c:\temp\" & itmContact.FullName & ".doc"
objLetter.Close
```

To add code that prepares a set of letters for selected contacts, follow these steps:

1. Start Outlook and choose Tools, Macro, Visual Basic Editor (or press Alt+F11) to open the VBA Editor.
2. In the Project window, select Project1.
3. Choose Insert, Module to add a custom module to the project.
4. In the Project window, select the new module.
5. Go to the Properties window (press F4 to display the window if it's not visible) and change the module name to basExternalApps.

6. Press F7 to open the Code window for the module.
7. Choose Tools, References to open a dialog box displaying a list of items.
8. Scroll through this list until you find the entry for Microsoft Word 12.0 Object Library and then select it. Close the dialog box.
9. Enter the following code in the Code window:

```
Sub AddToolbar2()
    Dim tlbCustomBar As CommandBar
    Dim btnNewCustom As CommandBarButton
    Set tlbCustomBar = Application.ActiveExplorer.CommandBars _
      .Add(Name:="Custom External Applications", _
      Position:=msoBarTop, Temporary:=True)
    tlbCustomBar.Visible = True
    Set btnNewCustom = _
      tlbCustomBar.Controls.Add(Type:=msoControlButton)
    btnNewCustom.OnAction = "SendLetterToContact"
    btnNewCustom.Style = msoButtonIconAndCaption
    btnNewCustom.Caption = "Send Letter to Contact"
End Sub

Public Function SendLetterToContact()
    Dim itmContact As Outlook.ContactItem
    Dim selContacts As Selection
    Dim objWord As Word.Application
    Dim objLetter As Word.Document
    Dim secNewArea As Word.Section
    Set selContacts = Application.ActiveExplorer.Selection
    If selContacts.Count > 0 Then
        Set objWord = New Word.Application
        For Each itmContact In selContacts
        Set objLetter = objWord.Documents.Add
        objLetter.Select
        objWord.Selection.InsertAfter itmContact.FullName
        objLetter.Paragraphs.Add
        If itmContact.CompanyName <> "" Then
               objWord.Selection.InsertAfter itmContact.CompanyName
               objLetter.Paragraphs.Add
        End If
        objWord.Selection.InsertAfter itmContact.BusinessAddress
        objWord.Selection.Paragraphs.Alignment = _
          wdAlignParagraphRight
        With objLetter
               .Paragraphs.Add
               .Paragraphs.Add
        End With
        With objWord.Selection
               .Collapse wdCollapseEnd
               .InsertAfter "Dear " & itmContact.FullName
               .Paragraphs.Alignment = wdAlignParagraphLeft
        End With
```

```
            Set secNewArea = _
              objLetter.Sections.Add(Start:=wdSectionContinuous)
            With secNewArea.Range
                    .Paragraphs.Add
                    .Paragraphs.Add
                    .InsertAfter "<Insert text of letter here>"
                    .Paragraphs.Add
                    .Paragraphs.Add
            End With
            Set secNewArea = _
              objLetter.Sections.Add(Start:=wdSectionContinuous)
            With secNewArea.Range
                    .Paragraphs.Add
                    .InsertAfter "Regards"
                    .Paragraphs.Add
                    .Paragraphs.Add
                    .InsertAfter
                    Application.GetNamespace("MAPI").CurrentUser
            End With
            Next
            objWord.Visible = True
        End If
    End Function
```

10. Go back to the main Outlook window and choose Tools, Macro, Macros.
11. Select AddToolbar2 in the list, and then click Run. A new custom button with the name Send Letter To Contact is added to the toolbar.
12. Switch to the Contacts folder in Outlook and select one or more contacts.
13. Click the button labeled Send Letter To Contact to display one Word document for each selected contact.

INSIDE OUT

Press F1 if you need help

If you get stuck while programming in VBA, your first course of action should be to press the F1 key to launch the context-sensitive Help feature. The Help window will provide information about whatever you're currently working with, be it a keyword, an object property, or a window in the environment.

Exchanging Data with Access

Now that you've seen how to manipulate Word and Excel from Outlook using instances of the *Word.Application* or *Excel.Application* object, it's time to look at how you can bring data into Outlook from Microsoft Access. Suppose that you have an Access application

that holds details of projects and associated tasks and you need to import the task data from Access and turn it into Outlook tasks. To accomplish this, you could use the following code:

```
Public Sub ImportTasksFromAccess()
   Dim fdrTasks As Outlook.MAPIFolder
   Dim itmTask As Outlook.TaskItem
   Dim rsTasks As ADODB.Recordset
   Dim conTasks As ADODB.Connection
   Dim strConnectionString As String
   'Set the connection string and open the connection
   Set conTasks = New ADODB.Connection
   strConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
               "Data Source=C:\Temp\OutlookBook.mdb;" & _
               "Persist Security Info=False"
   conTasks.Open strConnectionString
   'Attempt to retrieve task records from the database for the given job
   Set rsTasks = New ADODB.Recordset
   rsTasks.Open "select * from Tasks" , _
     conTasks, adOpenStatic, adLockReadOnly
   Set fdrTasks = _
     Application.GetNamespace("MAPI").GetDefaultFolder(olFolderTasks)
   'Add tasks
   Do While Not rsTasks.EOF
       Set itmTask = fdrTasks.Items.Add
       With itmTask
           'Add custom properties to the task item
           .UserProperties.Add "TaskID", olText
           'Populate the task properties
           .UserProperties("TaskID") = rsTasks.Fields("TaskID")
           .Subject = rsTasks.Fields("Name")
           .Body = IIf(IsNull(rsTasks.Fields("Description")), "", _
           rsTasks.Fields("Description"))
           .PercentComplete = rsTasks.Fields("PercentComplete")
           .Save
       End With
       rsTasks.MoveNext
   Loop
End Sub
```

This procedure uses ActiveX Data Objects (ADO), which provides a simple way to read data in a Microsoft data store such as Access or SQL Server. The code reads the Access database and then stores all retrieved task records as Outlook tasks. ADO is an efficient way to extract data from either an Access database or a more complex database system such as SQL Server. To change this procedure to work with SQL Server or a different Access database, you need to alter only the *Connection* string.

For every record in the ADO recordset, the *Items.Add* method creates a new task item in the Tasks folder. The basic properties of each task are then assigned values from the record.

In this example, you add a *UserProperty* to the Task object for storing the ID value of the Task record. You use a *UserProperty* to access a custom field. To add a *UserProperty* to an item, you use the *Add* method of the *UserProperties* collection associated with the item, giving the property a name and a type:

```
.UserProperties.Add "TaskID", olText
```

You can then assign data to it by referencing the specific property as you would any other collection item:

```
.UserProperties("TaskID") = rsTasks.Fields("TaskID")
```

To add code that opens Access and imports information, follow these steps:

1. Copy the OutlookBook.mdb file (from this CD) to C:\Temp.
2. Start Outlook and choose Tools, Macro, Visual Basic Editor (or press Alt+F11) to open the VBA Editor.
3. In the Project window, select Project1.
4. Select the basExternalApps module, which you created in the preceding section.
5. Press F7 to open the Code window for the module.
6. Choose Tools, References to open the References dialog box.
7. Scroll through the list until you find Microsoft ActiveX Data Objects 2.8 and then select it. (If 2.7 is not available, select the highest number in the list.) Close the dialog box.
8. Enter the following code in the Code window:

```
Public Sub ImportTasksFromAccess()
    Dim fdrTasks As Outlook.MAPIFolder
    Dim itmTask As Outlook.TaskItem
    Dim rsTasks As ADODB.Recordset
    Dim conTasks As ADODB.Connection
    Dim strConnectionString As String
    'Set the connection string and open the connection
    Set conTasks = New ADODB.Connection
    strConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" & _
       "Data Source=C:\Temp\OutlookBook.mdb;" & _
       "Persist Security Info=False"
    conTasks.Open strConnectionString
    'Attempt to retrieve task records from the database for the given job
    Set rsTasks = New ADODB.Recordset
    rsTasks.Open "select * from Tasks" , _
      conTasks, adOpenStatic, adLockReadOnly
    Set fdrTasks = _
      GetNamespace("MAPI").GetDefaultFolder(olFolderTasks)
    'Add tasks
    Do While Not rsTasks.EOF
       Set itmTask = fdrTasks.Items.Add
```

```
        With itmTask
        'Add custom properties to the task item
        .UserProperties.Add "TaskID", olText
        'Populate the task properties
        .UserProperties("TaskID") = rsTasks.Fields("TaskID")
        .Subject = rsTasks.Fields("Name")
        .Body = IIf(IsNull(rsTasks.Fields("Description")), "", _
          rsTasks.Fields("Description"))
        .PercentComplete = rsTasks.Fields("PercentComplete")
        .Save
        End With
        rsTasks.MoveNext
    Loop
End Sub
```

9. Locate the AddToolbar2 procedure and add the following code to the end of it (before the End Sub statement):

```
Set btnNewCustom = _
  tlbCustomBar.Controls.Add(Type:=msoControlButton)
btnNewCustom.OnAction = "ImportTasksFromAccess"
btnNewCustom.Style = msoButtonIconAndCaption
btnNewCustom.Caption = "Import Tasks From Access"
```

10. Go back to the main Outlook window and choose Tools, Macro, Macros.

11. In the list, select AddToolbar2, and then click Run. A new custom menu button with the name Import Tasks From Access is added to the toolbar.

12. Switch to the Tasks folder in Outlook and click Import Tasks From Access. Tasks are created within Outlook that represent the tasks stored in the OutlookBook.mdb database.

INSIDE OUT Using ADO

When you use ADO, it's important to ensure that all users have the same version of ADO installed on their computers. If the versions differ, you could get unpredictable results from any code written using ADO.

To check the ADO version from code, you can use the Version property of the ADO *Connection* object. Declare an ADO *Connection* object, as shown here:

```
Dim ado as New ADODB.Connection
```

and then investigate the *Version* property:

```
ado.Version
```

You can now display a message asking users to update if they're running a different version.

Automating Outlook from Other Applications

You've seen how to integrate other applications by writing code in Outlook to manipulate them. It's just as easy to add code to other VBA-enabled applications to automate Outlook functionality. A common task in an external application is to use Outlook to send an e-mail message. In the following example, Outlook is automated from Excel to send an e-mail message that includes the current workbook:

```
Sub SendCurrentWorkbook(streMail As String)
    Dim objOutlook As Outlook.Application
    Dim itmNewEMail As Outlook.MailItem
    Dim itmNewTask As Outlook.TaskItem
    Set objOutlook = New Outlook.Application
    Set itmNewEMail = objOutlook.CreateItem(olMailItem)
    With itmNewEMail
        .To = streMail
        If ActiveWorkbook.Path = "" Then
        ActiveWorkbook.Save
        End If
        .Attachments.Add ActiveWorkbook.Path & "\" & ActiveWorkbook.Name
    .send
    End With
End Sub
```

This function shows how to automate Outlook to send e-mail messages by creating an instance of Outlook, as you did with Word and Excel, using the *Outlook.Application* object. After this object exists, you work just as if you were inside Outlook, creating a mail item and adding an attachment to it. To send the current workbook for review, use the *SendForReview* method:

```
ActiveWorkbook.SendForReview stremail, , False, True
```

TROUBLESHOOTING

When an external application tries to access Outlook, access is blocked

When an external application tries to automate e-mail within Outlook, the security features of Outlook step in to block the access. You'll receive a message asking you whether you want to allow the external application to access your e-mails. This blocking is intended to stop viruses such as the Melissa virus from replicating by automatically sending themselves in e-mails from Outlook. Currently, there is no workaround for this issue, as any possible fix would allow viruses to bypass the security. You can choose to allow access, but you should do this only when you're certain that the application making the request is under your control.

Because you've sent an e-mail for someone to review, it would be useful to add a task to remind you to follow up on that review in a specified number of days. To do this, add the following block of code to the example procedure:

```
Set itmNewTask = objOutlook.CreateItem(olTaskItem)
With itmNewTask
   .Subject = "Excel Workbook sent to - " & streMail
   .Body = "Follow up on details"
   .DueDate = Date + 3
   .Attachments.Add ActiveWorkbook.Path & "\" & ActiveWorkbook.Name
   .Save
End With
```

This code creates a new task item with a due date three days from today and attaches the Excel workbook so that you'll have a record of what is being reviewed. You can send this task and the Excel file to someone by adding the following code:

```
.Recipients.Add streMail
.Assign
.send
```

Here, another recipient is added to the list, and then the *Assign* method of the task item is called. This method allows a task to be assigned (delegated) to another user and must be used to alter the task item before it can be sent to any other user. The item is then sent using Outlook. If you go into Outlook and look in the Sent Items folder, you can see all the items that have been created and sent by this procedure.

To add code that sends an Excel workbook, follow these steps:

1. Start Excel with a new worksheet.
2. Choose Tools, Macro, Visual Basic Editor (or press Alt+F11) to open the VBA Editor in Excel.
3. Select the project in the Project window, and then choose Insert, Module.
4. Name the module basSendExcelWorkBk and press F7 to open the Code window for the module.
5. Choose Tools, References to open the References dialog box.
6. Scroll through the list until you find Microsoft Outlook 12.0 Object Library and then select it. Close the dialog box.
7. Enter the following code in the Code window:

```
Option Explicit
Sub EMailWorkbook()
    SendCurrentWorkbook "nunnm@plural.com"
End Sub

Sub SendCurrentWorkbook(streMail As String)
    Dim objOutlook As Outlook.Application
    Dim itmNewEMail As Outlook.MailItem
```

```
    Dim itmNewTask As Outlook.TaskItem
    Set objOutlook = New Outlook.Application
    Set itmNewEMail = objOutlook.CreateItem(olMailItem)
    With itmNewEMail
        .To = streMail
        If ActiveWorkbook.Path = "" Then
        ActiveWorkbook.Save
        End If
        .Attachments.Add ActiveWorkbook.Path & "\" & _
          ActiveWorkbook.Name
        .Send
    End With
    Set itmNewTask = objOutlook.CreateItem(olTaskItem)
    With itmNewTask
        .Subject = "Excel Workbook sent to - " & streMail
        .Body = "Follow up on details"
        .DueDate = Date + 3
        .Attachments.Add ActiveWorkbook.Path & "\" & _
           ActiveWorkbook.Name
        .Save
        .Assign
        .Recipients.Add streMail
        .Send
End With
End Sub
```

8. Replace the e-mail address nunnm@plural.com with an e-mail address you can use.

9. With the cursor still in the function, press F5 to run the code. An e-mail is sent to the selected address with the Excel workbook as an attachment, and a task is placed in the Outlook Tasks folder.

Outlook and XML

Extensible Markup Language (XML) is the new standard format for data exchange throughout the IT industry. In Office 2007, Excel and Access have native support for XML as a data transfer method. Outlook can join in by using a little bit of VBA code to automate the building of XML.

You can use the following procedure to save the contents of a folder as XML:

```
Sub SaveAsXML()
    Dim fdrActive As Outlook.MAPIFolder
    Dim rsXML As ADODB.Recordset
    Set fdrActive = ActiveExplorer.CurrentFolder
    Set rsXML = New ADODB.Recordset
    Dim itmType As Object
    Dim iCol As Integer
    On Error Resume Next
```

```
        rsXML.AddNew
        For Each itmType In fdrActive.Items
           For iCol = 0 To itmType.ItemProperties.Count - 1
           If itmType.ItemProperties.Item(iCol).Type = olText Then
                  rsXML.Fields.Append _
                  itmType.ItemProperties.Item(iCol).Name, adVarChar, 5000
           End If
           Next iCol
        Next
        If rsXML.State = adStateClosed Then
           rsXML.Open
        End If
        On Error GoTo NextItem
        For Each itmType In fdrActive.Items
           rsXML.AddNew
        For iCol = 0 To itmType.ItemProperties.Count - 1
           If itmType.ItemProperties.Item(iCol).Type = olText Then
           rsXML.Fields(itmType.ItemProperties.Item(iCol).Name).Value = _
           IIf(IsNull(itmType.ItemProperties.Item(iCol).Value), "", _
           itmType.ItemProperties.Item(iCol).Value)
           End If
        Next iCol
        rsXML.Update

        NextItem:
           Next
           On Error GoTo 0
           rsXML.Save "c:\temp\" & fdrActive.Name & ".xml", adPersistXML
     End Sub
```

This procedure has two key pieces of code. The first uses ADO to create a new recordset in memory and adds all the possible fields for the types of items that are being processed. This is accomplished by looping through the *ItemProperties* collection, much as you did when writing data to Excel, and appending fields with the same names as the properties to the recordset object:

```
For iCol = 0 To itmType.ItemProperties.Count - 1
   If itmType.ItemProperties.Item(iCol).Type = olText Then
       rsXML.Fields.Append _
         itmType.ItemProperties.Item(iCol).Name, adVarChar, 5000
   End If
Next iCol
```

The second core piece of code is the one that writes the data into the recordset and then saves that data as an XML file. For every item in the folder, the code adds a new record

to the ADO recordset and fills each of its fields with the corresponding Outlook item value:

```
For Each itmType In fdrActive.Items
   rsXML.AddNew
   For iCol = 0 To itmType.ItemProperties.Count - 1
       If itmType.ItemProperties.Item(iCol).Type = olText Then
           rsXML.Fields(itmType.ItemProperties.Item(iCol).Name).Value = _
           IIf(IsNull(itmType.ItemProperties.Item(iCol).Value), "", _
itmType.ItemProperties.Item(iCol).Value)
       End If
   Next iCol
   rsXML.Update
Next
```

To save the data as XML, simply call the *Save* method of the ADO recordset object and save it as an XML type file:

```
rsXML.Save "c:\temp\" & fdrActive.Name & ".xml", adPersistXML
```

This exports a file using the ADO *Persist* method, which stores the ADO recordset data as XML-Data Reduced (XDR) format. You could just as easily have reimported the data into an ADO recordset by using the *Open* method of an ADO recordset object and opening a file:

```
rsData.Open mstrXMLPath & "TimeTasks.xml", , , , adCmdFile
```

The procedure shown here for exporting XML data is generic and will fail if the folder you're exporting contains multiple different item types or if some of the items have custom parameters and some don't. To solve these problems, you might want to take a more customized route, in which you make a decision about exactly what to export instead of looping through all the properties. The preceding method exports everything, including a number of ID values that are not required outside Outlook.

To add code that exports a folder as an XML file, do the following:

1. Start Outlook and choose Tools, Macro, Visual Basic Editor (or press Alt+F11) to open the VBA Editor.
2. In the Project window, select Project1.
3. Select the basExternalApps module, which you created previously.
4. Press F7 to open the Code window for the module.

5. Enter the following code in the Code window:

```
Sub SaveAsXML()
    Dim fdrActive As Outlook.MAPIFolder
    Dim rsXML As ADODB.Recordset
    Set fdrActive = ActiveExplorer.CurrentFolder
    Set rsXML = New ADODB.Recordset
    Dim itmType As Object
    Dim iCol As Integer
    On Error Resume Next
    rsXML.AddNew
    For Each itmType In fdrActive.Items
       For iCol = 0 To itmType.ItemProperties.Count - 1
       If itmType.ItemProperties.Item(iCol).Type = olText Then
              rsXML.Fields.Append _
              itmType.ItemProperties.Item(iCol).Name, adVarChar, 5000
       End If
       Next iCol
    Next
    If rsXML.State = adStateClosed Then
       rsXML.Open
    End If
    On Error GoTo NextItem
    For Each itmType In fdrActive.Items
       rsXML.AddNew
       For iCol = 0 To itmType.ItemProperties.Count - 1
       If itmType.ItemProperties.Item(iCol).Type = olText Then
              rsXML.Fields(itmType.ItemProperties.Item(iCol).Name).Value = _
              IIf(IsNull(itmType.ItemProperties.Item(iCol).Value), "", _
              itmType.ItemProperties.Item(iCol).Value)
       End If
       Next iCol
       rsXML.Update
    NextItem:
       Next
       On Error GoTo 0
       rsXML.Save "c:\temp\" & fdrActive.Name & ".xml", adPersistXML
End Sub
```

6. Locate the AddToolbar2 procedure and add the following code to the end of it:

```
Set btnNewCustom = _
  tlbCustomBar.Controls.Add(Type:=msoControlButton)
btnNewCustom.OnAction = "SaveAsXML"
btnNewCustom.Style = msoButtonIconAndCaption
btnNewCustom.Caption = "Save Current Folder As XML"
```

7. Go back to the main Outlook window and choose Tools, Macro, Macros.

8. In the list, select AddToolbar2 and then click Run. A new custom menu button with the name Save Current Folder As XML is added to the toolbar.

9. Select a folder in Outlook and click Save Current Folder As XML. An XML file containing the contents of that folder is created and placed in C:\Temp.