ARTICLE 2

Using VBA in Outlook

Microsoft® Visual Basic for Applications (VBA) is derived from the full Microsoft Visual Basic language and is featured in a number of products. Most notably, it is the primary programming language for all Microsoft Office applications.

In this chapter, you'll be introduced to the VBA environment and some of the important concepts required to work with VBA. After this, you'll see some examples of simple ways to use VBA within Microsoft Office Outlook® 2007 to automate common tasks. It will help if you're familiar with a programming language, but you'll be able to run the samples simply by following the instructions.

> **Note**
> You'll find the code used to create the VBA applications in this chapter on the companion CD.

There are a lot of great books on VBA—whether you're a beginner or a more advanced user. Check out the Microsoft Press Web site at *mspress.microsoft.com* and your favorite bookstore for a title to fit your needs.

A Quick Look at Object Models

VBA was designed to be integrated into other applications. It allows a user to easily work with the functionality of its parent language, Visual Basic, as well as to access external functions and applications. To understand how to get the most out of VBA, it's important to understand the concept of an object model.

The term *object model* refers to the exposed functions of an application (the functions belonging to an application that you can access from within code). These functions are exposed as a set of objects, where each object has properties, methods, and events. The object model contains a set of definitions, or *classes*, that you can use to create objects.

You might think of a class as a design. Consider an architect's house plans. These define what the house should look like and what functions it should perform, but until the builder builds the house, the plan doesn't do anything except describe the house. When the builder builds the house, he or she is creating an *instance* of the architect's design. Although the builder might make many instances of the design when building a housing development, after a house (an instance) is created, it stands separate from any other house in the development.

The way you use the object model in VBA is similar to the way the plan is used to build a house. Select the set of functions you require, and then create an instance of the class that defines them. To accomplish this, you use VBA keywords and syntax.

Referencing Objects

To use objects in VBA, you must add a reference to them. This tells VBA that a specific group of objects exists and what classes are defined for these objects. To add a reference, start the VBA Editor and choose Tools, References. A list of objects is displayed. To add a reference to an object definition, just select the box next to the appropriate item in the list.

By looking at the Help file you can usually find out which references you need to set to perform certain operations. Some of the major ones for the examples in this chapter and the next are references to the object models of other Office applications. These appear in the list with the following names:

Excel—Microsoft Excel 12.0 Object Library

Word—Microsoft Word 12.0 Object Library

To access Outlook functionality from another application, you set a reference to the Outlook object model:

Outlook—Microsoft Outlook 12.0 Object Library

For data access, it is useful to use ActiveX Data Objects (ADO):

Data access—Microsoft ActiveX Data Objects 2.8 Library

A number of object models for ADO are available; the most recent is ADO 2.8.

Declaring and Instantiating an Object

To create an instance of an object, you first declare the object using the *Dim* statement. This statement dimensions a variable of that object type. The following statement, for example, declares a new variable of the *Word.Application* type:

```
Dim objWord as Word.Application
```

After you declare an object, you use the *Set* and *New* keywords to create an instance of the object. The variable you declared previously is assigned to this instance, after which you can start using the instance:

```
Set objWord = New Word.Application
```

You can now access all the functions and properties of the Word.Application object through the newly defined instance.

Disposing of an instance

Whenever you create an instance of an object, especially an external application object, remember to dispose of it once you've finished working with it. You dispose of it by setting the object variable to Nothing:

```
Set objWord = Nothing
```

Note

VBA is supposed to clean up all object references and dispose of them automatically when they fall out of scope, but often this does not happen. You are then left with objects that have no associations but are using memory. This situation can be especially harmful if the object you created is a large application like Microsoft Word that uses a significant amount of memory.

Properties, Methods, and Events

A *property* is a value associated with an object. An example of a property is the name of the object. Properties can be read-only or read-write. You usually use properties to tell an object how to represent itself or how to act.

A *method* is equivalent to a function in code, but it belongs to a specific object and can be accessed only through that object rather than being generally available. You can call the method of an object to perform a task and perhaps return a result. Methods can take parameters. An *event* is used by the object you have instantiated to tell your application that something is happening outside normal program flow. A great deal of the Microsoft Windows architecture is based on the availability of events (referred to as messages in relation to Windows).

Understanding VBA in Relation to VBScript

Before Microsoft Outlook 2000, the only option for developing code applications in Outlook was to use Microsoft Visual Basic, Scripting Edition (VBScript). Although this allowed some customization, the introduction of VBA in Outlook 2000 allowed developers a far greater level of control throughout the entire product, as well as easier integration with other members of the Office family. VBA differs from VBScript in a number of key ways. Most noticeably, VBA is a much friendlier environment in which to work. This is a result of the editor supplied with the VBA engine and also because VBA is closer than VBScript to a full-featured programming language such as Visual Basic.

An important difference between VBA and VBScript in Outlook is related to where you use the two languages. VBA allows you far greater control over Outlook than VBScript, which permits you to work only behind a particular custom form. In VBA, you can work at an application level, where you can control many interactions between different areas of Outlook as well as automate almost every interface action.

Overview of the VBA Development Environment

If you have programmed in one of the many VBA-enabled applications, you'll be familiar with the VBA environment in Outlook. It is the same environment used in all Office products as well as in a number of third-party applications.

The VBA environment is hosted as a window separate from the main Outlook application environment. To open the VBA Editor window from Outlook, choose Tools, Macro, Visual Basic Editor.

The environment consists of a number of windows and toolbars, some of which are displayed in Figure A2-1. The Project, Properties, and Debugging windows are integral to the VBA environment and can be docked on the edges of the environment. The Code and Form editors operate as multiple-document interface (MDI) windows, similar to the way multiple documents are displayed in Word or Microsoft Excel. They all appear within the shell, and you can easily switch between them using the Window menu. To view or hide any of these windows, use the View menu.

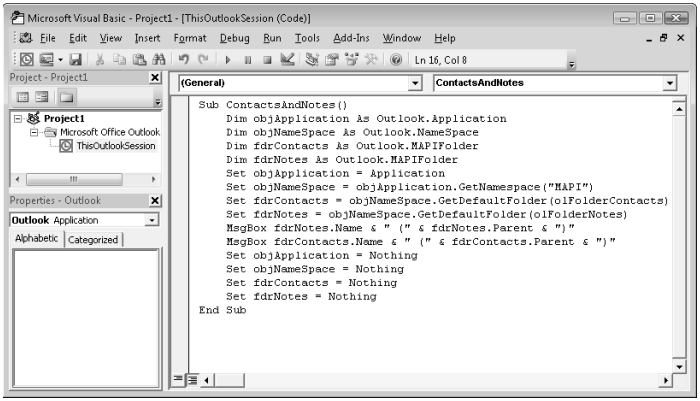

Figure A2-1. The VBA environment contains a number of windows and toolbars.

The Project Window

The Project window displays the projects and files you work on. The files that make up a project are arranged as a tree under the Project entry in the Project window.

Following are descriptions of the different sections of the project tree:

- **Microsoft Outlook Objects** This branch of the tree always exists and always contains at least one object called *ThisOutlookSession*. This is where you place code that works at the application level of Outlook. It represents the current instance of Outlook and gives you access to the application so that you can add code for some of the events that occur in Outlook. For example, if you want a task to occur when Outlook starts up, you place code in the *Startup* event of the *Application* object.
- **Forms** This branch of the tree holds an entry for any forms you have built. These are VB forms, which are not the same as Outlook forms. You would add VB forms to your application to allow a user to perform tasks that were not related to any of the custom areas of Outlook. For example, to perform a particular task, you might need to collect information from the user. You could build a custom VB form that is displayed when the user performs a specific task and that asks the user to provide some information.

For a demonstration of using a custom VB form in this way, see "Adding a Form" later in this article.

- **Modules** Under this branch of the tree, you find all the code modules that contain general code. This is a sensible place to put common code that must be available to many areas of the application. It is also a good place to store macro code used by buttons to launch various parts of your Outlook application. Later in this chapter, when you discover how to implement custom toolbar buttons, you'll see that the code used for this purpose is placed in a module. You can have many modules in your project; it's standard practice to divide your code into logical functional areas and to then implement each functional area as a different module. For example, you might have some functions that perform operations on contact information, some that perform operations with calendar items, some that relate to the handling of errors, and some that are simply general code functions that can be used by any of the functional areas. In such a case, you should implement four modules called basContacts, basCalendar, basErrorHandling, and basGeneral. (The bas prefix is historical, referring to the fact that these are files containing BASIC code.)
- **Class Modules** In this area, you can build class modules to utilize throughout your code. Although they permit you to build functions, class modules, unlike ordinary modules, require that to use the functions you must create an instance of the class rather than simply calling the function name. Class modules are used to encapsulate similar functionality in a single area that can be utilized like any other objects in Outlook.

> **Note**
>
> The discussion of how and why to use class modules to develop applications in an object-oriented way is too complex to be covered here. I suggest that you consult books on developing object-oriented applications to get an understanding of the techniques and then apply these techniques in your Outlook applications by using classes. One useful book on this subject is Programming Microsoft Visual Basic .NET, by Francesco Balena (Microsoft Press, 2002).

Follow these steps to add a new file to your VBA project:

1. Choose Insert.
2. Choose one of the three file types (UserForm, Module, Class Module). Under the appropriate branch of the project tree, a new file is displayed with a generic name.
3. Rename the file using the Properties window.

For instructions on renaming, see "The Properties Window," on the facing page.

Although the files in the Project window are displayed and listed as separate items, they are stored not as individual files but as placeholders to their storage in Outlook. You can, however, import and export them as individual files to allow their use in other applications. Follow these steps to export a file:

1. In the Project window, select the file you want to export.
2. Choose File, Export File or right-click and choose Export File.
3. Select the directory to which you want to export the file.
4. Name the file.

You also can import any VB or VBA module or class file into an Outlook project.

Follow these steps to import a file:

1. Right-click anywhere in the Project window or choose File, Import File.
2. Locate the file to import and select it in the dialog box. The imported file appears under the appropriate branch of the project tree.

All VBA code that you develop is saved in the VBAProject.OTM file. If you want to distribute an application, you need to send the VBAProject.OTM file and have people replace their existing VBAProject.OTM files with your new one. By default, the OTM file is located in Application Data\Microsoft\Outlook folder of your user profile.

The Properties Window

The Properties window is where you review and alter any configurable properties of the selected object. For any file selected in the Project window, a Name property appears in the Properties window, where you can change the name of the file. This window gets quite a workout when you're building a VB form because the form and every control you place on it are each associated with many properties.

For example, after adding the Module type file to the project, follow these steps to change the Name property of the file:

1. In the Project window, select the project file to rename. The Project window changes to display information about the selected file—in this case, the name.
2. Select the Name property and change it to something meaningful, such as basCommonFunctions.
3. Click away from the Name property, anywhere else in the VBA environment. The name of the file in the Project window changes to reflect the altered property.

To demonstrate a more complex set of properties, add a UserForm type file to the project.

Notice that the Properties window is now full of customizable properties (see Figure A2-2).

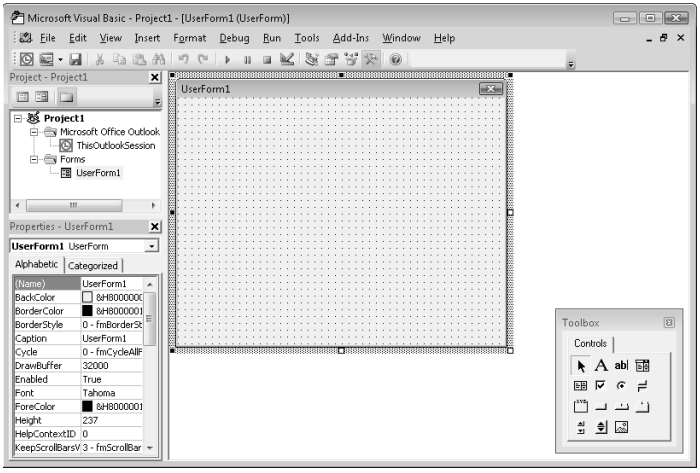

Figure A2-2. Forms and controls have multiple properties that you can change in the Properties window.

The Debugging Windows

A number of debugging tools are available for debugging your VBA applications. You use these tools to investigate and alter the states of the objects in your code while the code is executing. These tools appear with information in different debugging windows

that are displayed at the base of the screen. You can open the debugging windows by choosing View and then selecting the appropriate tools. Before you can use debugging tools, the code must be in break mode. To get into break mode while code is executing, press Ctrl+Break or set a breakpoint by selecting a line of code you're interested in and pressing F9. When the executing code reaches the breakpoint, the environment will go into break mode, and you can use the debugging tools.

Debugging tools include the following:

- **Immediate window** View values and execute statements while in break mode.
- **Watch window** View watches you have set up in VBA. A watch is an expression that evaluates as a program executes. This facility allows you to watch the value change without having to reexecute it manually, as you do in the Immediate window.
- **Locals window** View the values of all local variables and objects.

> **Note**
>
> For a view of the Immediate window, see Figure A2-1. The Immediate window is displayed at the bottom of the VBA environment.

The Code Window

When you double-click a file in the Project window, a new window opens in the main area of the screen. If you selected Module, ClassModule, or ThisOutlookSession, the Code window is displayed. You use this window to add code to Outlook.

The Object Window

If you double-click a form in the Project window, the Object window opens, displaying a graphical view of the form. To display the code for that form, do one of the following:

- Double-click the form or a control on it.
- Right-click the form entry in the Project window and choose View Code.
- Choose View, Code.

VBA Toolbars

You can use three special toolbars when building VBA applications. Most of the functions available by clicking toolbar buttons are also available on a menu and are usually accessible through a shortcut key combination as well.

- The Debug toolbar gives you easy access to the debugging tools.

- The Edit toolbar contains the tools that make you more productive when writing code.
- The UserForm toolbar aids in building VB forms.

Understanding the Outlook Object Model

Once you're comfortable with the VBA environment, it's time to have a look around Outlook's object model. If you're familiar with object models in other Office applications, you'll notice that the Outlook object model is slightly different. This is a result, for the most part, of the relatively late inclusion of VBA and the different role that Outlook as an application plays. Whereas other Office applications such as Word and Excel are frameworks within which other objects (documents, workbooks, and sheets) are hosted and manipulated, Outlook is very much an application in its own right.

An entire book could be written about the more than 30 objects and collections in the Outlook object model. This section gives only an introduction to the key objects and collections that are required to start developing applications. Use the Microsoft Visual Basic Help, accessed from within the VBA Editor by pressing F1, to find out about the other objects in the Outlook object model.

Here is a brief look at some key objects:

- **Items** The basic units of information in Outlook. An e-mail message, an appointment on your calendar, and a contact entry are all examples of items.
- **Folders** The basic storage units. Folders contain items. Outlook has many folders, such as the Inbox, Sent Items, and Tasks folders.
- **Explorers** The visual representation of the items in a folder. Outlook uses Explorers to display items. Examples of Explorers are the e-mail pane associated with the Inbox and the daily calendar view you see when you select the Calendar folder. Any one folder can be associated with a number of Explorers. For example, the Calendar folder has several Explorers so that you can display calendar items in different ways.
- **Inspectors** The Outlook forms used to display an item. Inspectors are to items what Explorers are to folders: that is, the graphical Outlook representation of the information.

These objects are explained more fully in the next section. Before looking at these important Outlook objects in more detail, however, it's important to understand what is meant by the term collection. Some pieces of the object model are flagged as an object only, and others are flagged as an object and a collection. A collection is a group of objects that have the same type. For example, your Inbox contains a number of e-mail messages. Each message is a mail item object, and the Inbox folder contains a collection of these mail items. You can get at a particular member of a collection by using the Item method of the collection and giving it an index that is an integer value or a value that

matches the default property of an object in the collection. For example, the following code returns the second e-mail message in the Inbox:

```
Set emlSecond = fdrInbox.Items.Item(2)
```

Alternatively, you can use a name that refers to the object. This example returns the e-mail folder called Outlook Book in the Folders collection:

```
Set fdrOutlookBook = myFolders.Item("Outlook Book")
```

To make the code that displays the second e-mail message work correctly, 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 and expand the tree until you see ThisOutlook- Session.
3. Select ThisOutlookSession and press F7 to open the Code window.
4. Enter the following in the Code window:

```
Sub GetEmailItem()
    Dim emlSecond As MailItem
    Dim nsMyNameSpace As NameSpace
    Dim fdrInbox As MAPIFolder
    Set nsMyNameSpace = Application.GetNamespace("MAPI")
    Set fdrInbox = nsMyNameSpace.GetDefaultFolder(olFolderInbox)
    Set emlSecond = fdrInbox.Items.Item(2)
    MsgBox "Second e-mail : " & vbCrLf & vbCrLf & _
        emlSecond.Subject & vbCrLf & emlSecond.Body
End Sub
```

5. Go back to the main Outlook window and choose Tools, Macro, Macros.
6. Select ThisOutlookSession.GetEmailItem and click the Run button. A dialog box opens, containing the e-mail message.

Application, Namespaces, and Folders

At the core of the Outlook object model is the *Application* object, referred to as the *root object* because the rest of the hierarchy grows from it. The *Application* object provides access to all other Outlook objects. If you're accessing the Outlook object hierarchy from an external application, you must create an instance of the *Application* object before you can access any other objects. If you're working in Outlook, an instance of the *Application* object is always in existence; to access it, you use the Application keyword.

Although the *Application* object gives you access to many fundamental building block objects in Outlook, you must create an instance of the *Namespace* object if you want to access Outlook data. The *Namespace* object is an abstract root for Outlook data sources,

which means that although you don't use it directly, it provides access to the objects below it in the object tree. Currently, the only data source supported is Messaging Application Programming Interface (MAPI), which allows access to all Outlook data stored in the user's mail files. To get at the *Namespace* object of the Outlook application, use the *GetNameSpace* method of the *Application* object:

```
Application.GetNamespace("MAPI")
```

As you know, information in Outlook is maintained in folders. Some folders, such as Inbox, Outbox, and Sent Items, contain mail items; other folders contain other types of items. After obtaining an instance of the *Namespace* object, you can easily connect to any folders in Outlook. The *Namespace* object has a *GetDefaultFolder* method that takes a parameter of type *olDefaultFolders*. Type *olDefaultFolders* represents one of the default Outlook folders and can be any of the constants shown in Table A2-1.

Table A2-1 olDefaultFolders Constants

Constant	Purpose
olFolderCalendar	Returns a folder containing all calendar items
olFolderContacts	Returns a folder containing all contact items
olFolderDeletedItems	Returns a folder containing all deleted mail items
olFolderDrafts	Returns a folder containing all draft mail items
olFolderInbox	Returns a folder containing all Inbox mail items
olFolderJournal	Returns a folder containing all journal items
OlFolderNotes	Returns a folder containing all note items
OlFolderOutbox	Returns a folder containing all Outbox mail items
OlFolderSentMail	Returns a folder containing all sent mail items
olFolderTasks	Returns a folder containing all task items
OlPublicFoldersAllPublicFolders	Returns a folder containing all public folder items

For example, you can use the following code to create an object that represents all contact items:

```
Dim fdrContacts As Outlook.MAPIFolder
Set fdrContacts = Application.GetNamespace("MAPI") _
   .GetDefaultFolder(olFolderContacts)
```

The *fdrContacts* variable has been declared as a MAPIFolder and assigned the Contacts folder. You could instead declare an *Application* object and a *Namespace* object as well, but unless you're going to use them repeatedly, it's just as efficient to use the defined *Application* property and the *GetNamespace* method call.

Efficient Use of the Application and Namespace Objects

If you want to retrieve a number of folders to work with, the following code is efficient because it retrieves an Application object and a Namespace object once and then uses them repeatedly:

```
Dim objApplication As Outlook.Application
Dim objNameSpace As Outlook.NameSpace
Dim fdrContacts As Outlook.MAPIFolder
Dim fdrNotes as Outlook.MAPIFolder
Set objApplication = Application
Set objNameSpace = objApplication.GetNamespace("MAPI")
Set fdrContacts = objNameSpace.GetDefaultFolder(olFolderContacts)
Set fdrNotes = objNameSpace.GetDefaultFolder(olFolderNotes)
'<Insert code here to work with the Contacts and Notes folders>
Set objApplication = Nothing
Set objNameSpace = Nothing
Set fdrContacts = Nothing
Set fdrNotes = Nothing
```

The following steps guide you through creating code that sets up objects for the Notes and Calendar folders and then allows you to display contents from within each folder:

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 and expand the tree until you see the heading ThisOutlookSession.
3. Select ThisOutlookSession and press F7 to open the Code window.
4. Enter the following code in the Code window:

```
Sub ContactsAndNotes()
Dim objApplication As Outlook.Application
Dim objNameSpace As Outlook.NameSpace
Dim fdrContacts As Outlook.MAPIFolder
Dim fdrNotes As Outlook.MAPIFolder
Set objApplication = Application
Set objNameSpace = objApplication.GetNamespace("MAPI")
Set fdrContacts = _ objNameSpace.GetDefaultFolder(olFolderContacts)
Set fdrNotes = _ objNameSpace.GetDefaultFolder(olFolderNotes)
MsgBox fdrNotes.Name & " (" & fdrNotes.Parent & ")"
MsgBox fdrContacts.Name & " (" & fdrContacts.Parent & ")"
Set objApplication = Nothing
Set objNameSpace = Nothing
Set fdrContacts = Nothing
Set fdrNotes = Nothing
End Sub
```

5. Go back to the main Outlook window and choose Tools, Macro, Macros.
6. Select ThisOutlookSession.ContactsAndNotes and click Run. Two dialog boxes are displayed. The first contains the name of the Notes folder followed by the name of its parent folder; the second contains the name of the Contacts folder followed by the name of its parent folder.

Explorers, Inspectors, and Items

So far, you have been introduced to the Outlook objects required to access data in Outlook. An item is one piece of Outlook data. For example, an appointment on your calendar is stored in Outlook as an appointment item. When you use Outlook to look at items in folders, you're actually using an *Explorer* object for that particular item type. Outlook provides different Explorers for the different types of items. If you use the Outlook interface to look at Article 2 Using VBA in Outlook A29 Part 9: Developing Custom Forms and Applications your calendar, contacts, notes, and journal items, you can see how markedly different the Explorers for the various items are.

The *Application* object contains a collection of Explorers that represent all the different Explorers available in Outlook. A number of methods are available for retrieving a specific *Explorer* object from Outlook:

- Use the *Item* method of the Explorers collection.
- Use the *ActiveExplorer* method, which returns the currently active Explorer in Outlook, if there is one:

```
Dim expActive As Outlook.Explorer
Set expActive = Application.ActiveExplorer()
```

- For a specific folder, use the *GetExplorer* method to return an instance of the Explorer for that folder:

```
Dim expContacts as Outlook.Explorer
Set expContacts = fdrContacts.GetExplorer
```

After you have an *Explorer* object, you can display that Explorer by calling the *Activate* method:

```
expContacts.Activate
```

Whereas the *Explorer* object is used to display a collection of items, an Inspector is used to display a specific item. You can think of an Inspector as the form you see when you look at a particular type of item. The ability to access Inspectors in code is useful if you want to create your own item and then allow your user to customize it. As with the Explorers, a collection of Inspectors is associated with the Application object. You can retrieve a specific Inspector in a number of ways:

- If an Inspector is open, a call to the ActiveInspector method returns that Inspector. You can access details of the item displayed in the Inspector by using the *CurrentItem* method:

```
Dim insActive As Outlook.Inspector
Dim itmCurrent As Object
Set insActive = Application.ActiveInspector
Set itmCurrent = insActive.CurrentItem
```

- You can access the Inspector associated with an item by using the *GetInspector* method:

```
Dim insAppointments As Outlook.Inspector
Dim itmAppointment As Outlook.AppointmentItem
Set insAppointments = itmAppointment.GetInspector
```

After you have an *Inspector* object, you can display it and its associated item by calling the *Activate* method:

```
insAppointments.Activate
```

An alternative to creating an Inspector object is to create a new item and then call the item's Display method to display the item and thus the item's Inspector:

```
Dim fdrCalendar As Outlook.MAPIFolder
Dim itmAppointment As Outlook.AppointmentItem
Set fdrCalendar = Application.GetNamespace("MAPI").GetDefaultFolder(olFolderCalendar)
Set itmAppointment = fdrCalendar.Items.Add
With itmAppointment
   .Subject = "Custom Appointment generated from Code"
   .Body = "Created in code and then displayed for you to edit"
   .Display
End With
```

To make the code that displays Explorers and Inspectors work, 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 and expand the tree until you see ThisOutlook- Session.
3. Select ThisOutlookSession and press F7 to open the Code window.
4. Enter the following code in the Code window:

```
Sub ShowExplorers()
   Dim objApplication As Outlook.Application
   Dim objNameSpace As Outlook.NameSpace
   Dim fdrContacts As Outlook.MAPIFolder
   Dim expContacts As Outlook.Explorer
   Set objApplication = Application
   Set objNameSpace = objApplication.GetNamespace("MAPI")
   Set fdrContacts = objNameSpace.GetDefaultFolder(olFolderContacts)
   Set expContacts = fdrContacts.GetExplorer
   expContacts.Activate
   Set objApplication = Nothing
   Set objNameSpace = Nothing
   Set fdrContacts = Nothing
   Set expContacts = Nothing
End Sub
```

```
Sub ShowContactsInspector()
   Dim itmContact As Outlook.ContactItem
   Dim insContact As Outlook.Inspector
   Dim fdrContacts As Outlook.MAPIFolder
   Set fdrContacts = GetNamespace("MAPI").GetDefaultFolder(olFolderContacts)
   Set itmContact = fdrContacts.Items.Add
   With itmContact
      .FirstName = "Auto"
      .LastName = "Created"
      .Body = "Created using the GetInspector method of an item"
   End With
   Set insContact = itmContact.GetInspector
   insContact.Activate
   Set itmContact = Nothing
   Set insContact = Nothing
   Set fdrContacts = Nothing
End Sub

Sub ShowAppointmentInspector()
   Dim fdrCalendar As Outlook.MAPIFolder
   Dim itmAppointment As Outlook.AppointmentItem
   Set fdrCalendar = GetNamespace("MAPI").GetDefaultFolder(olFolderCalendar)
   Set itmAppointment = fdrCalendar.Items.Add
   With itmAppointment
      .Subject = "Custom Appointment generated from Code"
      .Body = "Displayed by using the Display method of the item"
      .Display
   End With
   Set fdrCalendar = Nothing
   Set itmAppointment = Nothing
End Sub
```

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

6. Select one of the three new functions you have just built (*ThisOutlookSession.ShowExplorers*, *ThisOutlookSession.ShowContactsInspector*, or *ThisOutlookSession.ShowAppointmentInspector*) and click Run. *ShowExplorers* displays the Contacts Explorer; *ShowContactsInspector* displays the Contacts Inspector, which allows you to add a new contact; and *ShowAppointmentInspector* displays the Appointment Inspector, which allows you to enter a new calendar appointment.

TROUBLESHOOTING

Your macro doesn't show up

If you've written a macro in VBA, and it doesn't appear when you open the Macro dialog box (press Alt+F8 from the main Outlook window), you can perform a number of checks:

- Only Sub procedures appear in the Macro dialog box. Make sure that the procedures you've written begin with Sub <*name*> rather than Function <*name*>.
- The Sub procedures must be public. If your procedure is declared as Private Sub <*name*>, it will not appear.
- Only Sub procedures declared in either the *ThisOutlookSession* module or a code module can be seen in the Macro dialog box. Even public Sub procedures declared in classes cannot be seen, as you must instantiate the class to use them. The same is true of Sub procedures declared in forms.

Creating an Outlook Application with VBA

Now that you have seen how the VBA Editor works and how the Outlook object model allows you to interact with Outlook, it's time to discuss how you can use this knowledge to build an application in Outlook. You can enhance Outlook in many ways by developing a custom Outlook application. One way is to use the features of Outlook to perform tasks that are not specifically Outlook tasks but can be implemented in an Outlook interface. For example, you might enhance the Outlook calendar system so that users could record information—in your company's time-entry system—about the work they perform. If users already record their appointments for the day using the calendar, you could add custom application code that examines this information and exports it as data for your time-entry system, which means that users wouldn't have to enter the information twice.

Another way to benefit from building custom Outlook applications is to automate tasks in Outlook that would otherwise require users to perform a significant amount of work. For example, users often have hundreds of contacts stored in their copy of Outlook. If they deal with large companies, they might have many contacts for a single company. If that company rebrands itself with a new name, all the contacts for the company would need to be updated.

The rest of this chapter looks at adding custom Outlook functionality to automate the process involved in the example just mentioned. The application consists of a number of elements:

- Outlook data-access functions to manipulate contact details
- Two custom forms for entering information and reviewing details
- A custom toolbar and button to give the user access to the functionality
- A printing function so that users can review the information

Accessing Outlook Data

One of the main reasons for adding custom VBA code to Outlook is to allow access to some of the data stored in Outlook. Using the objects described earlier, you can easily access data stored in Outlook and then utilize it in another form. In this example, you need to update the company name of all contacts from a specific company because the company has rebranded itself.

The following code has two strings that represent the original company name and its new name. For each item where the company name matches the *strFrom* variable, change the name to the string contained in *strTo* and call the *Save* method of the item:

```
Public Sub UpdateCompanyName(ByRef strFrom As String, ByRef strTo As String)
   Dim fdrContacts As Outlook.MAPIFolder
   Dim objContactItem as Outlook.ContactItem 'Create an instance of the Contacts
      folder.
   Set fdrContacts = Application.GetNamespace("MAPI") _
      .GetDefaultFolder(olFolderContacts) 'Loop through all contact items checking
         the CompanyName and 'changing it if necessary.
   For Each objContactItem In fdrContacts.Items
      If objContactItem.CompanyName = strFrom Then
         objContactItem.CompanyName = strTo
         objContactItem.Save
      End If
   Next
End Sub
```

The preceding method looks at every contact item individually. This is not very efficient, however, especially if you have a large number of contacts. To improve efficiency, try using the following code in place of the *For...Each* loop. Rather than checking every item, this code uses the *Restrict* method of the *Items* collection of the folder object, which lets you apply a filter to the items before working with them. In this case, you should include only items in which the *CompanyName* property is equal to the value of the *strFrom* variable passed to the procedure. For a contacts list of 400 items, this reduces the processing time from 4 seconds to 1 second:

```
'Loop through the appropriate Contact items changing the Company Name.
For Each objContactItem In _
   fdrContacts.Items.Restrict("[CompanyName] = '" & strFrom & "'")
   objContactItem.CompanyName = strTo
   objContactItem.Save
Next
```

Adding a Form

The procedure in the preceding section converts one string value to another. One way to find out what these values are is to ask the user by adding a custom form to the application. To add a form to the project, right-click in the Project window and choose Insert, UserForm. The new form is displayed along with its properties and the Toolbox of controls. Add controls to the form until it looks like the form shown in Figure A2-3.

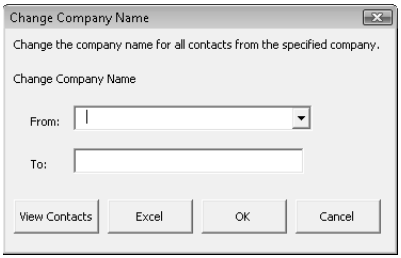

Figure A2-3. Use this VBA form to tell the program the original and changed name of the company.

Add the following code to the form's Code window. It calls the *Load* function to instantiate the form and passes the *strFrom* and *strTo* variables to the *UpdateCompanyName* procedure:

```
Dim bContinue as Boolean
Public Function Load(strFrom As String, strTo As String) As Boolean
   'Initialize the cancel boolean.
   bContinue = True
   BuildComboList 'Show the form.
   frmCompanyNameChange.Show
   If bContinue Then
      strFrom = cmbFrom
      strTo = txtTo
   End If
   Load = bContinue
End Function
```

This code maintains a *bContinue* Boolean value that stores information about whether to continue processing code when the form exits. It is also responsible for calling a function to fill the combo box with current company names and for displaying the form. The code to build the combo box list of existing companies follows:

```
Private Sub BuildComboList()
   Dim fdrContacts As Outlook.MAPIFolder
   Dim objContact As Outlook.ContactItem
   Dim strCompanyName As String 'Create an instance of the Contacts folder.
   Set fdrContacts = Application.GetNamespace("MAPI") _
   .GetDefaultFolder(olFolderContacts)
   fdrContacts.Items.Sort "[CompanyName]", False 'Loop through the contact items
       extracting unique company names.
   For Each objContact In fdrContacts.Items
   If Trim(objContact.CompanyName) <> Trim(strCompanyName) Then
       cmbFrom.AddItem
       objContact.CompanyName
       strCompanyName = objContact.CompanyName
   End If
   Next
End Sub
```

This code uses the *Sort* method of the folder's *Items* collection to arrange the company names in order. This permits the function to loop through everything and discard duplicate values.

Finally, to allow you to execute the function you've just created from within Outlook, add the following procedure, which is associated with the toolbar button. This procedure calls the *Load* method of the *CompanyNameChange* form and then calls the *UpdateCompanyName* procedure to change the company name:

```
Public Sub ChangeCompany()
   Dim strFrom As String
   Dim strTo As String
   If frmCompanyNameChange.Load(strFrom, strTo) Then
       UpdateCompanyName strFrom, strTo
   End If
End Sub
```


Creating Custom Toolbar Buttons

Now that you've written some code, you need a way to let users access the functions. Some of this can be accomplished automatically by events that occur within Outlook, but often you need to add elements to the main Outlook interface to allow users to execute a piece of the code. A great way to do this is to add command bars and toolbar buttons to the Outlook application.

Two options are available for customizing command bars. The first is to add and change the command bars from the Outlook interface, as demonstrated in Chapter 26, "Customizing the Outlook Interface." This is a good technique if the changes you're making are to your own copy of Outlook and they are permanent.

The second technique is to add the command bars using VBA. This gives you greater control over when the toolbars are added and removed and is particularly useful if you want a command bar or toolbar buttons to appear only at specific times. To add a command bar to the Outlook interface, you can use the following code:

```
Dim tlbCustomBar As CommandBar
Set tlbCustomBar = Application.ActiveExplorer.CommandBars _
   .Add(Name:="Custom Applications", Position:=msoBarTop, _
   Temporary:=True)
tlbCustomBar.Visible = True
```

This code declares an object of type *CommandBar* that will be used locally to perform commands on the command bar. To create a new command bar, you use the *Add* method of the *CommandBars* collection; this is a property of an *Explorer* object. To customize the new command bar, you can pass a number of parameters to the *Add* method. *Temporary:=True* tells the command bar to exist only as long as Outlook is open. If you were to close Outlook and reopen it again without rerunning this procedure, the command bar would no longer exist.

After you've added a command bar to the environment, you also need to add buttons. Follow three basic steps when adding buttons to a command bar:

1. Add the button to the command bar.
2. Associate the button with a function.
3. Format the button.

To add a button to the command bar, use the following code:

```
Dim btnNew As CommandBarButton
Set btnNew = tlbCustomBar.Controls.Add(Type:=msoControlButton)
```

You can associate a button with either the operation of any other button or menu in Outlook or with a custom function you've written in VBA. To associate the button with a built-in function, you must pass another parameter to the *Add* method:

```
Set btnNew = tlbCustomBar.Controls.Add(Type:=msoControlButton, _
   Id:=Application.CommandBars("Edit").Controls("Paste").ID)
```

This example adds a button to the custom toolbar that will perform the *Paste* function. The *ID* property takes an integer value, which in this case is retrieved from an existing control; you can achieve the same result by using *id:=22*.

If you want to allow the toolbar button to call a custom VBA procedure, you must set the *OnAction* property. Ensure that your function is declared as public and then enter the following code after creating the button to associate it with the named function:

```
Set btnNewCustom = _
  tlbCustomBar.Controls.Add(Type:=msoControlButton)
btnNewCustom.OnAction = "ChangeCompany"
```

This code associates the new button with a procedure called *ChangeCompany*.

Finally, when adding a button, you can format it by setting its style, giving it a caption, and perhaps giving it an icon. Buttons can have many styles that determine how they are displayed. To change the style, use the *Style* property:

```
btnNewCustom.Style = msoButtonIconAndCaption
```

To add a caption, use the *Caption* property. The *Caption* property specifies the text that is displayed on the button (if the chosen button style has a caption):

```
btnNewCustom.Caption = "Change Company Name"
```

Printing

As a final enhancement to the process of changing the company name for all contacts of that company, you can implement some print functionality. This will allow the users to view all the contacts that will be affected before the operation takes place and will give them the option to print a list of names, an individual contact detail, or all the contact details.

When using code to print information from Outlook, remember that all print methods use the default printer of the machine on which they are being performed. If there is no default printer or if the printer is unavailable, an error occurs. For this example, you can create the VBA form shown in Figure A2-4.

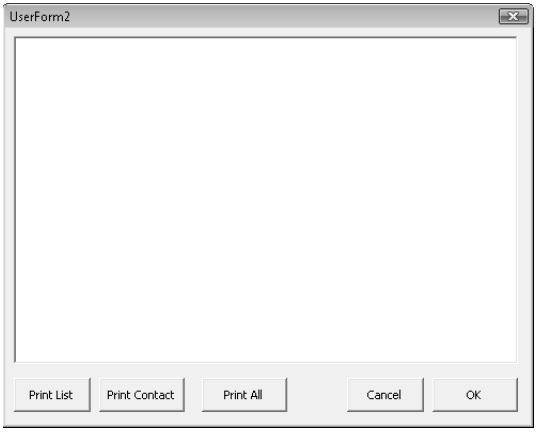

Figure A2-4. This form allows users to see and print details of all the contacts affected by the company name change.

The list box contains a list of the contact names affected by the company name change. The buttons allow the user to print the list (by printing the form), to print contact details for one contact, or to print all contacts. To print the form, you use the form's *PrintForm* method. Place the following code in the button click event handler. *Me* refers to the current form:

```
Me.PrintForm
```

The following function prints contact information. This code takes a contact name as a string and then uses the *Find* method of the Contacts folder to return a *ContactItem* object. The *PrintOut* method of the item is used to print the object:

```
Private Sub PrintContact(strContactFullName As String)
   Dim fdrContacts As Outlook.MAPIFolder
   Dim objContactItem As Outlook.ContactItem
   Dim strFullName As String 'Create an instance of the Contacts folder.
   Set fdrContacts = Application.GetNamespace("MAPI") _
     .GetDefaultFolder(olFolderContacts)
   Set objContactItem = _
     fdrContacts.Items.Find _
     ("[FullName] = '" & strContactFullName & "'") objContactItem.PrintOut
End Sub
```

To complete this form, use the following code to fill the list box with relevant items:

```
Public Sub LoadList(strCompany As String)
   Dim fdrContacts As Outlook.MAPIFolder
   Dim objContactItem As Object
   Dim i As Integer
   Dim arrContacts() As String
   'Create an instance of the Contacts folder
   Set fdrContacts = Application.GetNamespace("MAPI") _
       .GetDefaultFolder(olFolderContacts)
   For Each objContactItem In fdrContacts.Items _
     .Restrict("[CompanyName] = '" & strCompany & "'")
         If TypeOf objContactItem Is Outlook.ContactItem Then
             i = i + 1
             ReDim Preserve arrContacts(2, i)
         End If
      arrContacts(0, i - 1) = objContactItem.FullName
   Next
   lstAffectedContacts.Column = arrContacts
End Function
```

This function again creates a folder using the *Restrict* method, but this time it loops through the returned contacts and builds an array of contact names that can be used as the list for the list box.

To add code that changes the names of contacts for a company, 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 Projects window, select Project1.
3. Choose Insert, Module to add a custom module to the project.
4. Select the new module in the Project window, and then go to the Properties window. (Press F4 to show the window if it is not visible.)
5. Change the module name to basChangeCompanyName.
6. Press F7 to open the Code window for the module.
7. Enter the following code in the Code window:

```
Public Sub UpdateCompanyName(ByRef strFrom As String, ByRef strTo As String)
Dim fdrContacts As Outlook.MAPIFolder
Dim objContactItem As Outlook.ContactItem

'Create an instance of the Contacts folder
Set fdrContacts = Application.GetNamespace("MAPI") _
     .GetDefaultFolder(olFolderContacts)
```

```
'Loop through the appropriate Contact items changing the Company Name
For Each objContactItem In _
              fdrContacts.Items.Restrict("[CompanyName] = '" & _
              strFrom & "'")
              objContactItem.CompanyName = strTo
              objContactItem.Save
       Next
End Sub

Public Function ChangeCompany()
   Dim strFrom As String
   Dim strTo As String
   If frmCompanyNameChange.Load(strFrom, strTo) Then
      UpdateCompanyName strFrom, strTo
   End If
End Function

Sub AddToolbar()
   Dim tlbCustomBar As CommandBar
   Dim btnNew As CommandBarButton
   Dim btnNewCustom As CommandBarButton
   Set tlbCustomBar = Application.ActiveExplorer _
     .CommandBars _
     .Add(Name:="Custom Applications", _
     Position:=msoBarTop, Temporary:=True)
   tlbCustomBar.Visible = True
   Set btnNew = _
   tlbCustomBar.Controls.Add(Type:=msoControlButton)
   Set btnNew = _
      tlbCustomBar.Controls.Add(Type:=msoControlButton, _
      ID:=ActiveExplorer.CommandBars("Edit") _
     .Controls("Paste").ID)
   Set btnNewCustom = _
      tlbCustomBar.Controls.Add(Type:=msoControlButton)
   btnNewCustom.OnAction = "ChangeCompany"
   btnNewCustom.Style = msoButtonIconAndCaption
   btnNewCustom.Caption = "Change Company Name"
End Sub
```

8. Add a custom form to the project by choosing Insert, UserForm.
9. Drag four label controls, a combo box control, a text box control, and two buttons to this form. Refer back to Figure 44-3 to see how to arrange them and change the captions for these controls as appropriate. Name them as follows:
   - Name the combo box control cmbFrom.
   - Name the text box control txtTo.
   - Name the OK button cmdOK.
   - Name the Cancel button cmdCancel.
10. Name the form frmCompanyNameChange.

11. Press F7 to switch from Form view to Code view, and then enter the following code in the Code window:

```
Dim bContinue As Boolean
Public Function Load(strFrom As String, strTo As String) As Boolean
   'Initialize the cancel boolean.
   bContinue = True
   BuildComboList
   'Show the form
   frmCompanyNameChange.Show
   If bContinue Then
      strFrom = cmbFrom
      strTo = txtTo
   End If
   Load = bContinue
End Function

Private Sub BuildComboList()
   Dim fdrContacts As Outlook.MAPIFolder
   Dim objContact As Outlook.Object
   Dim strCompanyName As String
   'Create an instance of the Contacts folder.
   Set fdrContacts = Application.GetNamespace("MAPI") _
     .GetDefaultFolder(olFolderContacts)
   fdrContacts.Items.Sort "[CompanyName]", False
   'Loop through the contact items extracting unique company names
   For Each objContact In fdrContacts.Items
      If TypeOf objContact Is Outlook.ContactItem Then
         If Trim(objContact.CompanyName) <> Trim(strCompanyName) _
           Then
              cmbFrom.AddItem
              objContact.CompanyName strCompanyName = objContact.CompanyName

         End If
      End If
   Next
End Sub

Private Sub cmdCancel_Click()
 bContinue = False
     Unload Me
End Sub

Private Sub cmdOK_Click()
  bContinue = True
     Unload Me
End Sub
```

12. Add a second custom form to the project by choosing Insert, UserForm.

13. Drag a list box control and five buttons to this form. Arrange them and change their captions so that they look like the form shown in Figure A2-4. Name them as follows:

- Name the list box control *lstAffectedContacts*.
- Name the Print List button *cmdPrintList*.
- Name the Print Contact button *cmdPrintContact*.
- Name the Print All button *cmdPrintAll*.
- Name the Cancel button *cmdCancel*.
- Name the OK button *cmdOK*.

14. Name the form frmPrintContacts.

15. Press F7 to switch from Form view to Code view, and then enter the following code in the Code window:

```
Private Sub cmdCancel_Click()
   Unload Me
End Sub

Private Sub cmdOK_Click()
   Unload Me
End Sub

Private Sub cmdPrintAll_Click()
  Dim i As Integer
For i = 0 To lstAffectedContacts.ListCount - 1
      PrintContact lstAffectedContacts.List(i)
      Next i
End Sub

Private Sub cmdPrintContact_Click()
   PrintContact lstAffectedContacts
End Sub

Private Sub cmdPrintList_Click()
   Me.PrintForm
End Sub

Private Sub PrintContact(strContactFullName As String)
   Dim fdrContacts As Outlook.MAPIFolder
   Dim objContactItem As Outlook.ContactItem
   Dim strFullName As String
   'Create an instance of the Contacts folder.
   Set fdrContacts = Application.GetNamespace("MAPI") _
     .GetDefaultFolder(olFolderContacts) Set objContactItem = _
     fdrContacts.Items.Find("[FullName] = '" & _
     strContactFullName & "'")
   objContactItem.PrintOut
   Set objContactItem = Nothing
End Sub
```

```
Public Sub LoadList(strCompany As String)
Dim fdrContacts As Outlook.MAPIFolder
Dim objContactItem As Outlook.ContactItem
  Dim i As Integer
Dim arrContacts() As String
'Create an instance of the Contacts folder
Set fdrContacts = Application.GetNamespace("MAPI") _
   .GetDefaultFolder(olFolderContacts)
For Each objContactItem In fdrContacts.Items _
   .Restrict("[CompanyName] = '" & strCompany & "'")
       If TypeOf objContactItem Is Outlook.ContactItem Then
          i = i + 1
          ReDim Preserve arrContacts(2, i)
          arrContacts(0, i - 1) = objContactItem.FullName
       End If
   Next
   lstAffectedContacts.Column = arrContacts
End Sub

Public Function Load(strCompany As String)
   LoadList strCompany
  'Show the form.
   frmPrintContacts.Show
End Function
```

16. Add another button to the frmCompanyNameChange form with the caption View Contacts and the name cmdViewContacts. You use this button to open the second form.

17. Double-click View Contents to open the Code window, and enter the following code:

```
Private Sub cmdViewContacts_Click()
frmPrintContacts.Load cmbFrom
End Sub
```

18. Go back to the main Outlook window and choose Tools, Macro, Macros. Select AddToolbar from the list and click Run. Two new toolbar buttons should now be visible in Outlook. One has a Paste icon and the other is labeled Change Company Name.

19. Click the second button, and your first form is displayed.

20. Select a company in the box, type the new company name in the text box, and then click OK. The company names of all contacts are changed.